An Efficient Design of Intelligent Network Data Plane

Guangmeng Zhou, *Tsinghua University;* Zhuotao Liu, *Tsinghua University and Zhongguancun Laboratory;* Chuanpu Fu, *Tsinghua University;* Qi Li and Ke Xu, *Tsinghua University and Zhongguancun Laboratory*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhou-guangmeng

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

An Efficient Design of Intelligent Network Data Plane

Guangmeng Zhou[1] Zhuotao Liu[1,2] Chuanpu Fu[1] Qi Li[1,2] Ke Xu[1,2]
[1] *Tsinghua University* [2] *Zhongguancun Laboratory*

Abstract

Deploying machine learning models directly on the network data plane enables intelligent traffic analysis at line-speed using data-driven models rather than predefined protocols. Such a capability, referred to as *Intelligent Data Plane (IDP)*, may potentially transform a wide range of networking designs. The emerging programmable switches provide crucial hardware support to realize IDP. Prior art in this regard is divided into two major categories: (i) focusing on extracting useful flow information from the data plane, while placing the learning-based traffic analysis on the control plane; and (ii) taking a step further to embed learning models into the data plane, while failing to use flow-level features that are critical to achieve high learning accuracies. In this paper, we propose NetBeacon to advance the state-of-the-art in both model accuracy and model deployment efficiency. In particular, NetBeacon proposes a multi-phase sequential model architecture to perform dynamic packet analysis at different phases of a flow as it proceeds, by incorporating flow-level features that are computable at line-speed to boost learning accuracies. Further, NetBeacon designs efficient model representation mechanisms to address the table entry explosion problem when deploying tree-based models on the network data plane. Finally, NetBeacon hardens its scalability for handling concurrent flows via multiple tightly-coupled designs for managing stateful storage used to store per-flow state. We implement a prototype of NetBeacon and extensively evaluate its performance over multiple traffic analysis tasks.

1 Introduction

Artificial Intelligence (AI) experiences increasingly popularity in various networking designs, such as video bitrate adaption [34, 56], congestion control [1, 57], traffic optimizations [11], routing [29, 65], and network planning [66]. The learned models are typically deployed either on end-hosts [1, 34, 56, 66] or network control plane [11, 29, 57, 65] to perform inference, which are equipped with flexible general-purpose processors or GPUs.

Deploying the learned models for traffic analysis directly on the network data plane, however, is a relatively less charted area. The key merit of executing model inference on the network data plane is that *it enables intelligent traffic analysis at line-speed (*i.e., *zero reduction to the network forwarding speed) using data-driven models rather than predefined protocols.* Such a capability, referred to as *Intelligent Data Plane (IDP)*, may potentially transform a wide range of networking designs. For instance, the increasingly sophisticated network attacks often bypass the empirically learned traffic filters [31], motivating the community to design learning-based malicious traffic detection mechanisms, such as [14, 37]. Unlike these approaches, however, IDP achieves line-speed traffic analysis, while the state-of-the-art that deploys learning models on the network control plane can only process traffic at roughly 10 Gbps [14]. Further, placing the control plane on the critical path of traffic analysis introduces extra reaction time [2]. Besides the security domain, other networking designs (such as differentiated routing [29], ECN threshold adjustment [57], and buffer management [19]) may also benefit from IDP.

The design of IDP used to be "nearly impossible" due to the difficulty of realizing dynamic and data-driven AI models using the specialized and protocol-defined ASIC switching chips on the network data plane. Fortunately, the advances in programmable switches (e.g., P4 switch [8]) allow customizable packet processing logic based on the Protocol-Independent Switch Architecture (PISA). Under PISA, network packets are processed, at line-speed, by a programmable pipeline consisting of a parser, multiple match/action stages and a deparser. Although the PISA architecture is not Turing Complete (yet) to execute arbitrary computation over the packet bytes, the community has proposed significant network-security related designs centering around the programmable switches.

The first category of prior art focuses on extracting useful flow information from the data plane to support overarching applications. For instance, NetWarden [53] and FlowLens [4] rely on the programmable switches to collect flow distribution information, based on which they perform covert channel detection [53] and learning-based traffic classification [4], respectively, in the control plane. Similarly, Poseidon [61] and Jaqen [32] compare the collected flow information (*e.g.,* flow rates) against predefined traffic filtering policies (*e.g.,* threshold-based filters) to mitigate volumetric DDoS attacks. The key difference between this category of designs and IDP is that they do not directly embed the learning models into the programmable data plane.

The second category of prior art takes a step further to

realize IDP. For instance, pForest [10] and SwitchTree [26] propose to deploy decision tree models on programmable switches, by mapping one match/action stage to one layer of a decision tree. However, their designs, although verified using software switch BMv2, are undeployable on actual hardware (see § A for details). Planter [63] extends IIsy [55] and proposes an architecture including both feature tables and code tables to represent ensemble trees on the data plane. Although promising, Planter lacks several critical designs: for instance, it is unclear how Planter can simultaneously use one feature multiple times (which is quite common in reality). The most recent work Mousika [51] proposes a decision tree model, namely Binary Decision Tree (BDT), designed specifically for the programmable switch data plane. However, the number of table entries required to deploy BDT faces the problem of combinatorial explosion (see details in § A). *Crucially, all these prior deployable art employs only per-packet features in their models, fully ignoring flow-level features.* As we will show in § 2.2, flow-level features are necessary to boost the learning accuracies for various traffic analysis tasks.

In this paper, we present NetBeacon, a novel IDP design that advances state-of-the-art in both model accuracy and model deployment efficiency. In particular, NetBeacon is empowered by the following innovative designs:

(i) A data plane aware learning model design centering around a multi-phase sequential model architecture. Since packets at different phases of a flow carry different flow-level states, our model performs dynamic analysis at different phases of the flow as it proceeds, reducing the errors introduced by making premature classification decisions based on a single inference model. Meanwhile, our model uses carefully designed flow-level and per-packet features computable at line-speed on the data plane to ensure deployability.

(ii) NetBeacon proposes an efficient model representation mechanism to address the entry explosion problem when expressing decision tree or forest models into data plane matching tables. Compared with the state-of-the-art, Mousika [51], NetBeacon significantly reduces the table entry consumption (up to 75% in some cases).

(iii) We further harden the scalability of NetBeacon for handling concurrent flows, by differentiating the processing logic for short flows and long flows, as well as allowing safe storage multiplexing when observing storage index collisions. This potentially allows NetBeacon to handle more concurrent flows than the total number of registers used for maintaining per-flow state.

We implement a prototype of NetBeacon using the Tofino switch as the programmable data plane, and evaluate NetBeacon extensively with three use cases. The experimental results show that NetBeacon outperforms the state-of-the-art in both traffic analysis/classification accuracy and hardware table consumption. We also quantitatively study how switch hardware (e.g., the imperfection of hashing, and future hardware upgrade) may affect NetBeacon.

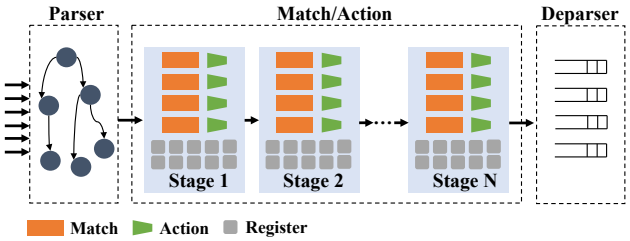


Figure 1: Protocol Independent Switch Architecture (PISA).

2 Background and Motivation

2.1 Programmable Data Plane

Traditional switches are equipped with ASIC chips customized for packet processing to achieve high-speed forwarding. Implementing new network protocols on ASIC chips therefore requires design, manufacturing, and rigorous testing from the device manufacturers, which is an expensive process. To enable agile protocol development, roughly two decades ago, the community proposed Software-Defined Networking (SDN) to allow software controllers to install customized flow entries on switches via OpenFlow protocols [36]. SDN is a tremendous success, experiencing wide deployment over years, especially, in data centers [13, 20]. Recently, the emerging programmable switch technology, centering around Protocol-Independent Switch Architecture (PISA), boosts network programmability to another level. Instead of fully relying on a software controller, the switching pipeline itself is flexible enough to allow direct programming through domain-specific programming languages, such as P4.

In PISA switching pipeline (Figure 1), a network packet first enters the parser for packet header parsing, then enters multiple match/action stages for packet manipulation, and finally reaches the deparser for packet serialization. The parser, match/action, and deparser can all be programmed to implement desired protocols. The match/action supports exact matching, ternary matching, and longest prefix matching (LPM). Each match corresponds to an action, where specific computation and storage modification can be executed. Mutually dependent actions need to be placed on different stages. Packet headers and metadata instances are stored using stateless storage that is reinitialized as new packets arrive. PISA also provides stateful and persistent storage, such as counters, meters, and registers. Finally, PISA provides various mechanisms (such as resubmit, recirculation, packet generation) to further extend programming capabilities.

Despite its flexibility, PISA has the following computation and storage constraints. First, it supports boolean, shift, add and subtract operations, but not multiplication and division. Float operations, loop operations, and complex conditional operations are not supported as well. The main calculation logic is implemented using the match/action stages, which are not unlimited (*e.g.,* Tofino 1 has 12 stages). Similarly, the storage resources are finite, *e.g.,* on Tofino 1, the SRAM of each

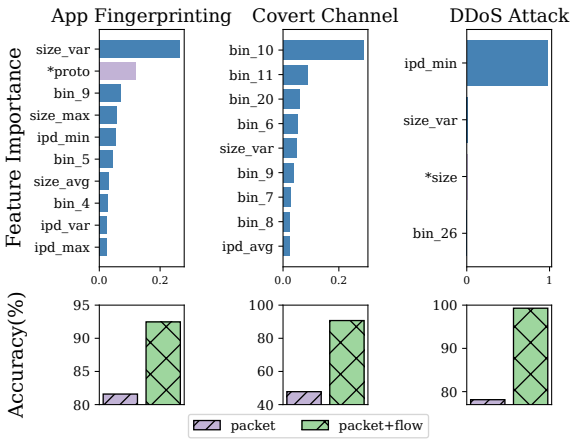


Figure 2: The necessity of flow-level features in traffic analysis tasks. Features started with ∗ are per-packet features, such as ∗proto and ∗size.

pipeline is 120MB and the TCAM is 6.2MB. Probably, the most "surprising" storage limitation is that each register can only be accessed once when a packet traverses the switching pipeline. As a result, operations such as read-and-then-update registers are not supported natively.

2.2 Motivation

The flexibility of PISA inspired significant research in realizing the potential of intelligent data plane (IDP). We divide prior art into three subcategories, as summarized in Table 1. First, NetWarden [53] and FlowLens [4] represent the designs using programmable switches to collect useful flow information, such as inter-packet delay distributions, and then perform traffic analysis on the control plane. Due to the interaction latency between the data plane and the control plane, traffic is not analyzed at line-speed.

Poseidon [61] and Jaqen [32] share the similar designs of NetWarden [53] and FlowLens [4], except that the collected flow information can be used directly on the data plane to classify volumetric DDoS attacks at line-speed (note that some of their actions still involve servers in the control plane). However, their classification logic is based on threshold-driven traffic filters, instead of machine learning models. Therefore, their approaches are not generalizable to the use cases where the traffic analysis logic cannot be accurately represented as hand-crafted filters.

Planter [63] and Mousika [51] take the initial steps to embed decision tree models in the network data plane by representing decision tree branches using the match/action tables. However, their designs have two major limitations: low efficiency in representing learning models on the data plane as demonstrated in § 7.3 and merely considering stateless per-packet features while ignoring stateful flow-level features.

We experimentally show the necessity of flow-level features (flow identified by its 5-tuple) for boosting the accuracy

Prior Works	Line-Speed	Accuracy / Generalization	
		Learning-Based	Flow-Aware
NetWarden[53], FlowLens[4]	✗	✓	✓
Poseidon[61], Jaqen[32]	✓¶	✗	✓
Planter[63], Mousika[51]	✓	✓	✗
NetBeacon	✓	✓	✓

¶ Some defense/detection polices or logic of Poseidon and Jaqen require control plane involvement. In such cases, they cannot be processed at line-speed.

Table 1: Comparison with prior art in IDP.

of learning-based traffic analysis. We first show that flow-level features are, in general, *more important* than per-packet features. To this end, we considered three scenarios: P2P application fingerprinting, covert channel detection, and DDoS attack detection. For each task, we train a decision tree model using both per-packet features (*e.g.,* ttl, packet size of n-th packet) and flow-level features (*e.g.,* the mean and variance of packet size of the first n packets). Then we plot the feature importance obtained based on the information gain of the feature in the top half of Figure 2. Flow-level features clearly show up with higher importance scores.

Second, we show flow-aware classifiers output more accurate results than flow-agnostic classifiers. In particular, we train two decision tree models: one flow-agnostic model that uses only per-packet features as in Mousika [51], and one flow-aware model that uses both flow-level and per-packet features, sorted by their importance. Both models use the same number of features. We report the classification accuracy results in the bottom half of Figure 2. The results demonstrate that incorporating flow-level features significantly improves accuracies: +11% in P2P application fingerprint, +43% in covert channel detection, and +21% in DDoS attack detection (the detailed descriptions about these tasks are given in § 7.1).

Design Goals. Compared with prior art (Table 1), we design NetBeacon to advance the state-of-the-art by simultaneously achieving line-speed and highly-accurate (learning-based) traffic analysis on the network data plane. Other art [45, 50] use SmartNICs instead of programmable switches as the data plane. These two types of hardware have drastically different characteristics: programmable switches have much higher throughput, with much limited computation capability. We focus on PISA-driven IDP design in this paper.

Assumptions, Threat Model, and Limitations. We assume that NetBeacon has access to task-related training datasets. Machine learning itself has security vulnerabilities, such as data and model poisoning attacks, which is out of the scope. We assume that the programmable switches hosting NetBeacon are secure. Since flow-level features consume the stateful storage on programmable switches, the number of concurrent flows for which NetBeacon can simultaneously maintain per-flow state is limited by hardware storage, although NetBeacon has dedicated designs to improve scalability. Meanwhile, it is possible to scale out traffic analysis capacity by deploying multiple NetBeacon instances in parallel. NetBeacon does not support flow-level features that are difficult to compute on the data plane (*e.g.,* percentile of packet sizes). We use the 5-tuple

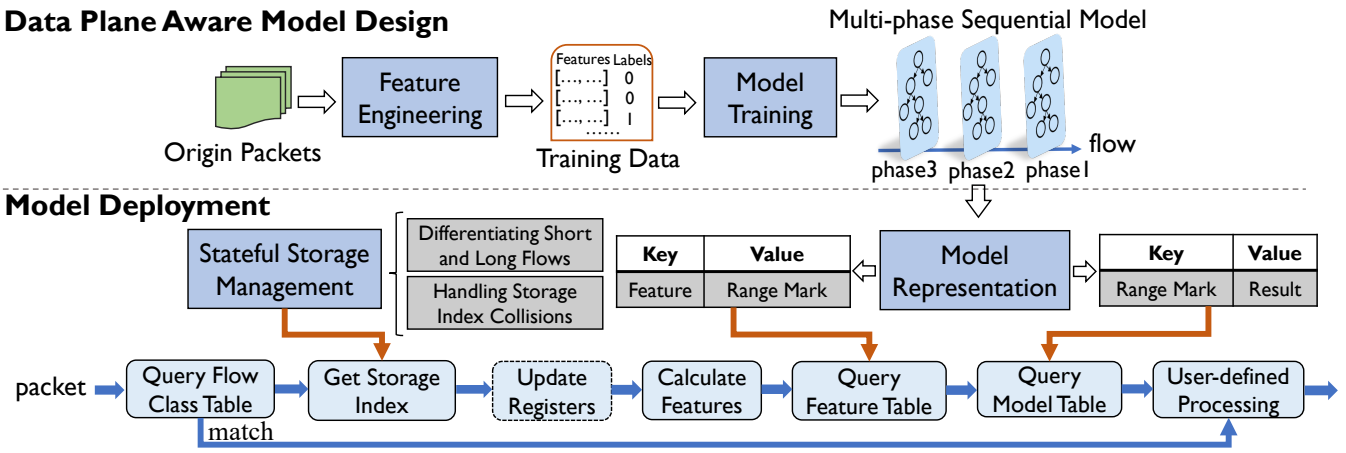


Figure 3: The architecture of NetBeacon.

(*i.e.,* src/dst IP, src/dst port, and protocol) to identify a flow, yet NetBeacon itself is not limited by any flow definition.

3 The Overview of NetBeacon

As shown in Figure 3, architecturally, NetBeacon is designed around two major components: data-plane aware model design and efficient model deployment. Data plane aware model design is a *co-design* approach to generate hardware-friendly learning models. Towards this end, our feature engineering relies on features that are extractable or computable at line-speed on the switching pipeline. Further, considering that flow-level features (*e.g.,* the mean of packet sizes) are changing as a flow proceeds, NetBeacon proposes a multi-phase sequential model architecture that can make multiple inference decisions as the flow proceeds, until the system is sufficiently confident to make the final decision.

The key design of the model deployment is the model representation module. It translates the learned models into multiple feature tables and one model table on the data plane, where the feature tables encode feature values as data structures named *range marks*, which are further mapped to the inference results stored in the model table. NetBeacon designs efficient coding mechanisms to greatly reduce the table entry consumption when representing models.

In addition, NetBeacon designs a stateful storage management module to achieve efficient per-flow state management on the data plane. On the one hand, this module enables NetBeacon to process short flows using purely per-packet features (*i.e.,* no per-flow state maintained for short flows), where short flows are classified using a learning model. On the other hand, NetBeacon exploits the hardware hashing to achieve storage multiplexing. In particular, when the 5-tuple of a new flow is hashed to occupied registers (*i.e.,* storage collision), the new flow can take this register if the stored flow is class-determined or timeout; otherwise, NetBeacon falls back to use stateless per-packet features for the new flow. If the packet belongs to the stored flow, the registers are updated and features are calculated for model inference, *i.e.,* query feature table and model table.

Feature Type		Definitions
Per-packet	–	packet size, ttl, protocol, etc
Flow-level	Aggregate	$F = \text{aggr}(a, c, d)$
	Summary	max/min, mean, variance, etc

Table 2: Features in NetBeacon.

Once the inference results for a packet are determined, users can design customized post-processing based on the results, such as making a binary decision of drop or allow, or assigning fine-grained different service priorities accordingly.

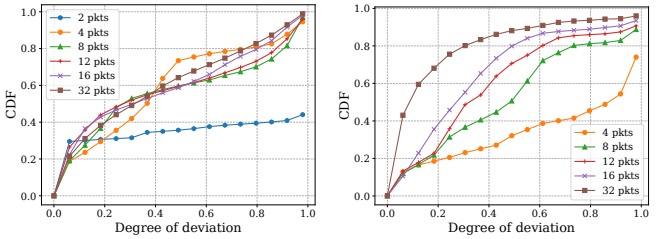
4 Data Plane Aware Model Design

4.1 Feature Engineering

The decision tree learning models in NetBeacon can use both per-packet features and flow-level features. Per-packet features can be obtained from individual packets, usually based on the fields in the packet header, such as packet size and Time To Live (TTL) values. Flow-level features are obtained by combining attributes from other packets in the same flow.

We categorize flow-level features into aggregate features and summary features. An aggregate feature is expressed as $F = \text{aggr}(a, c, d)$, where a denotes the attribute considered in F, c is the condition imposed on the attribute, and d represents the predetermined rule to update the value of F once a packet satisfies c. For example, feature $F = \text{aggr}(\textit{packet size}, [96, 112), +1)$ records the number of packets in a flow, whose packet sizes are within $[96, 112)$. Unlike aggregate features, the computation of summary features cannot be easily represented via predetermine update rules. Representative summary features are maximum/minimum, mean, and variance, which even involve multiplications or divisions that are not natively supported on hardware. We specify how summary features are computed in § 6.1.

We summarise the features considered in NetBeacon in Table 2. For a specific task, we select top features based on



(a) PeerRush, mean of IPD (b) ISCXVPN, variance of packet size

Figure 4: The CDF of the deviation degree of summary features at different flow phases. If the final feature value and the value at current phase (*i.e.,* first n packets) are a and b, respectively, the deviation degree is $abs(a-b)/max(a,b)$.

their importance calculated based on information gain.

4.2 Multi-Phase Sequential Models

Similar to prior art [26, 51, 63], NetBeacon sticks to decision tree based learning models because their constructions are more similar to the match/action operations on the data plane, compared with the neural networks that involve significant non-linear computations. We adopt the state-of-the-art decision tree forest models, *i.e.,* Random Forest (RF) and XGBoost (XGB). Considering that the number of table entries on hardware is limited, we can control model size by limiting the number of trees in the forest, the maximum tree depth, the maximum number of leaves, and so on.

Unlike the per-packet analysis where all packets are treated equally, packets in the flow-level analysis are located at different phases of the flow and therefore carry different flow-level states. Therefore, the flow-level features are dynamic as a flow proceeds. In Figure 4, we quantitatively visualize the dynamism of two flow-level features (*i.e.,* mean of inter-packet delay IPD, and variance of packet sizes) at different phases of flows, using two datasets. Clearly, both features experience significant changes over time. As a result, using a single model to perform inference based on flow-level features yields low accuracies, as shown in Figure 5.

The above observation motivates us to design a multi-phase model architecture to apply different models at different phases of flows. At each phase, NetBeacon uses the features computed at that phase for both training and inference, *i.e.,* flow-level features at the n-th packet are computed based on the first n packets. The packets where our model makes inference decisions are referred to as *inference points*. The exact arrangement of inference points is task-dependent. In particular, each inference-point essentially represents an analysis result for a flow after our model processes the n packets before the inference point. Thus, the inference points could be placed either uniformly or specifically, according to the task. For instance, if a model uses variance as features, the inference points should be placed at power-of-2 positions due to hardware limits (as detailed in § 6.1). The packets not

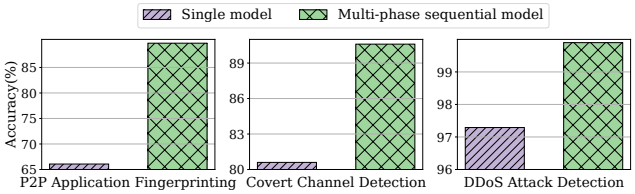


Figure 5: Single model vs. multi-phase sequential model.

selected as inference points use the inference result of its previous and closest inference point. Thus, the interval between two inference points should not be too large.

NetBeacon sets a *determination threshold* for each inference phase. When the classification probability at a specific inference point is above the corresponding determination threshold, it indicates that the multi-phase sequential model is confident to pre-decide the flow's class without using the subsequent inference points. This design facilitates the stateful storage management on the data plane (as described in § 5.2). In addition to flow-aware learning models, NetBeacon employs a flow-agnostic model to classify packets that cannot be analyzed using flow-level features, as detailed in § 5.2.

5 Model Deployment

5.1 Data Plane Model Representation

The individual models in our multi-phase sequential model architecture and the flow-agnostic tree model are represented in the same way. In this section, unless otherwise noted, we present how to represent a single model (*i.e.,* either a single decision tree model or a forest model with multiple decision trees) on the data plane. In § 6.2, we show how to merge the model representations of multi-phase sequential models.

We start with the data plane representation of a single decision tree. In a decision tree, the leaf nodes represent the classification results, and a path from the root node to a leaf node represents the matching rules for that leaf node, which is typically a concatenation of multiple feature ranges. For example, the path to leaf node 1 in Figure 6 is $f1 \in [0,25)$, $f2 \in [0,46)$ and $f3 \in [0,10)$. For features that do not appear on the path, its range is the maximum allowed range. For example, the path to node 5 is $f1 \in [65,103)$, $f2 \in [0,256)$ and $f3 \in [10,256)$. Therefore, if range matching were supported on programmable switches, a decision tree model can be easily implemented as the Model Table (1) in the Figure 6, where the key is a concatenation of multiple feature ranges, and the value is the leaf node.

Unfortunately, range matching is not universally supported on programmable switches, and it has to be coded into ternary matching (bits in a ternary entry are 0, 1, or $*$). Based on the classic prefix method [46], $[65,103)$ requires 8 ternary entries, range $[0,256)$ requires 1 ternary entries, and range $[10,256)$ requires 6 ternary entries. As a result, in order to simultaneously satisfy all three ranges, it takes 48 ($8 \times 1 \times 6$)

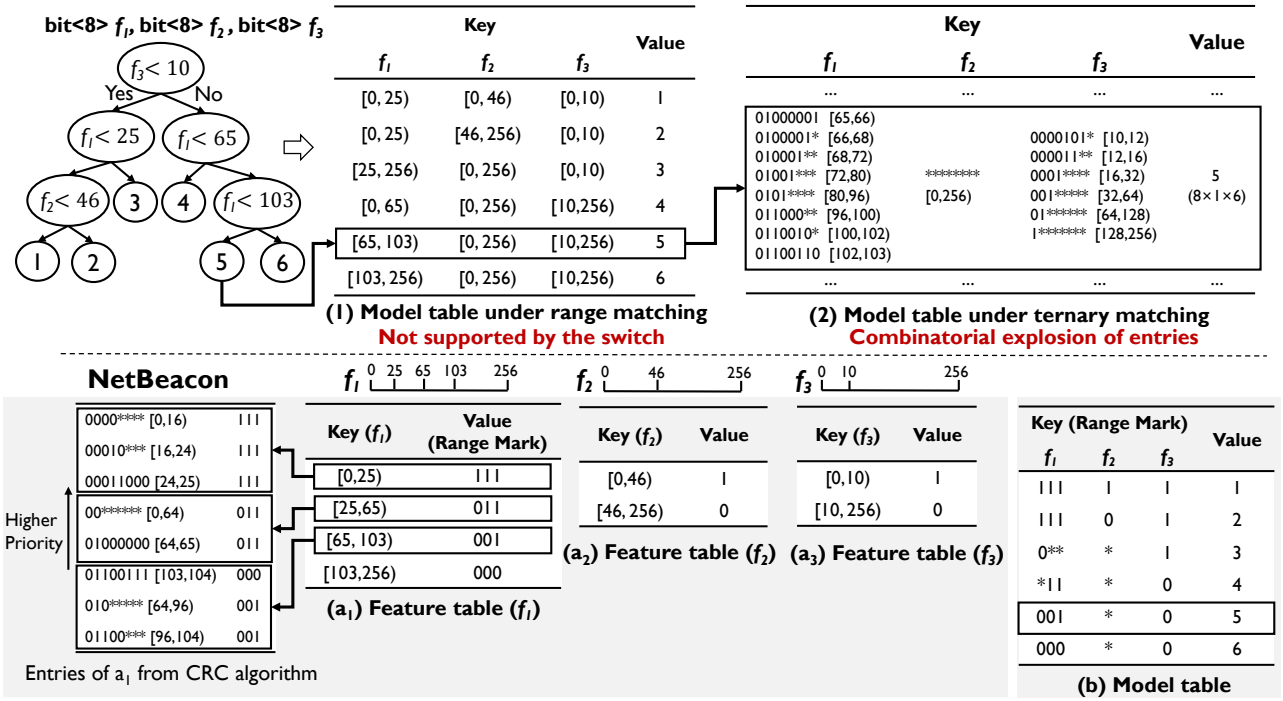


Figure 6: An example of model representation in NetBeacon. From range matching (1) to ternary matching (2), the number of table entries to represent leaf 5 increases from 1 to 48 ($8 \times 1 \times 6$). In NetBeacon, each leaf only consumes 1 entry in the model table. The feature tables are shared by all leaves and their entries are reduced by our novel range coding algorithm (CRC).

ternary entries, as shown in the Model Table (2) of Figure 6.

Range Marking. NetBeacon solves the above entry combinatorial explosion problem by introducing a novel mechanism named *range marking*. As introduced in § 3, a decision tree model is represented as multiple *features tables* (one for each feature) and one *model table*. The range marking mechanism ensures that *each leaf node only consumes a single ternary entry in the model table*, regardless of how many feature table entries are related with this leaf node.

In a decision tree model, each feature typically has multiple thresholds. For instance, in Figure 6, feature f_1 has three thresholds: 25, 65 and 103, which divides the entire possible value range (*i.e.,* $[0,256)$) into four consecutive and non-overlapping *basis ranges*, *i.e.,* $[0,25)$, $[25,65)$, $[65,103)$ and $[103,256)$. The goal of our range marking is to represent these ranges using ternary bit strings (*i.e.,* 0, 1, and $*$ matching both 0 and 1), such that (i) each basis range is mapped to a unique bit string and (ii) any new range obtained by combining multiple consecutive basis ranges, referred to as *associative ranges*, is also mapped to a unique bit string. The reason for the second condition is that one node in the decision tree could span multiple basis ranges. For instance, the parent of node 4 in Figure 6 has an associative range ($[0,65)$) spanning two consecutive basis ranges ($[0,25)$ and $[25,65)$).

Formally, we define our range marking as follows. Consider there are n thresholds $\{r_i\}, i \in [1,n]$ diving a range $[0,\mathcal{R})$ into a set of $n+1$ consecutive and non-overlapping basis ranges $\mathcal{S} = \{[0,r_1),[r_1,r_2),...,[r_n,\mathcal{R})\}$. Our range marking algorithm $\mathcal{F}$ defines a mapping from a range (either a basis range or an associative range) to a ternary bit string satisfying

- $\forall\, i \in [1,n], \mathcal{F}\left([r_i,r_{i+1})\right) = \{0,1\}^k$;
- $\forall\, i, i+m \in [1,n], \mathcal{F}\left([r_i,r_{i+m})\right) = \{0,1,*\}^k$;
- $\forall\ i,j,l,l+m \in [1,n], \mathcal{F}\left([r_i,r_{i+1})\right) \neq \mathcal{F}\left([r_j,r_{j+1})\right) \neq \mathcal{F}\left([r_l,r_{l+m})\right)$

	Range	Range Mark
Feature Table	$[0,r_1)$	$(1)^n$
	$[r_i,r_{i+1})$	$(0)^i(1)^{n-i}$
	$[r_n,\mathcal{R})$	$(0)^n$
Model Table	$[0,r_i)$	$(*)^{i-1}(1)^{n-i+1}$
	$[r_i,r_{i+j})$	$(0)^i(*)^{j-1}(1)^{n-i-j+1}$
	$[r_j,\mathcal{R})$	$(0)^i(*)^{n-i}$

Table 3: One range marking algorithm.

Table 3 shows one range marking algorithm designed based on a intuition that give a basis range $[r_i,r_{i+1})$, set the j-th bit in its range mark as 1 (default 0) if the basis range is within $[0,r_j)$. In model table, however, its keys are associative ranges. We therefore include wild matching bit $*$ so that multiple keys in a feature table (*i.e.,* basis ranges) are collapsed into a single range mark in the model table. Consider the leaf node 4 in the Figure 6 as an example. The value of f_1 on its path span two basis ranges (*i.e.,* $[0,65)$ spans $[0,25)$ and $[25,65)$) in

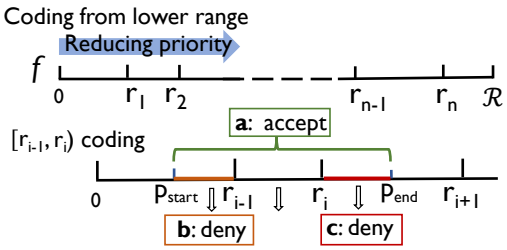


Figure 7: Illustration of consecutive ranges coding (CRC).

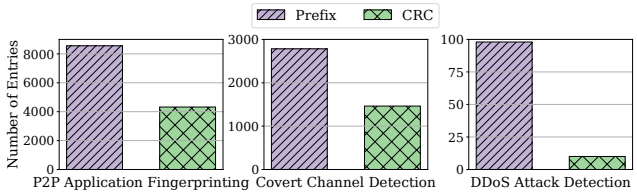


Figure 8: Comparisons between Prefix [46] and CRC.

f_1's feature table, which are concisely represented as a single ternary bit string ($*11$) in the model table.

Range Coding in Feature Tables. The keys in feature tables are still feature value ranges (*e.g.,* $[0,25)$). Although existing range coding designs (such as Prefix [46], SRGE [9], OPT [43]), Bit-Map [30], DIRPE [24]) can translate value ranges into ternary entries, they are not as efficient as our consecutive range coding (CRC) algorithm, as described below.

In our scenario, the ranges to be coded in each feature table are consecutive and cover all possible values of the feature. Based on this characteristic, we propose the following CRC algorithm. CRC starts coding from the lower basis ranges (*i.e.,* from $[0,r_1)$ to $[r_n,\mathcal{R})$). Specifically, given a basis range $[r_{i-1},r_i)$, CRC tries to find an optimal *parent range* $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$ (*i.e.,* $\mathsf{p}_{\mathsf{start}} \in [0,r_{i-1}]$ and $\mathsf{p}_{\mathsf{end}} \in [r_i,r_{i+1})$) so that it takes a minimal number of prefixes to represent the parent range on the data plane. CRC divides prefixes for representing $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$ into three parts: the acceptance prefixes representing $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$ (*i.e.,* **a** in Figure 7), and the denial prefixes for representing $[\mathsf{p}_{\mathsf{start}},r_{i-1})$ and $[r_i,\mathsf{p}_{\mathsf{end}})$, respectively (*i.e.,* **b** and **c** in Figure 7). The denial prefixes should be prioritized over the acceptance prefixes to ensure that the $[r_{i-1},r_i]$ and $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$ accept the same range.

To find the optimal $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$, CRC solves the optimization problem in Equation (1), where $\mathsf{Prefix}(R)$ is the number of prefixes required to represent range R under [46]. Note that $\mathsf{Prefix}([\mathsf{p}_{\mathsf{start}},r_{i-1}))$ is not considered in the equation. This is because that the prefixes for representing any of the lower-rank basis ranges (*i.e.,* $[r_0,r_1),\ldots,[r_{i-2},r_{i-1})$) are always prioritized over the prefixes for $[r_{i-1},r_i)$. As a result, we can safely discard the prefixes for $[\mathsf{p}_{\mathsf{start}},r_{i-1})$ since they are overshadowed anyway by those representing the lower-ranked basis ranges.

$$\min_{\substack{\mathsf{p}_{\mathsf{start}}\in[0,r_{i-1}]\\ \mathsf{p}_{\mathsf{end}}\in[r_i,r_{i+1})}} \mathsf{Prefix}([\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})) + \mathsf{Prefix}([r_i,\mathsf{p}_{\mathsf{end}})) \quad (1)$$

The intuition why CRC can reduce the total number of prefixes for representing $[r_{i-1},r_i)$ although it actually considers a larger range $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$ is that some prefixes in $[r_{i-1},r_i)$ can actually be merged into one larger prefix when expanding $[r_{i-1},r_i)$ into $[\mathsf{p}_{\mathsf{start}},\mathsf{p}_{\mathsf{end}})$. For instance, as shown in Figure 6, several smaller prefixes for $[65,103)$ (*i.e.,* 01000001, 0100001*, etc.) are merged to two larger prefixes 010***** and 01100*** after expanding $[65,103)$ into $[64,104)$.

The above optimization problem can be straightforwardly solved by enumerating all possible values of $\mathsf{p}_{\mathsf{start}}$ and $\mathsf{p}_{\mathsf{end}}$, with complexity $O(\mathcal{R}^2)$. We provide a faster solver with complexity $O((\lg\mathcal{R})^2)$, deferred to § B.

Another optimization is that CRC does not need to encode the last basis range (*i.e.,* $[r_n,\mathcal{R})$). Note that our range marking mechanism always assigns 0 to $[r_n,\mathcal{R})$. As a result, if a feature is not matched by the feature table, its range mark will take the default value 0, which is the same as the value assigned by our range marking mechanism. Thus, it is unnecessary to represent $[r_n,\mathcal{R})$ in the feature table.

In Figure 8, we quantitatively show that our CRC algorithm CRC greatly reduces entry usage compared with Prefix [46].

Handling Forest Models. One way to deploy a forest model is to represent each tree separately and aggregate the inference result of each tree to get the final result. This approach is suitable for forest models (*e.g.,* random forest models) where the final result is the plurality of the inference classes given by individual trees. In particular, a plurality matching table is appended after the model tables of individual trees. The key of each row in the plurality table is the concatenation of the inference classes of individual trees, and its corresponding value is the majority of individual inference classes. The "combinatorial explosion" problem in this case is minor as the number of individual trees and the number of inference classes of each tree are much smaller.

However, the above design is inapplicable to gradient boosting tree models (*e.g.,* GBDT, XGBoost, LightGBM). Because the final inference probability is obtained by aggregating the results from individual trees via a non-linear function (*e.g.,* Sigmoid), which is difficult to execute on the data plane.

Therefore, instead of representing individual trees separately, NetBeacon merges their model representations. Specifically, given a feature f_1, each individual tree may have a feature table for it. Merging these feature tables is the same as creating a new feature table considering all the feature value ranges that appeared in these tables, using our range marking algorithm. As for the model table, each entry represents *one combination of the leaves* from individual trees. Thus, the key of a model table entry is the range mark obtained by considering all the nodes (representing feature ranges) on the paths to the leaves associated with this entry, and the corresponding value is the aggregate (*e.g.,* Sigmoid) of these leaves, which can be computed offline. For instance, the combination of leaf 1-2 and leaf 2-1 in Figure 9 merges the $f_1 \in [25,256)$ and $f_1 \in [0,35)$ forming $f_1 \in [25,35)$. And the value of the combination is Sigmoid(-0.01+0.5).

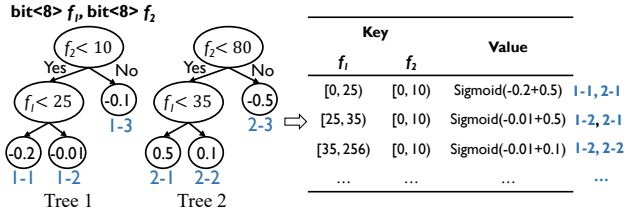


Figure 9: An example of merged representations of a forest model with two decision trees.

When exploring leaf combinations in the model table, NetBeacon removes the conflicting combinations. (i) Feature value conflict: the leaf 1-1 in Figure 9 has $f_1 < 25$ on its path and leaf 2-2 has $f_1 \geq 35$ on its path, so the combination of the above two leaves is excluded. (ii) Features semantic conflict: if one node with Max(packet_size) < 10 on one leaf path and one node with Avg(packet_size) > 20 on another leaf path, the combination of the above two leaves is excluded. By preserving all compatible leaf combinations, the above mechanism retains the accuracy of the original forest model.

5.2 Stateful Storage Management

In order to leverage flow-level features, NetBeacon relies on stateful storage to maintain the per-flow state. Prior art [4, 53] involves the control plane to allocate a non-conflicting storage index upon receiving new flows. To achieve line-speed traffic analysis, NetBeacon instead relies on hardware hashing readily available on the data plane to allocate storage indices. In particular, assume that there are N stateful registers available for storing flow state, NetBeacon computes the storage index for a flow as $\mathcal{H}$(5-tuple) % N, where $\mathcal{H}$ is a hash function.

Hash-based storage index allocation, however, has the problem of allocation collisions, *i.e.,* two different flows (with different 5-tuples) may receive the same storage index. So it is necessary to store the true flow ID (*e.g.,* the 5-tuple) alongside the storage index so that NetBeacon is aware of collisions.

Once a storage collision occurs, the per-flow states of both the original and new flows will become dirty if the new flow overwrites the storage. Therefore, NetBeacon proposes two designs to mitigate this problem: one design to reduce the overall chance of collision, and the second design to enable safe storage overwrites.

Differentiating Short and Long Flows. As introduced in § 2.2, the reason to incorporate flow-level features on top of per-packet features is to boost classification accuracy. Since the primary goal of traffic analysis is to improve the overall packet classification accuracy across all flows, maintaining per-flow states for shorter flows with fewer packets has lower marginal returns than maintaining per-flow states for longer flows. We quantitatively demonstrate this observation using the P2P application fingerprinting task. In particular, we consider five cases: incorporating flow-level features only for flows with more than 8 (16, 1024, 2048) packets (*i.e.,* cutoff 8, cutoff 16, etc.), and considering flow-level features for all flows. We plot the overall packet classification accuracy across all flows in Figure 10. Clearly, the overall accuracies in the first three cases are very close.

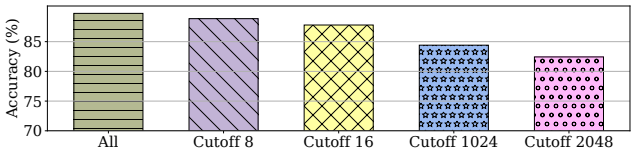


Figure 10: The packet classification accuracy *across all flows* in five cases where we use flow-level features for all flows and only flows with more than 8 (16, 1024, 2048) packets.

Given this observation, NetBeacon proposes to maintain per-flow states only for long flows. This greatly reduces the number of flows that are competing for the limited stateful storage. In reality, it is difficult to know the flow length in advance. Thus, NetBeacon introduces a long-short flow binary classification model using only per-packet features to decide whether a packet belongs to a long flow or not. Specifically, we divide the long and short flows according to the 80th percentile of the flow length in the training set. The first N packets in the long (short) flows are taken as long (short) flow samples. Then per-packet features are used for training and classification. The long-short flow binary classification model is task-specific. For instance, $N = 7$ and the accuracy is 82% in the P2P application fingerprinting task.

Handling Storage Index Collisions. When storage indices do collide, NetBeacon allows the new flow to use the occupied register if the inference class of the existing flow is determined or the flow has been finished (*i.e.,* its last packet arrival time exceeds a predefined timeout). Otherwise, NetBeacon falls back to use per-packet stateless features for the new flow.

A flow's inference class/result is determined if it has passed the last inference point defined in the multi-phase sequential model (§ 4.2) or the model is confident enough about the flow's inference result at an intermediate inference point. If so, NetBeacon saves the inference result of the flow in a *flow class table*, keyed by the flow's 5-tuple to match subsequent packets from this flow. Meanwhile, it updates the flow's storage to mark that its inference result is determined, indicating the occupied register is ready to be overwritten by future flows upon storage index collision. New entries are dynamically inserted into the flow class table by the control plane after receiving messages from the data plane indicating that a flow's class is determined. Meanwhile, some entries from the flow class table need to be regularly removed (based on either the FIFO or LRU principle) to prevent table overflow.

5.3 Integrated Data Plane Processing Logic

We now present the integrated data plane processing logic. When a packet $\mathcal{P}$ arrives, its flow hash H_1 is computed by hashing over the packet's 5-tuple (line 1 in Alg. 1). Then the last *IndexSize* bits of the hash value are used as the storage

Algorithm 1 Data Plane Processing Logic

```
1:  H_1 ← Hash(P.5t)                                  ▷ Derive flow hash from packet 5-tuple
2:  S_id ← H_1[IndexSize : 0]                         ▷ Derive storage index
3:  if not P.IsResubmit then
4:      if P.5t hit T_Class then                      ▷ T_Class table stores flow class
5:          C ← ReadT_Class(P.5t)                     ▷ Derive packet class
6:      else
7:          ID_stored ← ReadReg_ID(S_id)
8:          C_stored ← ReadReg_Class(S_id)
9:          if P.ID = ID_stored then                  ▷ packet belongs to the flow
10:             UpdateAllRegs(S_id)                   ▷ Update all registers
11:             if P is an inference point then
12:                 F_f ← CalF()                      ▷ Calculate flow-level features
13:                 C, P ← InferClass(F_p, F_f)       ▷ P is probability
14:                 if P ≥ dt then                    ▷ dt is determination threshold
15:                     UpdateReg_sure(S_id, True)
16:                     RecordT_Class(P.5t, C)
17:                 Resubmit(C)                       ▷ Update class by resubmit
18:             else
19:                 C ← C_stored
20:         else                                      ▷ packet from a new flow
21:             C ← InferClass(F_p)                   ▷ F_p is per-packet features
22:             S ← InferFlowSize(F_p)                ▷ S is flow size
23:             if S > Lth then                       ▷ Lth is the long flow threshold
24:                 sure ← ReadReg_sure(S_id)
25:                 if timeout or sure then           ▷ Reusable
26:                     InitAllRegs(S_id)             ▷ Initialize all registers
27:                     Resubmit(C)                   ▷ Initialize class by resubmit
28: else
29:     UpdateReg_Class(S_id, C)
30:     UpdateReg_ID(S_id, P.ID)
31: User-defined processing according to the packet class C
```

index S_{id} (line 2 in Alg. 1). If $\mathcal{P}$ is a normal packet (*i.e.,* not a resubmit packet, described below), it is matched by the flow class table T_{Class} (line 4 in Alg. 1). If matched, the packet's inference result/class is assigned directly as the matched class.

Otherwise, NetBeacon checks whether a per-flow state storage has been allocated for packet $\mathcal{P}$ in the stateful storage. Towards this end, it retrieves the stored true flow ID (*e.g.,* 5-tuple) using S_{id} and compares it with $\mathcal{P}$'s flow ID. If they are equal, NetBeacon identifies a new packet for the flow stored at S_{id}, and then updates the flow's state accordingly. Meanwhile, if $\mathcal{P}$ happens to be an inference point, NetBeacon calculates the flow-level features F_f, based on which NetBeacon performs model inference, together with the per-packet F_p extracted from $\mathcal{P}$. If the classification probability is greater than a predefined threshold dt, the flow's class is determined. Afterwards, the data plane first updates the flow's storage Reg_{sure} to indicate that its class is determined, and then signals the control plane to insert the flow into the flow class table. If $\mathcal{P}$ is not an inference point, it uses the stored inference result (*i.e.,* the result obtained at the most recent inference point) as its own classification result.

On the contrary, if $\mathcal{P}$ belongs to a flow without existing storage, NetBeacon obtain its classification result using only per-packet features of $\mathcal{P}$. If $\mathcal{P}$ is classified as a long-flow packet, NetBeacon checks whether the storage on S_{id} is empty or is ready to be overwritten. If so, NetBeacon initiates the per-flow state for $\mathcal{P}$'s flow using the storage indexed by S_{id}. Therefore, a storage register is lazily released when the stored flow is class-determined (or timeout), and meanwhile a new flow is hashed to the register.

Throughout the traffic analysis process, NetBeacon uses the resubmitted packets to (i) update the new inference result for an existing flow or (ii) initiate storage for a new flow. Since the purpose of Resubmit is to trigger the modification of the registers in previous stages (not modifying the packet itself), we could trigger modifications by mirroring the packet to the loopback port, instead of Resubmit or Recirculation, so that the original packet is not delayed. Further, only the inference-point-packets that trigger inference result updates are mirrored, counting for a very small fraction of packets, as measured in § 7.4.

5.4 Control Plane Logic

In NetBeacon, the control plane is responsible for (i) installing the feature tables and model table on the data plane at the very beginning, (ii) updating the *flow class table* upon receiving requests (digest in Tofino switch) from the data plane when the flow class is determined. Note that the latency for updating the flow class table does not impact traffic analysis, because the packets that are not matched by the flow class table will instead traverse the regular model inference pipeline. Therefore, the control plane is *off the critical path* of packet classification in NetBeacon, ensuring line-speed traffic analysis.

6 Implementation

We implement a prototype[1] of NetBeacon on Barefoot Tofino 1 programmable switch, including 12 MAU stages, 120MB SRAM and 6.2MB TCAM per pipeline. The development effort on the switch includes about 1300 lines of $P4_{16}$ code for the data plane and 300 lines of Python code for the control plane. We introduce two key implementations here, *i.e.,* the computation of flow-level features and multi-phase model inference. More implementations are deferred to § C.

6.1 Computing Flow-level Features

As described in § 4.1, flow-level features are divided into aggregate features and summary features.

[1] Source code: https://github.com/IDP-code/NetBeacon

```
1  action UpdateAggF1(d){ig_md.AggF1 = ig_md.AggF1 + d;}
2  table AggF1{
3      key = {a:ternary;}
4      actions = {UpdateAggF1;}
5  }
   ……
6  ig_md.total_packets = Update_total_packets.execute(ig_md.ind); //1st stage
7  ig_md.total_size = Update_total_size.execute(ig_md.ind); //1st stage
8  ig_md.size_square_sum = Update_size_square_sum.execute(ig_md.ind); //1st stage
9  if(ig_md.total_packets == 8){
10     ig_md.size_mean = ig_md.total_size>>3; //2nd stage
11     ig_md.size_square_sum_mean = ig_md.size_square_sum>>3; //2nd stage
12     ig_md.size_mean_square = Update_size_mean_square.execute(0); //3rd stage
13     ig_md.size_var = ig_md.size_square_sum-ig_md.size_mean_square; //4th stage
14 }
```

Figure 11: P4 pseudocode for computing flow-level features.

Aggregate Features. One aggregate feature is formalized as $F = \text{aggr}(a, c, d)$. As shown in Figure 11 (line 1 to line 5), NetBeacon creates a table for each aggregate feature, where attribute a is the key, condition c is the table entry, and the feature value updates d upon matching.

Summary Features. Summary features include the maximum, minimum, variance and mean of certain attributes, *e.g.,* packet size and inter-packet delay (IPD). For maximum and minimum, NetBeacon compares the stored value in the corresponding register with the new value and then updates the register accordingly. Computing mean involves division, which is not supported on the data plane. One way to work around this limitation is to multiply the threshold by the position of the corresponding inference point, which can be done offline, after obtaining the multi-phase sequential models. An alternative way is to rely on the shift operation, which, however, requires the positions of inference points to be a power of 2.

The calculation of variance $Var(X) = E(X^2) - E(X)^2$ involves both division and square. The data plane supports an approximate square calculation which only considers the first 4 bits of the value. For instance, the square of 0b101000001 (320) and square of 0b101011111 (351) are the same. Division in variance is only solvable via shift operations. This essentially requires the positions of inference points to be a power of 2 if the variance feature is used in the learning model.

As shown in Figure 11, NetBeacon uses two registers to calculate the variance: one is *total_bytes* to record the flow size in bytes (line 7), and the other is *size_square_sum* to record the sum of the squared packet sizes (line 8). When the number of packets is a power of 2 (*e.g.,* 8 in line 9), the mean packet size is calculated from *total_bytes* first (line 10), and then *size_square_sum* is averaged (line 11). Afterwards, the square of the mean packet size is calculated (line 12). Finally, the subtraction of the two returns the variance (line 13). Due to computational dependencies, the variance calculation requires 4 stages, where *total_size* and *size_square_sum* are calculated on the first stage, *size_mean* and *size_square_sum_mean* on the second stage, *size_mean_square* on the third stage and *size_var* on the fourth stage.

6.2 Multi-phase Model Inference

Our multi-phase sequential model architecture applies different models at different inference points. Intuitively, we represent each individual model at each phase separately, with its own feature tables and model table. Alternatively, we can merge their representations. As shown in Figure 12, both the feature tables and model table has an extra key named *total_packets*, which is used to distinguish models in different phases. Considering that these models may use different features, if one feature is not used by a specific model, the range mark for this feature is set as $*$ to represent any range marks. In general, model inference takes two stages: one stage for matching feature tables in parallel and one stage for matching the (aggregate) model table.

```
1  action Mark_Feature1(mark){ig_md.feat1_range_mark = mark;}
2  action Mark_Feature2(mark){ig_md.feat2_range_mark = mark;}
3  action Set_Result(result){ig_md.result = result;}
4  table Feature1{
5      key = {ig_md.total_packets:exact;
6             ig_md.size_var:ternary;}
7      actions = {Mark_Feature1;}}
8  table Feature2{
9      key = {ig_md.total_packets:exact;
10            ig_md.size_mean:ternary;}
11     actions = {Mark_Feature2;}}
12 table Model{
13     key={ig_md.total_packets:exact;
14          ig_md.feat1_range_mark:ternary;
15          ig_md.feat2_range_mark:ternary;}
16     actions={Set_Result;}}
……
17 Feature1.apply(); Feature2.apply(); //1st stage
18 Model.apply(); //2nd stage
```

Figure 12: P4 pseudocode for multi-phase model inference.

7 Evaluation

We evaluate NetBeacon extensively to demonstrate: (i) End-to-end performance improvement: NetBeacon achieves significant improvements on traffic analysis accuracies compared with baseline methods, and our results are comparable with the ideal cases with unlimited stateful storage; (ii) Efficient model representation on the data plane: NetBeacon achieves higher classification accuracy, and meanwhile consumes fewer data plane table entries compared with the baselines; (iii) We study various moving pieces in NetBeacon.

7.1 Experiment Setup

Testbed Setup. For the end-to-end experiments, we connect the programmable switch to two Linux servers. One server replays the pcap file via tcpreplay, and the other server captures the packets received from the programmable switch.

Metric and Features. We use packet-level *macro-accuracy* (defined as the average of Recalls for different classes) as the metric. When the dataset is balanced, macro-accuracy equals to accuracy. Unless otherwise stated, we use macro-accuracy and accuracy interchangeably when reporting evaluation results. The full set of per-packet features is the same as in [51]. The full set of flow-level features is the aggregate features and summary features of IPD and packet sizes. For different tasks, we select the most important features from them. Model training utilizes Python's sklearn library.

Tasks. We evaluate NetBeacon using the following tasks.

- P2P application fingerprinting. This task classifies P2P application traffic. We use traffic from three P2P applications

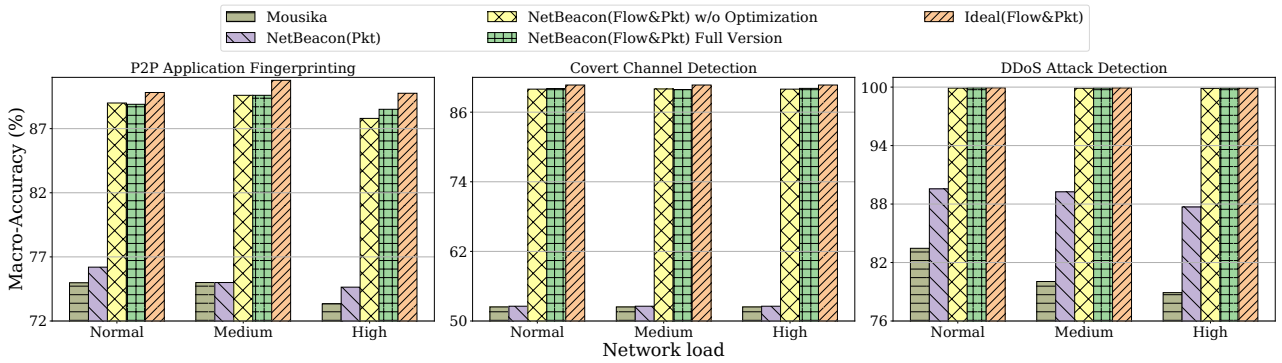


Figure 13: End-to-end performance of NetBeacon on three tasks. Compared to Mousika, NetBeacon has a significant accuracy improvement, *i.e.,* 14% in P2P application fingerprinting, 38% in covert channel detection and 20% in DDoS attack detection.

(eMule, uTorrent, and Vuze) in the PeerRush dataset [41]. Thus, it is a three-class classification task.

- Covert channel detection. The task identifies covert channel traffic encoded by censorship resistance tool Facet [28] from benign Skype traffic. We use the FacetTraffic dataset [3], and choose the 12.5% type as the covert channel traffic.
- DDoS attack detection. The task identifies DDoS traffic from benign traffic. We collect eight advanced DDoS traffic, including amplification attacks (*i.e.,* RIPv1, CLDAP, NTP, DNS, and Memcached disclosed by a real-world measurement [22]) and pulsing DDoS attacks (according to the three original settings in [33]). To avoid real-world damage, we perform the attacks in a virtual private cloud with 650 VMs. We use a MAWI dataset [49] collected in a backbone network (15th Jan. 2020) as benign background traffic.

7.2 End-to-end System Performance

In this section, we demonstrate the performance of NetBeacon under various network loads. Similar to [53], we use the media number of new flows arrived per second to represent network loads. The *IndexSize* is 16, meaning the system can simultaneously maintain up to 65536 per-flow states. We summary the main setting in the Table 4 and more details are deferred to § D. We compare NetBeacon with the following five methods. We use the same number of features in NetBeacon and these methods for fair comparisons.

- Mousika [51]: a reproduced version of Mousika. We select the model with the higher accuracy among BDT and post-distillation BDT.
- NetBeacon(Pkt). a trimmed version of NetBeacon that uses only per-packet features.
- NetBeacon(Flow&Pkt) w/o Optimization. a trimmed version of NetBeacon that uses both flow-level and per-packet features, yet without proper management of stateful storage (*i.e.,* without short/long flow differentiation and storage index collision management).
- NetBeacon(Flow&Pkt) Full Version: a full version of NetBeacon.

Tasks	P2P App Fgpt.	Covert Channel	DDoS Detection
Model	RandomForest	XGBoost	XGBoost
Training (flows)	2565270	1600	47752
Testing (flows)	246278	400	191009
Class Ratio	3:6:2	1:1	6:1(DDoS)
Network load §	1555	3144 ¶	13357

§ The media number of new flows arrived per second in the high network load case.
¶ Including Skype traffic and background traffic.

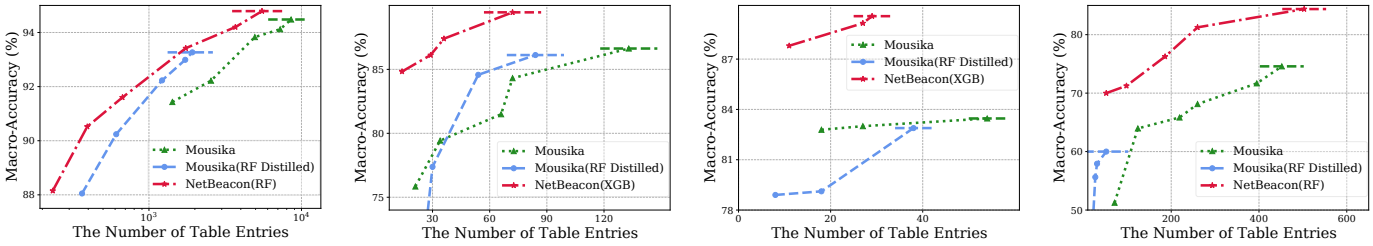
Table 4: Experimental settings.

- Ideal(Flow&Pkt): the ideal case (simulated). We use pure software code in Python to simulate the match/action based packet processing and allocate storage for each flow.

P2P Application Fingerprinting. Each pcap file in the PeerRush dataset is one hour of traffic. In order to create varying network load (*i.e.,* the media number of new flows per second), we control the number of simultaneously replayed flows. The medium (high) network load is roughly 2 (3) times higher than the normal network load. The seven most important features used are the maximum, minimum, mean, and variance of the packet size, the minimum of IPD, and the numbers of packets with packet size within $[48,64)$ and $[80,96)$. The inference points are located at the {2nd, 4th, 8th, 32th, 256th, 512th, 2048th} packet.

We plot the results in the leftmost subfigure in Figure 13. (i) Even the trimmed version of NetBeacon using only per-packet features outperforms Mousika. We provide further explanations in § 7.3. With flow-level features used, the optimized NetBeacon significantly outperforms Mousika. (ii) The full version of NetBeacon has comparable performance with the ideal solutions, demonstrating the effectiveness of stateful storage management. The minor accuracy loss is because a small fraction flows (*e.g.,* 0.85% in the high network load case) fall back to per-packet features due to hardware limitations. (iii) In this task, NetBeacon uses 12 stages, along with 17.29% of SRAM and 31.25% of TCAM. Our experiments were conducted with the 1st generation Tofino chip. The amount of stage, TCAM and SRAM resources have almost doubled in the latest Tofino chips, so the resources consumed by NetBeacon are acceptable.

Covert Channel Detection. There are only 2000 flows in the covert channel detection task, of which 1000 are benign and



(a) P2P Application Fingerprinting (b) Covert Channel Detection (c) DDoS Attack Detection (d) Traffic Type Classification

Figure 14: The comparison of model representation efficiency. On the one hand, NetBeacon achieves higher accuracies when consuming the same number of table entries; on the other hand, to represent two similarly-performing models, NetBeacon uses much fewer table entries on the data plane.

the rest 1000 are malicious. Since the dataset is Skype traffic, all flows are very long. Therefore, we add background traffic from the Univ2 dataset [6] so the entire traffic includes a more balanced distribution of short and long flows. We adjust the pcap file replay rate to control the network load. The features used in NetBeacon are the numbers of packets with packet size in in $[96,112)$, $[112,128)$, $[128,144)$, $[144,160)$, $[160,176)$ and $[176,192)$ respectively. The inference points are located at the {512th, 1024th, 1500th, 2048th, 3000th, 4096th, 6000th, 8192th, 10000th} packet.

We show the experimental results in the middle subfigure of Figure 13. (i) Because the covert channel is constructed based on flow-level information, per-packet features have very limited contributions in the covert channel detection. (ii) Both versions of NetBeacon with flow-level features achieve similar classification performance as the ideal method. We notice that storage index collisions in this task are not frequent since the number of relevant long flows is relatively small. Thus, the quantitative benefits of stateful storage management are not significant in this task. (iii) The full version of NetBeacon consumes 12 stages, along with 13.44% of SRAM and 34.03% of TCAM, for this task.

DDoS Attack Detection. We merge both benign traffic and DDoS traffic to create network traffic and control the network load by adjusting the number of simultaneously replayed flows. The features used in this task are the minimum IPD and packet sizes. The inference points are located at the {2nd, 5th, 8th, 16th} packet.

We show the experimental results in the rightmost subfigure of Figure 13. (i) The accuracies of flow-agnostic version of NetBeacon are 6% to 9% higher than those of Mousika in various network loads. (ii) After incorporating flow-level features, the detection accuracies are 20% higher and very close to the ideal method. We notice that the quantitative benefits of the stateful storage management are not significant in this task. This is because the accurate flow class can be obtained at an earlier phase before a storage collision happens. (iii) NetBeacon occupied 9 stages, along with 11.11% of SRAM and 1.85% of TCAM, for this task.

Caveats. In the Table 5, we further report the FPR/FNR in each task. In general, due to hardware constraint, the ML models designed in the area of IDP are limited in features, model sizes (*e.g.,* tree depth) and the locations of inference points. Deeper investigation of the results reveals that the flow-level macro-accuracy is fairly decent. For instance, if we consider a covert channel flow is detected when at least five of the inference points correctly identify its packets, then flow-level FPR and FNR are 5.3% and 7.3%. Interpreting flow-level macro-accuracy is particularly useful for malicious flow detection in intrusion detection/prevention systems. Nevertheless, further improving inference accuracies for traffic analysis tasks that require complex model and features is an active research direction in IDP for both the networking and vendor communities.

Tasks	P2P App Fgpt.	Covert Channel	DDoS Detection
FPR/FNR	7.6%/12%/14.7%*	8.8%/10.9%	0.7%/0.05%

* The multi-class FPRs are defined under 1 v.s. all.

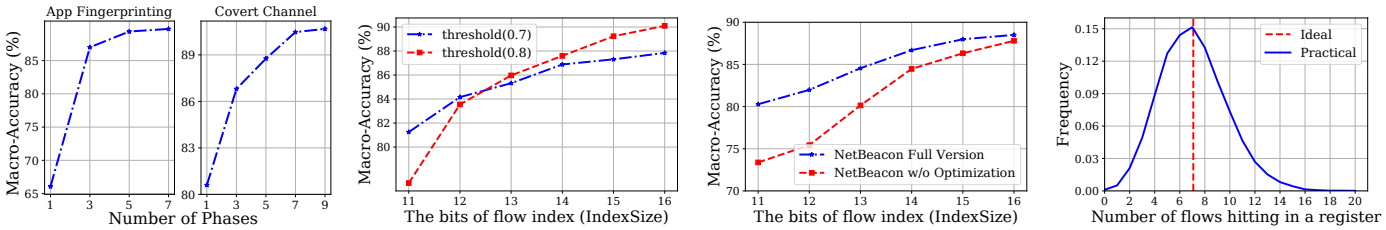
Table 5: The FPRs/FNRs in the three tasks.

7.3 Model Representation Performance

In this section, we show that the data plane model representation of NetBeacon is more efficient compared with other IDP works. We use Mousika and Mousika (RF Distilled) [51] as two baselines. Mousika represents the directly trained BDT model. Mousika(RF Distilled) represents the BDT model obtained by distillation of the trained RandomForest model.

Experiment Setup. In addition to the three tasks above, *i.e.,* P2P application fingerprinting, covert channel detection, and DDoS attack detection, we further add a traffic type classification task in this part. We use the ISCXVPN dataset [12] for this task, including six traffic types: chat, P2P, email, VoIP, streaming, and file transfer. The RandomForest is selected for the traffic type classification task. To ensure a fair comparison, NetBeacon and Mousika use the same training data. We first extract the features from the original packets as the training data for NetBeacon. Since the model in Mousika is binary decision tree (BDT), we transformed the features into binary as the training data for Mousika.

The evaluation metrics are the classification accuracy and the number of data plane table entries required to represent the model. For NetBeacon, the number of table entries is the sum



(a) The impact of number of phases. (b) The impact of determination threshold. (c) The impact of the index size. (d) Impact of the hash function in switch.

Figure 15: The impact of different factors on NetBeacon performance. The tasks in (b) and (c) are covert channel detection and P2P application fingerprinting, respectively. The experimental configuration in (d) is 58000 flows and 8192 registers.

of the number of entries in all feature tables and model table. The models in Mousika are represented using a single giant table. For NetBeacon, we adjust the number of trees and the maximum tree depth to obtain different learning accuracies and different numbers of table entries. Mousika does not have a similar mechanism to balance the accuracy and number of table entries, so we adjust the size of training data to produce different models for Mousika.

Experimental Results. We present the results in Figure 14. (i) When consuming the same number of table entries, NetBeacon achieves higher accuracies on all four tasks compared with Mousika. Two possible reasons may contribute to this performance gain. The first one is that the features used by BDT itself are binary features which may not be expressive. The second is that the design of BDT itself may be less efficient than the state-of-the-art tree-based models used by NetBeacon, *i.e.,* RF and XGB. (ii) To represent two similarly-performing models, NetBeacon uses much fewer table entries on the data plane (*e.g.,* 75% reduction in covert channel detection). This demonstrates the effectiveness of our range marking and range coding mechanisms introduced in § 5.1. In fact, we realize that the design of BDT itself tends to generate a large number of table entries; we provide a detailed discussion in § A.

7.4 NetBeacon Deep Dive

We explore the factors affecting NetBeacon performance.

Throughput. The PISA pipeline on programmable switches ensures that any compiled program will run at line-speed. The measured throughput for P2P application fingerprinting, covert channel detection, and DDoS attack detection are 99.12 Gbps, 99.13 Gbps and 99.18 Gbps, respectively, on a 100Gb forwarding port. Since one 100Gb loopback port is shared by 16 forwarding ports in a pipeline, the ratio of mirrored packets in each forwarding port should be, on average, smaller than 1/16 to ensure that the loopback port will not become the bottleneck. The packet-mirroring ratios are 3.8%, 0.09% and 1.28%, respectively, in our tasks.

The Number of Inference Points. The number of inference points/phases employed in the multi-phase sequential model is an important factor affecting performance. As presented in § 4.2, the flow-level information could be premature at earlier phases, which may affect classification accuracy. As we can see from Figure 15a, the accuracy increases as we append additional inference points.

Determination Thresholds. As presented in § 4.2, we set a determination threshold for each inference point. In Figure 15b, we explore two determination thresholds. A lower determination threshold allows earlier classification and release of storage registers, but may lead to misclassifications. As we can see, when the number of available registers is small (*i.e.,* small *IndexSize*), the smaller lower determination threshold allows the registers to be released earlier, which compensates for the accuracy losses of making inference decisions at earlier phases. As a result, the lower determination threshold achieves better accuracy results than the higher determination threshold. On the contrary, when the number of storage registers is larger, using a higher determination threshold is more beneficial. Therefore, the determination thresholds should be decided considering the available hardware resources.

The Size of Stateful Storage (*i.e., IndexSize*). Stateful storage is necessary to maintain per-flow states and compute flow-level features. Thus, it is critical to mitigate the problem of storage index collisions caused by different flows. In Figure 15c, we show that the size of stateful storage and classification accuracy are positively correlated. Meanwhile, our stateful storage management improves packet classification accuracy by non-trivial margins upon the limited stateful storage.

Imperfection of Hardware Hashing. We obtain the number flows hashed to each storage register, and plot the distribution in Figure 15d. Ideally, all registers should have the same number of hits, *i.e.,* the red dashed line in the Figure 15d. The actual distribution does deviate from the ideal distribution, which could introduce extra storage index collisions.

8 Discussion

Addressable Market Analysis. Prior art on learning-based traffic analysis [4, 14, 37] places feature engineering and/or ML model inference on the control plane (*e.g.,* an auxiliary server). Thus, they can use rather sophisticated features and models, yet yielding much smaller analysis throughput than the line-speed. Another category of art [2, 32, 61] primarily considers DDoS mitigation by encoding defense rules

on programmable switches, achieving line-speed traffic filtering. Yet their approaches are not generalizable to realize other learning-based traffic analysis. In contrast, NetBeacon advocates IDP that directly embeds general-purpose learning models into the network data plane to empower intelligent traffic analysis at line-speed. Compared with existing IDP-related designs [10, 26, 51, 55, 63], NetBeacon surpasses them in both analysis accuracy and model representation efficiency. However, due to the constraints of the PISA pipeline, both features and model architectures are limited in NetBeacon, which might affect the analysis accuracies in case of complex tasks, as quantified in Table 5. Besides programmable switches, prior art also consider SmartNICs [45, 50] and DPDK-driven host networking [35] as the intelligent data plane to enable more flexible computation and storage, at the expense of much lower throughput (for instance, NetFPGA-PLUS has 200 Gbps while the Tofino 1 programmable switch in aggregate has 6.4 Tbps).

Scalability Analysis. NetBeacon has the following traffic analysis hierarchy: the flow class table stores flows whose classification classes have been determined (line 4-5 in Alg. 1). When matched by the flow class table, a packet will not be re-analyzed by the ML model deployed after the flow class table. Otherwise, the packet enters the typical ML inference. If NetBeacon maintains the per-flow state in registers (SRAM) for the packet (line 11-19 in Alg. 1), the ML inference will use flow-level features; otherwise, the inference is purely based on per-packet features (line21 in Alg. 1). Thus, there is *no hard-limit on the number of concurrent flows that can be processed by NetBeacon*, since it can fall back to use per-packet features whenever necessary.

The theoretically maximum number of per-flow states that NetBeacon can simultaneously maintain is determined by the size of registers (SRAM) and the flow-level features being used. In the DDoS task where scalability matters the most, one pipeline on Tofino 1 switch (with 4 pipelines in total) can support up to 140,000 flow indices in registers (SRAM). The SRAM size in Tofino 3 increases by 80% and the total number of pipelines doubles. Architecturally, we can deploy multiple NetBeacon instances in parallel and load balance flows across these instances. Each NetBeacon instance independently processes traffic at line-speed.

Security Analysis. We discuss several adaptive (or white-box) attacks against NetBeacon. The first type is resource exhausting attack where the adversary generates numerous concurrent bogus flows to overflow the available SRAM on a switch. This is essentially a form of Denial-of-Capability (DoC) attack [38]. Fortunately, DoC attacks will not fully break down NetBeacon since it can fall back to use per-packet features. The probably-secure defense against DoC attacks is through dedicated set-up protocols (*e.g.,* Portcullis [40]). Practically, DoC attacks are mitigated by either aggregating flows or deploying multiple NetBeacon instances in parallel.

The second category of attacks aims at exploiting the detailed system parameters in NetBeacon. Two representative examples are: (i) the stealth attack where the attacker sends benign packets first, and then transmits malicious packets after NetBeacon determines the flow class; (ii) the low-rate-long-flow attack where the attacker forces the flow to timeout. In general, these types of attacks have entry barriers since the adversary needs to probe the parameter setting. To mitigate the stealth attack, NetBeacon can enforce a maximum flow length limit on the entries in the flow class table. To mitigate the low-rate-long-flow-attack, NetBeacon can determine a flow-specific timeout (*e.g.,* twice the maximum observed IPD) for each flow.

9 Related Work

ML-powered Traffic Analysis. Learning-based traffic has been studied for a long time [7, 38]. Recently, the development of AI and the enrichment of network scenarios encourage studies in this area. Some works focus on feature design. [5, 14, 15, 42] perform matrix transformation, compression and frequency transformation respectively for feature. Some works focus on classification under encrypted traffic. [16, 27, 47, 59] build different models according to the scenario for encrypted traffic. Some works address the common problem of traffic classification by employing the latest techniques in AI, such as deep learning [42], autoencoder [37], data augmentation [21], automated machine learning [17], etc. These art, however, does not primarily focus on IDP.

Applications for programmable switches. Programmable switches are applied for a wide range of fields due to its flexibility, including switch function optimization [18, 39, 60], network measurement and monitoring [48, 58, 62], network security [23, 52, 54, 64], distributed systems [25, 44], etc.

10 Conclusion

Intelligent Data Plane (IDP) may potentially enable a new paradigm of networking designs, by achieving intelligent traffic analysis at line-speed using data-driven models rather than predefined protocols. In this paper, we present NetBeacon, the most advanced IDP design that outperforms prior art in both learning model accuracy and model representation efficiency. At its core, NetBeacon is empowered by (i) a data plane aware model design centering around a multi-phase sequential model architecture, (ii) efficient model representation mechanisms, and (iii) stateful storage management designs. We implement a prototype of NetBeacon and extensively evaluate its performance over multiple traffic analysis tasks.

11 Acknowledgments

We thank our shepherd and anonymous reviewers for their thoughtful comments. This work was supported in part by the

China National Funds for Distinguished Young Scientists under Grant 61825204, in part by NSFC under Grant 61932016 and 62132011, and in part by the Beijing Outstanding Young Scientist Program under Grant BJJWZYJH01201910003011. Ke Xu is the corresponding author.

References

[1] S. Abbasloo, C. Y. Yen, and H. J. Chao. Classic Meets Modern: A Pragmatic Learning-Based Congestion Control for the Internet. In *Proc. SIGCOMM*, 2020.

[2] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever. Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense. In *Proc. SIGCOMM*, 2022.

[3] D. Barradas, N. Santos, and L. Rodrigues. Effective Detection of Multimedia Protocol Tunneling using Machine Learning. In *Proc. USENIX Security*, 2018.

[4] D. Barradas, N. Santos, L. Rodrigues, et al. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *Proc. NDSS*, 2023.

[5] K. Bartos, M. Sofka, and V. Franc. Optimized Invariant Representation of Network Traffic for Detecting Unseen Malware Variants. In *Proc. USENIX Security*, 2016.

[6] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. IMC*, 2010.

[7] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic Classification on the Fly. *SIGCOMM-CCR*, 2006.

[8] P. Bosshart, D. Daly, G. Gibb, et al. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM-CCR*, 2014.

[9] A. Bremler-Barr and D. Hendler. Space-Efficient TCAM-Based Classification using Gray Coding. *IEEE Transactions on Computers*, 2010.

[10] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever. pForest: In-Network Inference with Random Forests. *arXiv preprint arXiv:1909.05680*, 2019.

[11] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization. In *Proc. SIGCOMM*, 2018.

[12] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani. Characterization of Encrypted and VPN Traffic using Time-Related Features. In *Proc. ICISSP*, 2016.

[13] A. D Ferguson, S. Gribble, C. Y. Hong, et al. Orion: Google's Software-Defined Networking Control Plane. In *Proc. NSDI*, 2021.

[14] C. Fu, Q. Li, M. Shen, and K. Xu. Realtime Robust Malicious Traffic Detection via Frequency Domain Analysis. In *Proc. CCS*, 2021.

[15] C. Fu, Q. Li, M. Shen, and K. Xu. Frequency Domain Feature Based Robust Malicious Traffic Detection. *ToN*, to appear.

[16] C. Fu, Q. Li, and K. Xu. Detecting Unknown Encrypted Malicious Traffic in Real Time via Flow Interaction Graph Analysis. In *Proc. NDSS*, 2023.

[17] J. Holland, P. Schmitt, N. Feamster, and P. Mittal. New Directions in Automated Traffic Analysis. In *Proc. CCS*, 2021.

[18] K. F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker. Contra: A Programmable System for Performance-Aware Routing. In *Proc. NSDI*, 2020.

[19] S. Huang, M. Wang, and Y. Cui. Traffic-Aware Buffer Management in Shared Memory Switches. In *Proc. INFOCOM*, 2021.

[20] S. Jain, A. Kumar, S. Mandal, et al. B4: Experience with a Globally-Deployed Software Defined WAN. *SIGCOMM-CCR*, 2013.

[21] S. T. Jan, Q. Hao, T. Hu, et al. Throwing Darts in the Dark? Detecting Bots with Limited Data using Neural Data Augmentation. In *Proc. SP*, 2020.

[22] M. Jonker, A. King, J. Krupp, et al. Millions of Targets Under Attack: a Macroscopic Characterization of the DoS Ecosystem. In *Proc. IMC*, 2017.

[23] Q. Kang, L. Xue, A. Morrison, et al. Programmable In-Network Security for Context-aware BYOD Policies. In *Proc. USENIX Security*, 2020.

[24] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. *SIGCOMM-CCR*, 2005.

[25] C. Lao, Y. Le, K. Mahajan, et al. ATP: In-network Aggregation for Multi-tenant Learning. In *Proc. NSDI*, 2021.

[26] J. Lee and K. Singh. SwitchTree: In-Network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications*, 2020.

[27] J. Li, S. Wu, H. Zhou, et al. Packet-Level Open-World App Fingerprinting on Wireless Traffic. In *Proc. NDSS*, 2022.

[28] S. Li, M. Schliep, and N. Hopper. Facet: Streaming over Videoconferencing for Censorship Circumvention. In *Proc. WPES*, 2014.

[29] C. Liu, M. Xu, Y. Yang, and N. Geng. DRL-OR: Deep Reinforcement Learning-based Online Routing for Multi-type Service Requirements. In *Proc. INFOCOM*, 2021.

[30] H. Liu. Efficient Mapping of Range Classifier into Ternary-CAM. In *Proceedings 10th Symposium on High Performance Interconnects*, 2002.

[31] Z. Liu, H. Jin, Y. C. Hu, and M. Bailey. Practical Proactive DDoS-Attack Mitigation via Endpoint-Driven In-

Network Traffic Control. *TON*, 2018.

[32] Z. Liu, H. Namkung, G. Nikolaidis, et al. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *Proc. USENIX Security*, 2021.

[33] X. Luo and R. K. C. Chang. On a New Class of Pulsing Denial-of-Service Attacks and the Defense. In *Proc. NDSS*, 2005.

[34] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proc. SIGCOMM*, 2017.

[35] M. Marty, M. de Kruijf, J. Adriaens, et al. Snap: A Microkernel Approach to Host Networking. In *Proc. SOSP*, 2019.

[36] N. McKeown, T. Anderson, H. Balakrishnan, et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM-CCR*, 2008.

[37] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection. In *Proc. NDSS*, 2018.

[38] A. W. Moore and D. Zuev. Internet Traffic Classification using Bayesian Analysis Techniques. In *Proc. SIGMETRICS*, 2005.

[39] T. Pan, N. Yu, C. Jia, et al. Sailfish: Accelerating Cloud-Scale Multi-Tenant Multi-Service Gateways with Programmable Switches. In *Proc. SIGCOMM*, 2021.

[40] B. Parno, D. Wendlandt, E. Shi, et al. Portcullis: Protecting Connection Setup from Denial-Of-Capability Attacks. *SIGCOMM-CCR*, 2007.

[41] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li. Peerrush: Mining for Unwanted P2P Traffic. In *Proc. DIMVA*, 2013.

[42] V. Rimmer, D. Preuveneers, M. Juárez, T. Van Goethem, and W. Joosen. Automated Website Fingerprinting through Deep Learning. In *Proc. NDSS*, 2018.

[43] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. Optimal in/out TCAM Encodings of Ranges. *TON*, 2015.

[44] A. Sapio, M. Canini, C. Y. Ho, et al. Scaling Distributed Machine Learning with In-Network Aggregation. In *Proc. NSDI*, 2021.

[45] G. Siracusano, S. Galea, D. Sanvito, et al. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *Proc. NSDI*, 2022.

[46] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *Proc. SIGCOMM*, 1998.

[47] T. van Ede, R. Bortolameotti, A. Continella, et al. Flowprint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In *Proc. NDSS*, 2020.

[48] W. Wang, X. C. Wu, P. Tammana, A. Chen, and T. E. Ng. Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon. In *Proc. NSDI*, 2022.

[49] WIDE. MAWI working group traffic archive. http://mawi.wide.ad.jp/mawi/, Accessed Nov. 2021.

[50] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello. Programmable Switches for In-networking classification. In *Proc. INFOCOM*, 2021.

[51] G. Xie, Q. Li, Y. Dong, et al. Mousika: Enable General In-Network Intelligence in Programmable Switches by Knowledge Distillation. In *Proc. INFOCOM*, 2022.

[52] J. Xing, K. F. Hsu, Y. Qiu, et al. Bedrock: Programmable Network Support for Secure RDMA Systems. In *Proc. USENIX Security*, 2021.

[53] J. Xing, Q. Kang, and A. Chen. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *Proc. USENIX Security*, 2020.

[54] J. Xing, W. Wu, and A. Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *Proc. USENIX Security*, 2021.

[55] Z. Xiong and N. Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proc. HotNets*, 2019.

[56] F. Y. Yan, H. Ayers, C. Zhu, et al. Learning in situ: A Randomized Experiment in Video Streaming. In *Proc. NSDI*, 2020.

[57] S. Yan, X. Wang, X. Zheng, et al. ACC: Automatic ECN Tuning for High-Speed Datacenter Networks. In *Proc. SIGCOMM*, 2021.

[58] T. Yang, J. Jiang, P. Liu, et al. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proc. SIGCOMM*, 2018.

[59] Q. Yin, Z. Liu, Q. Li, et al. Automated Multi-Tab Website Fingerprinting Attack. *TDSC*, to appear.

[60] Z. Yu, C. Hu, J. Wu, et al. Programmable Packet Scheduling with a Single Queue. In *Proc. SIGCOMM*, 2021.

[61] M. Zhang, G. Li, S. Wang, et al. Poseidon: Mitigating volumetric DDoS attacks with programmable switches. In *Proc. NDSS*, 2020.

[62] Y. Zhang, Z. Liu, R. Wang, et al. CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query. In *Proc. SIGCOMM*, 2021.

[63] C. Zheng and N. Zilberman. Planter: Seeding Trees within Switches. In *Proc. SIGCOMM Poster*. 2021.

[64] J. Zheng, Q. Li, G. Guo, et al. Realtime DDoS Defense Using COTS SDN Switches via Adaptive Correlation Analysis. *TIFS*, 2018.

[65] G. Zhou, G. Chen, F. Lin, et al. Primus: Fast and Robust Centralized Routing for Large-scale Data Center Networks. In *Proc. INFOCOM*, 2021.

[66] H. Zhu, V. Gupta, S. S. Ahuja, et al. Network Planning with Deep Reinforcement Learning. In *Proc. SIGCOMM*, 2021.

```
action CheckFeature1(next_node, threshold){
        ig_md.prev_node = next_node;
        ig_md.is_True = 0;
        if (hdr.ipv4.total_len < threshold){
            ig_md.is_True = 1;}
    }
action CheckFeature2(next_node, threshold){
        ig_md.prev_node = next_node;
        ig_md.is_True = 0;
        if (hdr.ipv4.ttl < threshold){
            ig_md.is_True = 1;}
    }
table tree1_level_i{
        key={ig_md.prev_node: exact;
            ig_md.is_True: exact;}
        actions={
            CheckFeature1;
            CheckFeature2;
            SetResult1;
        }
    }
Tree1_level_1.apply();
      ……
Tree1_level_n.apply();
```

Conditions in an action must be simple comparisons of an action data parameter.

SwitchTree

```
action Feature1_Code(code1,code2){
      ig_md.tree1_feat1_code = code1;
      ig_md.tree2_feat1_code = code2; }
action Feature2_Code(code1,code2){
      ig_md.tree1_feat2_code = code1;
      ig_md.tree2_feat2_code = code2; }
action Set_Result1(result){ig_md.result1 = result;}
action Set_Result2(result){ig_md.result2 = result;}
table Feature1{
    key = {hdr.ipv4.total_len:exact;}
    actions = {Feature1_Code;}}
table Feature2{
    key = {hdr.ipv4.ttl:exact;}
    actions = {Feature2_Code;}}
table tree_1{
    key={ig_md.tree1_feat1_code:exact;
         ig_md.tree1_feat2_code:exact;}
    actions={Set_Result1;}}
table tree_2{
    key={ig_md.tree2_feat1_code:exact;
         ig_md.tree2_feat2_code:exact;}
    actions={Set_Result2;}}
Feature1.apply(); Feature2.apply();
Tree1.apply(); Tree2.apply();
```

Planter

Figure 16: The main P4 pseudocodes of SwitchTree and Planter.

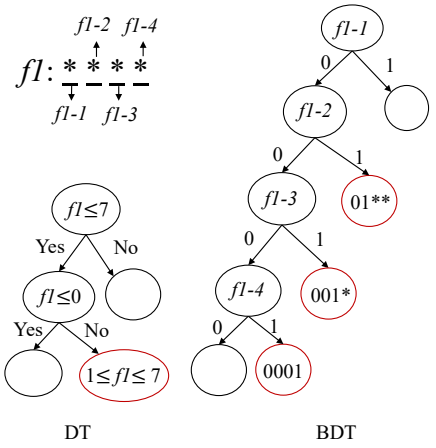


Figure 17: An example of BDT structure.

The appendix is divided into four parts. Appendix A is an additional description of state-of-the-art IDP works. Appendix B gives a faster solver of CRC algorithm. Appendix C gives more detail of NetBeacon's implementation. Appendix D provides a more detailed description of the experiments.

A Supplements for IDP Works

pForest and SwitchTree. In pForest [10] and SwitchTree [26], the stages from left to right correspond to each layer of the decision tree from top to bottom. The table key in each stage is the node of the corresponding layer of the decision tree and the next branch. The parameter of action is the information of the node on the branch, including feature and threshold. After matching, the comparison between the current feature value and the feature threshold is executed in action. The number of switch stages limits the depth of the decision tree model in this way. The main P4 pseudocode of SwitchTree is shown in Figure 16. And table $tree1\_level\_i$ represents the nodes at level i of the decision tree 1. We create a check action for each feature, eliminating the need to determine which feature is currently to be compared as the open source code does. Unfortunately, complex conditions are not allowed in the action, and both sides of the comparison operation cannot be variables in production-grade switches. We show the compilation error (compiler version: 9.1.0) in red in Figure 16. These factors result in the above two works being unfeasible in production grade switches.

IIsy and Planter. In IIsy [55] and Planter [63], the features are first encoded and then the decision tree is encoded. Specifically, feature encoding maps feature values to decision tree branches, and decision tree encoding maps branches to leaf nodes. As shown in Figure 16, $ig\_md.tree1\_feat1\_code$ represents the branch of feature 1 in tree 1. Planter only gives a simple example, and is unclear how to support complex decision tree models, *e.g.,* how Planter encodes branches when using one feature multiple times in a tree.

Mousika. Mousika [51] proposed a data plane friendly decision tree model, binary decision tree (BDT), that uses the binary bits of features as tree nodes. As shown in Figure 17, feature $f1$ is converted into 4 binary features in the BDT, denoted $f1$-n, corresponding to the four bits of $f1$. The rule $1 \leq f1 \leq 7$ in traditional DT needs to be covered by three leaf nodes together in the BDT, where a leaf node corresponds to a table entry in the data plane. The three nodes are exactly the three prefixes of the range [1,7]. Further, if both $1 \leq f1 \leq 7$ and $1 \leq f2 \leq 7$ need to be satisfied, $3*3$ leaf nodes will be needed, which means 9 table entries on the data plane. In brief, the number of leaves of BDT and table entries on the data plane will face a combination explosion issue as the number of features to be combined increases. Instead of transforming directly from DT, Mousika obtains BDT by training binary features, which somewhat alleviates the combination explosion issue of BDT.

B CRC Algorithm Optimization

As described in §5.1, the CRC algorithm needs to solve the optimization problem in Equation (1). We introduce a solver here. The intuition of our solver is gradually merging the first two prefixes in the prefix list of range into one prefix. For instance, the first two prefixes of [68,103) are 010001** and 01001*** as shown in Figure 6. To merge the two prefixes, the P_{start} needs to be moved to the left by 4. Because the merged prefix needs to have more covers than the existing prefixes, *i.e.,* twice the maximum of the existing prefixes' covers. In the first two prefixes of [68,103), 01001*** has the maximum covers, *i.e.,* 8, so the merged prefix needs to have 16 covers. The moved steps is 4, *i.e.,* 16 (covers of merged prefix) - 8 (covers of 01001***) - 4 (covers of 010001**) = 8 - 4. We show the operation in line 14 of Alg. 2. When only one prefix in the prefix list of range (line 12) or the $diff$ is negative (line 15), it means the P_{start} cannot be moved to merge prefixes. In this way, the number of moves of P_{start} does not exceed

Algorithm 2 Consecutive Ranges Coding (CRC)

Input: $r_1, r_2, ..., r_n$ are the n thresholds of a feature and the feature occupy w bits. $\mathsf{Prefix}([r_i, r_{i+1}))$ returns the list of prefixes obtained by the Prefix method [46] for range $[r_i, r_{i+1})$. $\mathsf{Num}(plist)$ returns the size of list $plist$. $\mathsf{Cov}(p)$ returns the number of values covered by prefix p, *i.e.,* covers of p.

Output: $Tern$ is the list of prefixes of the n ranges.

$r_0 \leftarrow 0$, $r_{n+1} \leftarrow 2^w$, $Tern \leftarrow []$
for each thresholds $i = 1, ..., n$ **do**
  $Minm \leftarrow \infty$ ▷ $Minm$ is the minimum number of prefixes
  $start \leftarrow r_{i-1}$, $end \leftarrow r_i$
  while $end < r_{i+1}$ **do**
    $plist1 \leftarrow \mathsf{Prefix}([r_i, end))$
    while $start \geq 0$ **do**
      $plist2 \leftarrow \mathsf{Prefix}([start, end))$
      if $\mathsf{Num}(plist1) + \mathsf{Num}(plist2) < Minm$ **then**
        $Minm \leftarrow \mathsf{Num}(plist1) + \mathsf{Num}(plist2)$
        $Best_s \leftarrow start$, $Best_e \leftarrow end$
      if $\mathsf{Num}(plist2) == 1$ **then**
        Break
      $diff \leftarrow \mathsf{Cov}(plist2[1]) - \mathsf{Cov}(plist2[0])$
      if $diff < 0$ **then**
        Break
      $start \leftarrow start - diff$
    if $\mathsf{Num}(plist2) == 1$ **then**
      Break
    $diff \leftarrow \mathsf{Cov}(plist2[-2]) - \mathsf{Cov}(plist2[-1])$
    if $diff < 0$ **then**
      Break
    $end \leftarrow end + diff$
  $Tern$ extend $\mathsf{Prefix}([r_i, best_e))$ with higher priority $2i$
  $Tern$ extend $\mathsf{Prefix}([best_s, best_e))$ with priority $2i+1$

the number of prefixes in the prefix list of range $[r_{i-1}, r_i)$ and thus the complexity is $O(\lg \mathcal{R})$. The search idea of P_{end} is the same as P_{start}. Therefore, the total complexity is $O((\lg \mathcal{R})^2)$.

C Supplements for Implementation

Determined Threshold. As shown in Alg. 1 line 14, the data plane needs to judge whether the probability exceeds the determination threshold. NetBeacon puts the classification probabilities and threshold comparisons on the control plane so that the determination threshold could be updated without downtime. Specifically, NetBeacon takes $n+x$ (n) as the action parameter of the model table for class n if the classification probability does (not) reach the threshold, where x is an arbitrary number greater than the total number of classes. In this way, the data plane only needs to judge whether the action parameter is greater than x to know whether the probability is greater than the determination threshold.

Flow Class Table Entry Replacement. When the flow class table is full, NetBeacon removes table entries to allow for the insertion of new entries. In analogy to the OS page replacement algorithm, there are two table entry replacement algorithms.

FIFO (First In, First Out): The control plane maintains a list of flows in order of insertion. When entry replacement is required, FIFO selects the earliest inserted table entry as the replacement entry. FIFO is suitable for scenarios with little difference in flow duration. **LRU (Least Recently Used)**: A counter can be set to record the number of packets matched to a table entry in the flow class table. The time that the counter value has not changed is considered the time that the table entry has not been used. When entry replacement is required, the table entry that has not been used for the longest time is selected. LRU is suitable for scenarios with a large difference in flow duration.

D Experiment Setting

End-to-end Experiment Details. We provide a more detailed description of the experimental setting. We show the flow duration distributions in Figure 18.

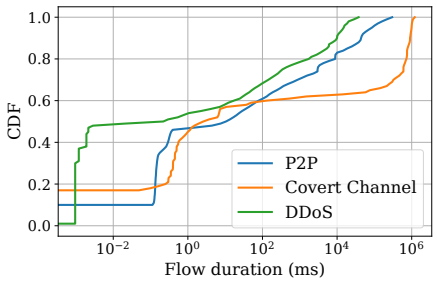


Figure 18: The flow duration distributions in three tasks.

P2P Application Fingerprinting. The model size in NetBeacon(Pkt) is 2*9 (two trees and max depth is 9). The model sizes in seven phases of NetBeacon(Pkt&Flow) are {1*9, 1*9, 1*9, 1*9, 1*9, 1*9 and 1*9}, respectively. The model size of flow size prediction is 1*8. We choose 10% flows of pcaps from 72 hours as the test set, and the training set is the rest of the data from the 72 hours.

Covert Channel Detection. The model size in NetBeacon(Pkt) is 1*10 (1 tree and max depth is 10). The model sizes in nine phases of NetBeacon(Pkt&Flow) are {2*5, 3*6, 3*6, 3*6, 3*7, 2*7, 3*6, 3*5 and 3*4}, respectively. We merge nine pcaps in the Univ2 dataset and replace the long flows (>4000 pkts) with 20% Skype flows as the test set. The rest Skype traffic is the training set.

DDoS Attack Detection. The model size in NetBeacon(Pkt) is 1*4 (1 tree and max depth is 4). The model sizes in four phases of NetBeacon(Pkt&Flow) are {1*1, 1*4, 1*1 and 1*1}, respectively. We merge three 40s of the MAWI dataset and malicious dataset as the whole set, of which 80% is the test set and the rest is the training set.