SPLIDT: Partitioned Decision Trees for Scalable Stateful Inference at Line Rate

Murayyiam Parvez[*], *Annus Zulfiqar*[◇*], *Roman Beltiukov*[○], *Shir Landau Feibish*[‡], *Walter Willinger*[⋆†], *Arpit Gupta*[○], *Muhammad Shahbaz*[◇]

Purdue University [◇]*University of Michigan* [○]*UCSB* [‡]*University of Haifa* [⋆]*Northwestern University*

Abstract

Machine learning (ML) is increasingly being deployed in programmable data planes (switches and SmartNICs) to enable real-time traffic analysis, security monitoring, and in-network decision-making. Decision trees (DTs) are particularly well-suited for these tasks due to their interpretability and compatibility with data-plane architectures, i.e., match-action tables (MATs). However, existing in-network DT implementations are constrained by the need to compute all input features upfront, forcing models to rely on a small, fixed set of features per flow. This significantly limits model accuracy and scalability under stringent hardware resource constraints.

We present SPLIDT, a system that rethinks DT deployment in the data plane by enabling partitioned inference over sliding windows of packets. SPLIDT introduces two key innovations: (1) it groups individual subtrees of a DT into partitions and allows each subtree to have its own feature set, and (2) it leverages an in-band control channel (via recirculation) to reuse data-plane resources (both stateful registers and match keys) across partitions at line rate. These insights allow SPLIDT to scale the number of stateful features a model can use without exceeding hardware limits. To support this architecture, SPLIDT incorporates a custom training and design-space exploration (DSE) framework that jointly optimizes feature allocation, tree partitioning, and DT model depth. Evaluation across multiple real-world datasets shows that SPLIDT achieves higher accuracy while supporting up to 5× more stateful features than prior approaches (e.g., NetBeacon and Leo). It maintains the same low time-to-detection (TTD) as these systems, while scaling to millions of flows with minimal recirculation overhead (≤0.05%).

1 Introduction

Machine Learning (ML) is rapidly becoming a cornerstone of modern networking, driving increasingly sophisticated applications such as DDoS detection (LUCID [25], Flowlens [5]), intrusion detection [14,15,75], encrypted traffic analysis [4,74,80], malware classification [2,29], IoT botnet detection [24] as well as congestion control [23,39,52,77,86], and variable bitrate (VBR) video streaming [55,85]. These use cases demand real-time, high-throughput inference [71] to keep up with the ever-growing scale and complexity of network traffic [6,14–18,65].

[*]Both authors contributed equally to this work.

[†]Work done while at NIKSUN, Inc.

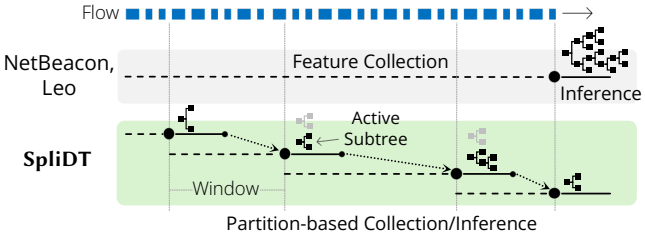

Figure 1: Comparison of in-network decision tree (DT) classification approaches. State-of-the-art methods (top) perform one-shot inference by collecting features over the entire flow duration. In contrast, SPLIDT (bottom) collects features and performs inference incrementally across partitions using windows of packets—significantly scaling the number of stateful features, while achieving higher F1 scores at line rate.

Programmable data planes—including *modern switches* (e.g., Broadcom Trident [12], Xsight X2 [84], Intel Tofino [45,46], and NVIDIA Spectrum-X [61]) and *emerging SmartNICs* (e.g., Intel IPU [50], AMD Pensando DPU [1], and NVIDIA BlueField-3 DPU [60])—with their ability to process packets at line rate, have emerged as a powerful platform for deploying ML models directly in the network [47,71,72,79,83,88,89]. By offloading inference tasks to network hardware, these data planes eliminate the need for control-plane intervention, enabling low-latency and high-performance decision-making [47,66,71,89]. Increasingly, such programmable data planes are deployed beyond traditional datacenters, including edge environments and wide-area networks (WANs) [8,30].

A major focus of recent work has been on mapping decision tree (DT) models to programmable data planes [13,47,79,83,89], mainly because of these models' interpretability and natural alignment with the (reconfigurable) match-action table (MAT) architecture [10–12,45,46,60,81,82,84]. While systems such as IIsy [83], NetBeacon [89], and Leo [47] have demonstrated the feasibility of deploying DTs in the data plane—to be able to operate within the stringent resource constraints of these programmable switches and SmartNICs—they have mainly addressed the challenge of rule explosion and concentrated on optimizing model representations (e.g., pruning DTs [47] and compressing MAT rules [83,89]).

However, despite these advancements, the critical aspect of feature collection and engineering, i.e., selecting and computing complex (stateful) input features, remains largely unexplored in this context, primarily due to the resource limitations

of underlying network hardware. Existing approaches either constrain the number of stateful features to a small, fixed set (e.g., the top-k most important features, as in NetBeacon [89] and Leo [47]) or avoid using the stateful features altogether (as in IIsy [83] and Mousika [79]). As we show in §2, these strategies result in poor model performance (e.g., reduced F1 scores) and prevent deployed DT models from capturing complex, real-world traffic patterns effectively [47, 83, 89].

This limited focus on feature engineering in prior work has its roots in two commonly made assumptions. The first assumption is that all selected features must be computed upfront before DT traversal can begin Figure 1 (top). Second, programmable data planes are assumed to be limited to executing DT models in a single, one-shot manner, prohibiting resource reuse across different portions of the DT. As a result, existing approaches view reducing the number of stateful features as the only viable solution to satisfy the given resource constraints, thus making it necessary to sacrifice model performance for scalability (i.e., supporting more concurrent flows) or vice versa [47, 89].

In this paper, we challenge these assumptions and present SpliDT, a system that enables scalable and resource-efficient deployment of DTs in the data plane by rethinking how features are computed and reused. SpliDT is built on two key insights. First, *feature computation can be deferred:* DTs do not require all features to be computed upfront. Instead, SpliDT divides the DT into partitions, where each partition—a group of consecutive layers containing one or more subtrees—computes only the features relevant to its specific subtree. This way, feature sets can vary (in size and membership) from subtree to subtree and features can be computed incrementally as the DT traversal progresses, Figure 1 (bottom). Second, *resources can be reused across subtrees:* by leveraging packet recirculation as an in-band control channel, SpliDT reuses data-plane resources (i.e., registers and match keys) between subtrees, enabling more efficient use of constrained hardware without sacrificing line rate.

SpliDT leverages these insights in an intuitive way to significantly *scale the total number of stateful features* that a DT can utilize. Instead of applying the same top-k features across the entire DT, SpliDT assigns each subtree—resulting from tree partitioning—its own set of relevant features, allowing feature selection to vary across subtrees. These subtrees are then triggered sequentially, via recirculated control packets in the data plane, reusing the stateful registers and match keys at each stage of DT traversal for the currently active subtree, Figure 1 (bottom). We demonstrate in §5 that SpliDT supports up to five times more stateful features (i.e., the total number of unique features across all subtrees) than state-of-the-art approaches [47, 89], all while achieving higher model accuracy and scaling to millions of concurrent flows at line rate.

In enabling these benefits, SpliDT must overcome two key challenges: (1) determining how to select and compute the appropriate features at runtime for each subtree during inference, and (2) designing an effective partitioning strategy at training time that balances model accuracy and hardware resource efficiency to maximize the number of supported concurrent flows in the data plane.

To address the first challenge, SpliDT processes each flow in windows of packets, specific to each subtree of a partition. Ideally, every subtree should have access to the entire flow during inference; however, since the data plane operates (and monitors traffic) at line rate without buffering, this is not feasible in practice [10, 12, 45, 46, 81, 82, 84]. Instead, SpliDT splits each flow into uniform windows,[1] allowing each subtree to observe a portion of the flow during inference, Figure 1 (bottom). Modern datacenter transport protocols (e.g., Homa [56] and NDP [40]) embed flow size information in packet headers, which can be parsed in the hardware to determine window boundaries. This information allows the data plane to halt feature collection at the designated boundary, trigger the selection of active subtrees, and transition to the next partition via recirculation.

To tackle the second challenge, SpliDT employs an iterative design search methodology integrated with a custom training framework to fine-tune DT configurations, including partitioning strategies and resource allocation policies, for specific use cases (and datasets). The goal is to optimize the trade-off between model accuracy and the number of flows that can be supported, identifying configurations that lie on the Pareto frontier. Leveraging Bayesian Optimization (BO), such as HyperMapper [57], SpliDT systematically explores the design space to identify the most effective hyperparameters, including the maximum number of features (k) per subtree, the number and size of partitions, and the overall tree depth. Intuitively, smaller values of k enable support for more flows,[2] while deeper subtrees generally improve model accuracy. Similarly, increasing the number of subtrees expands the total set of unique features across the DT model, but reduces the number of packets each subtree can observe, limiting the temporal window for feature computation. SpliDT navigates this trade-off space to derive Pareto-optimal models that balance inference accuracy and flow scalability within the constraints of the underlying hardware.

Our results (§5) demonstrate that SpliDT sets a new state-of-the-art for deploying DT models in the data plane. It consistently outperforms NetBeacon [89] and Leo [47], achieving a superior accuracy-to-flow count Pareto frontier across all levels of supported flows (Table 2). SpliDT supports 5× more stateful features, enabling richer in-network inference, all while maintaining a low recirculation overhead—just 50 Mbps (0.05%) in the worst case—and matching the low time-to-detection (TTD) performance of current systems. To

[1] Window sizes are uniform within flows and vary across flows (§3.2, §6).

[2] In Tofino1 switch [45], $k = 4$ supports up to 100,000 flows, which decreases to 65,000 with $k = 6$, and so on [47, 89]. SmartNICs (e.g., AMD Pensando DPU) exhibit similar behavior, with flow capacity dropping from about 64,000 ($k = 4$) to 40,000 ($k = 6$) [1, 60].

facilitate reproducibility and further research in this area, we have publicly released the complete artifact, including training scripts, models, and evaluation datasets [33].

2 Background & Motivation

We first review prior work on in-network classification systems that use decision tree (DT) models and highlight the need for more stateful features for DT-based inference (§2.1). We then revisit the anatomy of DTs to derive domain-specific insights (§2.2) that guide us in addressing the challenges of scaling DTs on modern programmable switches (§2.3).

2.1 The Need for More Stateful Features

As network traffic scales to multi-Tbps rates, existing approaches, such as IIsy [83] and Planter [88], aim to support real-time inference by mapping DTs onto match-action tables (MATs) while relying solely on stateless, per-packet features. These methods optimize DT representation to fit within the constraints of available switch resources (i.e., MATs) but lack flow-level context, limiting their classification accuracy and adaptability [79, 83, 88, 89]. While they efficiently handle large flow volumes in the data plane, their reliance on per-packet features significantly reduces accuracy, yielding F1 scores nearly 2× lower than models with full features access (Figure 2).

More recent work, such as NetBeacon [89] and Leo [47], improves classification accuracy by incorporating stateful features (i.e., top-k), allowing DT models to leverage flow-level context, which provides richer insights than per-packet features alone. While this approach enhances accuracy, it places substantial pressure on the limited memory resources of programmable data planes [10, 12, 45, 46, 81, 82, 84], ultimately limiting the number of flows that can be supported concurrently.

First, stateful features must be stored in registers, which share limited space with match-action tables (MATs) within each stage of the data-plane pipeline. This creates a trade-off between feature storage and model complexity. For example, in Tofino [45], allocating just four registers per flow exhausts an entire switch stage at 65K flows (or about 40K flows on a Pensando DPU-based SmartNIC [1]), preventing that stage from being used for model execution. Increasing the number of registers or supporting more flows further reduces available MAT stages, limiting DT depth and restricting feature selection. Second, adding more stateful features increases match key sizes, which inflates the size of table entries and exacerbates TCAM memory usage—making it harder to map DTs efficiently onto MATs in the data plane [45, 47, 89].

As a result, prior work has been limited to a maximum of 200K flow rules and a handful of stateful features (top-$k \leq 6$)—bounds imposed by hardware memory constraints and the need to balance register usage with model depth—yielding only moderate accuracy gains [47, 89]. Beyond this threshold, performance degrades: DT depth becomes restricted, lowering

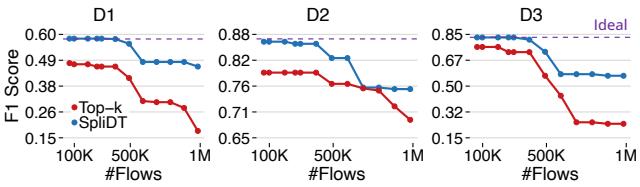

Figure 2: SPLIDT and top-$k \leq 7$ model versus the ideal scenario with unlimited resources. SPLIDT, with access to all features, achieves higher F1 score than top-k for the datasets, D1–3 (details in §5). The per-packet models peak at 0.41, 0.56, and 0.59, respectively (not shown).

classification accuracy, while larger match key sizes increase TCAM overhead, reducing scalability (§5). These trade-offs highlight the core challenge of integrating stateful features into DT-based inference without sacrificing scalability on resource-constrained programmable data planes.

Observation: The constraints of prior DT-based systems are often perceived as intrinsic to programmable data-plane architectures, but are they truly fundamental? We argue that these limitations do not stem from hardware constraints but from ingrained assumptions about how DTs should be processed. Conventional approaches assume that all stateful features must be collected before inference begins, necessitating upfront register allocation. Furthermore, these approaches treat DT execution as a single-pass, feed-forward process, confining computation to the spatially available pipeline resources.

This paper challenges the prevailing belief that feature richness and scalability must always be in conflict. We demonstrate that by decoupling stateful feature selection from DT execution, both can scale independently. Unlike prior work (e.g., NetBeacon [89] and Leo [47]), which sacrifices feature expressiveness for flow scalability, SPLIDT dynamically selects and reuses stateful features across inference steps, efficiently managing available hardware resources without restricting model complexity. Compared to traditional top-k systems, SPLIDT achieves a significantly improved Pareto frontier, simultaneously enhancing F1 scores and number of flows (Figure 2). These results challenge the notion that hardware-imposed constraints inherently limit DT scalability, showing instead that the primary bottleneck arises from rigid execution models that preallocate features and enforce single-pass processing.

2.2 Domain-Specific Properties of DTs

DT inference begins at the root node, where a decision is made based on selected features to determine the next node to visit. This process continues at each level, using different combinations of features at each step until a leaf node is reached. Instead of processing the tree level by level, we can group consecutive levels into *partitions* (Figure 3) and focus on the subtrees in each partition. With this approach, inference progresses one partition at a time, where the decisions resulting from traversing the *active* subtree in one partition

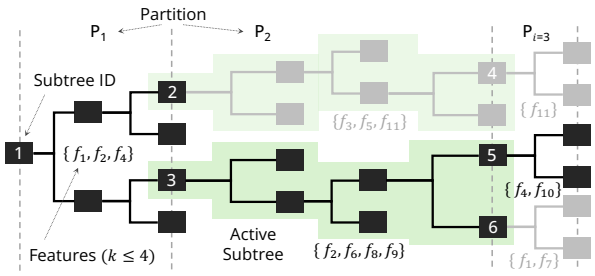

Figure 3: Domain-specific properties of DTs: Partitions (P_i) comprising multiple subtrees, each with its subset of features (k). During traversal, active subtrees are selected within each partition.

determine which subtree to traverse in the next partition. This allows for more efficient traversal, selecting only the next subtree based on relevant features and their conditions rather than evaluating entire levels sequentially.

This subtree-by-subtree execution enables features to be collected incrementally and on demand for the active subtree in each partition (Figure 1, bottom). Unlike traditional approaches that require gathering all features upfront, this method loads only the features needed for the current subtree. With just k available feature slots, we can dynamically allocate and process only the relevant features at each step, avoiding the restrictive top-k selection enforced by existing systems [47, 89]. This approach maximizes feature utilization without discarding valuable contextual information across the entire DT.

However, two key conditions must hold for this approach to be effective: (a) *feature density across subtrees*: each subtree within a partition must use at most k features out of the total N available features. Unlike traditional top-k approaches that select a fixed set of k features for the entire tree, here k represents the number of feature slots available at any given step (subtree), which can be dynamically reassigned to different features as the inference progresses. If even a single subtree requires more than k features, the benefits of incremental feature computation are lost, as more than k features would need to be allocated upfront; (b) *incremental computation architecture*: the system must support subtree-by-subtree traversal, dynamically collecting and computing features while reusing the same k feature slots at each step. This ensures efficient execution without requiring all features to be preloaded simultaneously, overcoming the limitations of prior top-k selection methods that discard all but the most globally important features across the entire DT.

To examine the feasibility of the first condition, we studied feature usage across subtrees in multiple datasets (Table 1). In the considered datasets (D1–3),[3] we found that at most only 10% of features were required in any given subtree. For instance, in dataset D1, where $N = 41$ features, subtrees required on average only 3.73 features. Note that this 11× reduction in storage requirement makes it possible to execute the entire DT using only k registers (e.g., 4 for D1) and avoids the need to impose a strict limit of k on the overall features.

[3]This trend extends beyond D1–3; similar feature sparsity holds for D4–7 (§5) and is commonly observed in real-world DT classification tasks [2, 38].

Data	Feature Density (%) / Partition	Feature Density (%) / Subtree	Recirc. Bandwidth (Mbps) E1	Recirc. Bandwidth (Mbps) E2
D1	47.15 ± 38.44	6.15 ± 2.95	2.93 ± 2.44	5.99 ± 3.51
D2	53.49 ± 44.19	7.28 ± 2.72	6.01 ± 4.01	12.32 ± 5.76
D3	53.95 ± 43.42	6.08 ± 3.37	3.58 ± 3.21	7.33 ± 4.62

Table 1: Feature density (%) across partitions and subtrees in a DT, and max. recirculation bandwidth (Mbps) when processing datasets (D1–3) for two datacenter environments, Webserver (E1) and Hadoop (E2), §5.

For the second condition, we demonstrate in the next section (§2.3) how modern programmable architectures, with support for packet recirculation [12, 45, 46, 81, 82, 84], can be reimagined as time-shared machines, enabling efficient reuse of resources (e.g., registers and match keys) across partitions within the data plane.

2.3 Switch as a Time-Shared Machine

Programmable data planes are traditionally viewed as spatial architectures with fixed resource constraints, where exceeding available resources leads to failures during program compilation [9, 10, 45]. However, we argue that because this perspective focuses mainly on the static aspects of both programmable data planes and spatial architectures, it is overly restrictive and in need of being revisited.

For example, modern data planes (switches [45, 46] and SmartNICs [81, 82]) support packet recirculation, with bandwidths reaching 100 Gbps (e.g., Tofino [45], X2 [84], and Trident [12]), without impacting line rate. This capability adds an important dynamic element to the traditional view. In effect, it enables temporal execution, allowing different stages of a program (e.g., in P4 [9] or NPL [31]) to activate across multiple recirculations in a time-shared manner. By leveraging this mechanism, the state can be distributed over time, facilitating the reuse of limited resources (e.g., registers and match keys in MATs).[4]

By carefully restructuring DTs (e.g., expressed in P4 [9] or NPL [31]), we can exploit this dynamic capability to scale DT-based inference beyond the physical limits of available resources—akin to how CPUs abstract resource constraints by reusing registers over time. Table 1 shows that for the evaluated datasets (D1–3),[3] the maximum recirculation path usage is within 20 Mbps for the two datacenter environments E1–2 (§5), significantly lower than the total available bandwidth of 100 Gbps.

In the next section, we show how SpliDT leverages the domain-specific properties of DTs and reuse of switch resources (via recirculation) to optimize the Pareto frontier (F1 score vs. number of flow rules) while supporting all stateful features at line rate.

[4]Recirculation in SpliDT is used as a fast, in-band control channel—not for forwarding data traffic—avoiding the software control plane overhead.

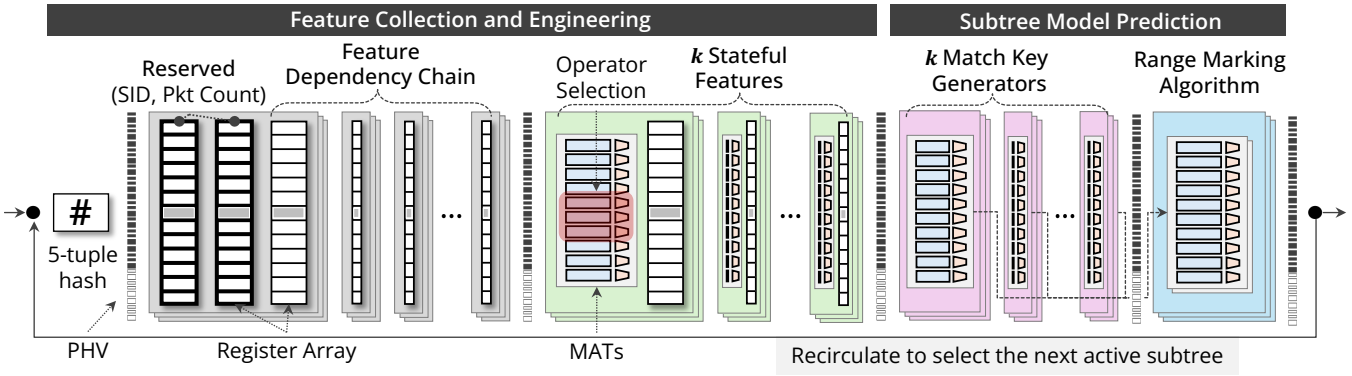

Figure 4: SPLIDT's partitioned inference architecture, processing flow windows in two phases: (1) Feature Collection and Engineering (left) and (2) Subtree Model Prediction (right)—leveraging resource reuse (i.e., registers and match keys) via recirculation, for efficient execution within each DT partition.

3 Design of SPLIDT

We now present the SPLIDT design; its *partitioned inference architecture* for efficient DT execution in the data plane (§3.1) and the *custom training framework*, for effective model configuration through design search (§3.2). We then bring these components together to show how SPLIDT enables scalable and resource-efficient inference in practice (§3.3).

3.1 Partitioned Inference Architecture

As illustrated in Figure 4, SPLIDT's partitioned inference architecture operates in two phases: (1) *Feature Collection and Engineering* (§3.1.1) and (2) *Subtree Model Prediction* (§3.1.2). For each flow, it iteratively processes windows of packets using the active subtree within each DT partition, leveraging *resource reuse* (i.e., registers and match keys) via recirculation (§3.1.3). We consider a programmable RMT-based data plane [45, 46, 50, 60] and assume flow sizes are available in packet headers [40, 56].

3.1.1 Feature Collection and Engineering. In SPLIDT, we maintain three distinct register-array sets to manage stateful feature collection and subtree selection for prediction, as shown in Figure 4. These registers include: (1) reserved state registers for tracking metadata such as the subtree ID (SID) and per flow packet counters, (2) registers for computing intermediate and dependent states (e.g., timestamps for inter-arrival time (IAT) calculations), and (3) k registers for storing stateful features specific to the active subtree within the current DT partition.

> **INFO:** *Reserved and dependency chain registers can significantly limit the number of features per subtree (k) as they must scale alongside the k features to support the same number of flows. This contention for register space creates a tradeoff that must be carefully managed to balance feature capacity and flow scalability in* SPLIDT, *§3.2.*

Upon packet arrival, SPLIDT hashes its 5-tuple using CRC32 [41] to determine the register index corresponding to the flow. First, it retrieves the subtree ID (SID) from the reserved register array and updates the packet count in the second register array. Next, depending on the use case (and dataset), some DT models require intermediate values to compute stateful features before prediction. For instance, IAT computation requires storing the previous packet's timestamp to calculate the inter-packet gap.

To support such computations, SPLIDT implements a dependency chain, a sequence of register arrays distributed across multiple pipeline stages to enable hierarchical computation. Since programmable data planes [10–12, 45, 46, 60] cannot process dependent data within a single stage, computation must be spread across multiple stages. In our evaluations, the deepest observed dependency chain was 3 stages—a depth well within the capabilities of modern data-plane devices.

Operator Selection. Since each subtree may require a different operation to compute its stateful features, SPLIDT dynamically updates the operation applied to each feature in every flow window. To achieve this, SPLIDT utilizes match-action tables (MATs) for each stateful feature, acting as selectors to apply the appropriate operation on demand (Figure 4).

At compile time (§3.2.1), SPLIDT populates these tables with rules that define which operator to apply for each subtree. The MATs match on the subtree ID (SID) and select the corresponding action to perform the necessary computation. Modern data-plane architectures support the parallel execution of multiple MATs within a stage. For instance, Tofino1 supports up to 16 MATs with 750 entries each, which is well within the requirements of our design—SPLIDT utilizes only six MATs to support $k = 6$ stateful features in our evaluations (§5), with each table containing at most 200 entries.

Lastly, to select the appropriate features to populate in the feature registers, prevent continuous feature updates on every arriving packet, and identify window boundaries, SPLIDT incorporates additional match fields in the MATs. For instance, to update a stateful feature only on SYN packets (such as SYN packet count), the MATs can include TCP flags as a match condition, ensuring the feature update is triggered only when a SYN is received.

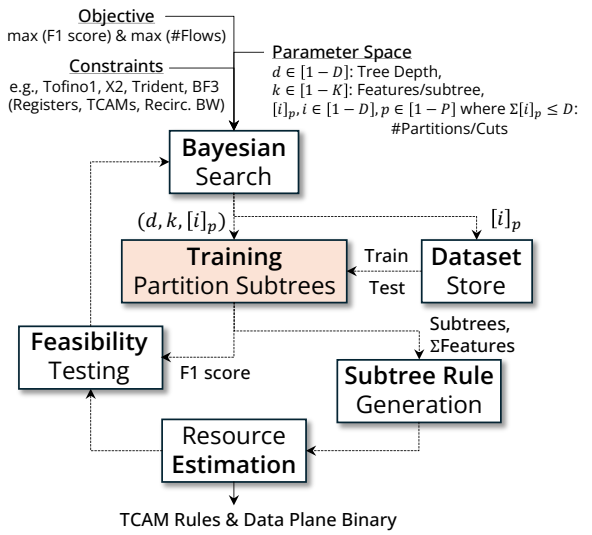

Figure 5: Workflow of SPLIDT's Model Design Search.

3.1.2 Subtree Model Prediction. The prediction phase (Figure 4) executes the decision tree (DT) model using the Range Marking Algorithm [89], which maps the partitioned DT into a sequence of MATs. The first group of MATs, the k match key generators, constructs the match keys required for the DT model. As with the operator selection tables, SPLIDT maintains a separate MAT for each feature, using precomputed rules to determine which stateful feature (register) serves as the key. Current feature values—accumulated over the most recent flow window—are stored in dedicated metadata headers. These metadata values are then used as match keys in the generator tables. Each table's action produces a range mark, a unique bit string written to a corresponding metadata field. These per-feature range marks, combined with the subtree ID (SID), form the match keys for a final MAT that implements the Range Marking Algorithm, encoding the DT model rules and performing classification.

If the current subtree is in an intermediate partition, classification yields the next SID; if it is in the final partition or an early-exit node, it outputs the flow's final class label. In the latter case, the label is sent to the controller as a digest. Otherwise, the next SID is recirculated via the resubmission channel (§3.1.3), updating the SID register and resetting the dependency chain and k stateful feature registers in preparation for the next flow window.

3.1.3 Resource Reuse via In-Band Control Channel (Recirculation). At the end of inference for each subtree, the next subtree ID (SID) is propagated to the stateful registers in the feature collection and engineering stages via resubmission, which serves as an in-band control channel [45]. A single packet—triggered after processing the corresponding flow window—is resubmitted with the updated SID encoded in a metadata header field. This resubmission imposes minimal bandwidth overhead (see Table 1), as only one control packet is required per flow window. This mechanism allows SPLIDT to perform partitioned DT inference incrementally, reusing the same pipeline resources across subtrees without introducing resource conflicts or execution overhead.

3.2 Custom Search/Training Framework

We now describe SPLIDT's design search framework for generating optimal DTs (§3.2.1) and partitioning them into subtrees using a custom training algorithm (§3.2.2).

3.2.1 SPLIDT Design Search. Figure 5 illustrates the overall workflow of the SPLIDT framework. The goal is to generate a Pareto frontier of partitioned DT configurations. Given a set of optimization objectives, input parameter ranges, and hardware/performance constraints, a Bayesian Optimization (BO) search phase begins, iteratively proposing model configurations for evaluation. For each configuration, we query a corresponding window-based training/test dataset based on the suggested number of partitions,[5] and use SPLIDT's custom partitioned DT training algorithm (§3.2.2) to train the model and evaluate its F1 score. We then generate the corresponding TCAM entries, compute hardware resource usage, determine flow scalability, and assess whether the model can be deployed at line rate on the target switch. The results—F1 score, supported flow count, and feasibility metrics—are fed back into the BO loop to guide the next iteration; this process continues for a predefined number of iterations.

In the following sections, we detail each stage of this workflow, beginning with the inputs to the BO search.

Parameter Space, Objectives, and Constraints. To initiate the design search, the user specifies three key components that define the configuration space and guide the optimization:

- *Model Hyperparameters:* These define the structure and complexity of the partitioned DT. The search space includes: the maximum tree depth (D), the number of features per subtree (k), and a list of partition sizes $[i_1, i_2, \ldots, i_p]$, where p is the number of partitions and the sum of the partition sizes equals the total tree depth, i.e., $D = \sum[i_1, i_2, \ldots, i_p]$. This enforces a uniform window size across partitions within a flow, while allowing it to vary across flows.
- *Hardware and Performance Constraints:* These reflect the capabilities of the target platform (switch or SmartNIC), including available TCAM blocks, register space, pipeline stages, and recirculation bandwidth. They ensure that candidate models are not only accurate but also deployable within the hardware's resource envelope, at line rate.
- *Optimization Objectives:* The design search jointly optimizes model accuracy (e.g., F1 score) and flow scalability (e.g., number of concurrent flows), producing a Pareto frontier that captures the trade-offs between these objectives.

Together, these inputs, along with the dataset, are fed into the Bayesian Optimization (BO) loop that drives the search for feasible and high-performing DT configurations for SPLIDT.

[5] These datasets are preprocessed offline and efficiently stored and queried using commercial databases, e.g., PostgreSQL [36] or MongoDB [43].

Bayesian Search. Given the input search space and objectives, the framework performs a design-space exploration to generate a Pareto frontier of SPLIDT DT configurations for a target dataset. Bayesian Optimization (BO) is a black-box optimization method [76] designed for optimizing expensive-to-evaluate functions [27, 53, 57]. It constructs a probabilistic surrogate model, typically a Gaussian Process (GP) or Random Forest, to approximate the objective function, and employs an acquisition function to guide the selection of promising candidates. By balancing exploration (sampling new regions) and exploitation (refining high-performing regions), BO efficiently searches high-dimensional spaces with minimal evaluations—ideal for costly tasks such as hyperparameter tuning or model training [69].

At each step, the BO search proposes several DT configurations to evaluate in parallel. Each configuration is used to train a partitioned DT using SPLIDT's custom training algorithm (§3.2.2), which is then evaluated on a test dataset. Subsequent stages assess the model's hardware resource usage, deployment feasibility, and supported flow count; these metrics are fed back to the BO loop to inform the next iteration. This process continues for a predetermined number of iterations.

Subtree Rule Generation. Given a trained partitioned DT, we generate the corresponding TCAM rules to represent the model in the data plane. We adopt the Range Marking algorithm [89], which efficiently encodes decision tree rules by mapping feature value ranges to compact ternary bit strings. It segments each feature's domain into non-overlapping ranges and assigns a unique range mark (bit string) to each, ensuring that merged ranges retain distinct and unambiguous encodings. For each feature, its thresholds from the trained partitioned DT are translated into ternary matches, with the associated range mark output actions (i.e., using P4). These TCAM entries are installed in feature tables, producing range marks as match keys for the subsequent model table (Figure 4).

A second set of TCAM rules is then generated to encode the model logic: these rules match on feature range marks and return either the next subtree ID (for intermediate partitions) or the final prediction class (at the last partition). This encoding maps each DT leaf to a single TCAM rule, effectively avoiding rule explosion.

Both feature and model TCAM entries are generated for each subtree in the partitioned DT and installed into the corresponding tables in the data-plane pipeline (Figure 4). Each rule also includes an exact match on the subtree ID (SID) to ensure the correct subtree is selected for each partition.

Resource Estimation and Feasibility Testing. To evaluate each candidate configuration, we estimate the number of TCAM blocks and pipeline stages required for the feature and model tables using a target-specific analytical model (e.g., BF-SDE [21], P4Insight [20], NetASM [67], and Nvidia DOCA P4 Developer Toolkit [59]). We then determine the number of remaining stages available for register allocation and bookkeeping logic, which directly impacts the number of flows the model can support concurrently. Recirculation overhead, in terms of in-band control traffic, is estimated using: (1) the number of partitions, which dictates the number of recirculated packets per flow; (2) flow-size distribution observed in real-world datacenter workloads (§5); and (3) the number of active flows concurrently issuing recirculations. A design is deemed feasible if it fits within the target's TCAM, register, and MAT stage budgets while keeping recirculation traffic within available bandwidth limits.

The feasibility outcome (yes/no), along with the model's F1 score and supported flow count, is fed back into the BO loop to guide subsequent iterations. At each step, the BO search proposes new parameters (and configurations), gradually converging on models that jointly maximize accuracy and flow scalability, while satisfying the target resource constraints.

Algorithm 1 SPLIDT partitioned DT training algorithm.

```
1:  procedure TrainPartitionedDT(dataset, depths, partition, k)
2:      /* k is the number of features per subtree */
3:      if partition ≥ len(depths) then
4:          return
5:      depth ← depths[partition]
6:      /* train one subtree at this partition */
7:      tree ← TrainSubtree(dataset[partition], depth, k)
8:      /* get subset of data samples for each leaf node */
9:      leaf_subsets ← PartitionSamplesByLeaves(tree, dataset)
10:     /* train subtrees for the next partition */
11:     for all (leaf, subset) ∈ leaf_subsets do
12:         /* train only if subset is non-empty */
13:         if len(subset) > 0 then
14:             TrainPartitionedDT(subset, depths, partition + 1, k)
```

3.2.2 SPLIDT Custom Training.

Algorithm 1 outlines the algorithm used to train SPLIDT's partitioned decision trees. Given the overall tree depth, partition sizes, and the number of features per subtree, training begins by learning a single subtree for the first partition using all samples from the corresponding initial window. For each leaf node in this subtree, we identify the subset of training samples that reach that node and use only those samples—along with their associated window for the next partition—to train the corresponding subtree in the following partition. Training starts with the full feature set, after which the top-k features are selected by importance and the subtree is retrained on this reduced set. Leaf nodes that do not reach the maximum depth of a partition do not spawn further subtrees, enabling SPLIDT to exit early during data-plane inference (§3.1). This recursive approach allows each subtree to specialize based on the subset of flow packets it receives, enabling window-based inference matching the data distribution observed during training.

3.3 Putting It All Together

To demonstrate how SPLIDT's design search and partitioned inference architecture operate in practice, we walk through a complete end-to-end example.

The process begins with the user providing a labeled dataset, model hyperparameters, resource constraints, and op-

Dataset	Description	Classes
D1: CIC-IoMT2024	A cybersecurity dataset [16] with Internet of Medical Things (IoMT) traffic for intrusion detection in healthcare.	19
D2: CIC-IoT2023-a	A simplified version of the CIC-IoT-2023 dataset [17], categorized into four primary classes of IoT traffic.	4
D3: ISCX-VPN2016	A dataset containing VPN and non-VPN traffic [18] for evaluating VPN detection and privacy-related analyses.	13
D4: CampusTraffic	UCSB campus dataset [38] containing various application types, including web, cloud, social, and, streaming traffic.	11
D5: CIC-IoT2023-b	A comprehensive IoT dataset [17] containing multi-class network traffic data for evaluating IoT security threats.	32
D6: CIC-IDS2017	A network intrusion detection dataset [14] for various attack scenarios, including DoS, DDoS, and brute force.	10
D7: CIC-IDS2018	An anomaly detection dataset [15] capturing network traffic for diverse attacks and benign activities.	10

Table 2: Real-world network traffic datasets used for evaluating SPLIDT across diverse security scenarios.

timization objectives (§3.2.1). SPLIDT then invokes Bayesian Optimization (BO) to explore the configuration space and identify a Pareto-optimal model that jointly maximizes F1 score and flow scalability. For one such data point (Figure 3), the selected model had a tree depth of $D = 6$, with 4 features per subtree and partition sizes of $[2, 3, 1]$ across 3 partitions, supporting up to 1M concurrent flows (§5).

Given this configuration (Figure 3), SPLIDT uses its custom training algorithm (§3.2.2) to train 6 subtrees: subtree 1 in the first partition, subtrees 2–3 in the second, and subtrees 4–6 in the third. Some subtrees are skipped due to early exits. SPLIDT then generates TCAM rules for each subtree using the Range Marking Algorithm [89], and compiles target-specific code (e.g., P4) for feature extraction and inference logic. These rules are installed into the data plane's MATs: feature tables encode stateful features into range marks, while model tables use them to either classify flows or select the next subtree (Figure 4). All subtrees are present on the switch, but only one is active per flow at a time, requiring just 4 feature registers per flow.

Referring to the example in Figure 3, at runtime, a new flow begins with subtree ID, SID = 1, in partition P_1. The switch collects the features required by this subtree (e.g., f_1, f_2, f_4) over the first window of packets. Once the window concludes, the subtree predicts SID = 3 in partition P_2, prompting a single recirculation that updates the flow's SID and clears its feature and dependency-chain registers. The second window is processed using the features needed by subtree 3 (e.g., f_2, f_6, f_8, f_9). Following this, the model selects subtree 5 in partition P_3, and another recirculation updates the SID = 5. The third window is then processed with the features required by subtree 5 (e.g., f_4, f_{10}), and the final prediction is emitted. No further recirculations are needed.

This pipeline executes independently per flow, with register state indexed via 5-tuple hashing to avoid conflicts. In this example, SPLIDT supports 1 million concurrent flows, each maintaining only 4 stateful feature registers and a single SID register—demonstrating its scalability and efficiency under practical resource constraints.

4 Implementation

We implement the SPLIDT framework in Python `v3.8.10`, allowing seamless integration with widely used libraries such as Pandas [62], Scikit-Learn [63], and TensorFlow [73]. At the core of SPLIDT's optimization process is HyperMapper [57] (`commit:3dfa8a7`), which uses Bayesian Optimization (BO) to efficiently search for the best model configurations. Unlike other BO frameworks such as OpenTuner [3], HyperOpt [7], and GPflow Opt [49], HyperMapper supports multi-objective optimization, diverse parameter types (real, integer, categorical), and feasibility testing [57]. These features allow us to optimize SPLIDT models for multiple goals simultaneously, such as maximizing F1 score and flow capacity. The feasibility testing feature also helps eliminate DT configurations that exceed available switch resources (e.g., TCAM limits) or introduce excessive resubmission traffic. The BO experiments are configured via YAML, specifying datasets, parameter space, objectives, and switch constraints.

To train and test SPLIDT's partitioning strategy, we use the `DecisionTreeClassifier` class from Scikit-Learn [63] and recursively generate subtrees for all partitions. We employ the Range Marking Algorithm, as described in NetBeacon [89], to generate TCAM rules. We parallelize evaluations using Python's `ProcessPoolExecutor` from the `concurrent.futures` module to speed up the BO search. For data-plane inference, we implement SPLIDT in P4 [9] and compile it using BF-SDE `v9.11.0` for the Tofino1 [45] switch. Finally, to install TCAM rules into the switch, we use `bfrt_grpc` client API in Python.

In total, the SPLIDT framework consists of 4,700 lines of Python code. The P4 implementation and controller required 1,600 and 240 lines, respectively. Each experiment, corresponding to a dataset, was defined using 400 lines of YAML, totaling 2,800 lines for all seven datasets (D1–7).

5 Evaluation

We evaluate SPLIDT DTs for their end-to-end classification performance and resource utilization (§5.2) and perform microbenchmark experiments (§5.3).

5.1 Experiment Setup

Testbed Environment. Our testbed includes two servers, one as a traffic generator and the other as a receiver, connected through an Edgecore Wedge 100-32X Tofino1 switch [45] running Stratum OS [28].[6] Each server has a 64-core Intel Xeon Platinum 8358P CPU @2.60 GHz, 512 GB RAM, and runs Proxmox VE `v8.0.4` [64]. They are equipped with dual-port

[6] SPLIDT can operate on any architecture that supports RMT-like pipelines with stateful registers. We use Tofino1 [45] in our evaluation, but the design is not limited to it.

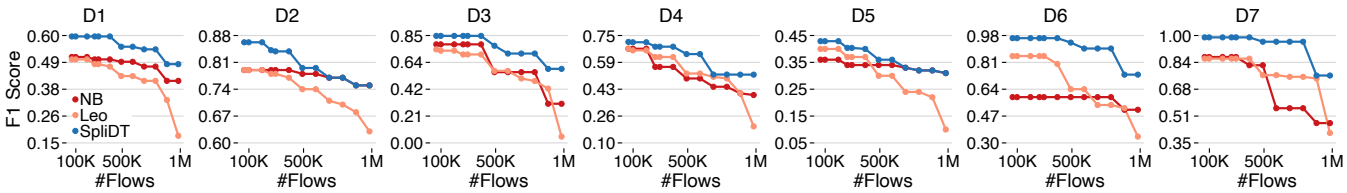

Figure 6: Pareto frontier of SPLIDT vs. baselines, indicating the best F1 score for a given #flows in the data plane.

Data	#Flows	F1 Score			Depth / #Partitions			#Features			#TCAM Entries			Register Size (bits)		
		NB	Leo	SPLIDT	NB	Leo	SPLIDT	NB	Leo	SPLIDT	NB	Leo	SPLIDT	NB	Leo	SPLIDT
D1	100K	0.51	0.50	0.60	13	11	24 / 5	6	6	20	6,596	16,384	4,583	192	192	160
	500K	0.49	0.43	0.55	13	6	21 / 5	4	4	21	6,154	2,048	4,509	128	128	96
	1M	0.41	0.18	0.48	12	3	13 / 1	2	2	2	1,534	2,048	8,238	64	64	64
D2	100K	0.79	0.79	0.86	12	11	45 / 3	6	6	30	8,479	16,384	17,083	192	192	160
	500K	0.78	0.74	0.80	12	6	27 / 3	2	4	30	11,996	2,048	12,002	64	128	64
	1M	0.75	0.63	0.75	18	3	12 / 1	1	2	2	2,540	2,048	465	32	64	64
D3	100K	0.78	0.73	0.85	13	11	21 / 2	6	5	23	5,056	16,384	2,562	192	160	160
	500K	0.56	0.57	0.77	12	6	16 / 5	4	4	28	1,566	2,048	2,931	128	128	64
	1M	0.31	0.05	0.59	4	3	18 / 4	2	1	15	170	2,048	928	64	32	32
D4	100K	0.67	0.66	0.71	13	10	28 / 2	6	7	27	11,361	8,192	14,507	192	224	160
	500K	0.49	0.52	0.64	10	11	20 / 2	4	2	26	4,355	16,384	12,934	128	64	64
	1M	0.39	0.20	0.51	9	3	27 / 2	2	1	1	4,221	2,048	393	64	32	32
D5	100K	0.36	0.40	0.43	10	10	49 / 5	6	7	20	7,755	8,192	27,302	192	224	96
	500K	0.34	0.30	0.36	12	6	33 / 4	2	4	21	30,891	2,048	21,433	64	128	64
	1M	0.31	0.10	0.31	7	3	11 / 1	2	2	2	2,063	2,048	639	64	64	64
D6	100K	0.59	0.85	0.96	5	10	15 / 5	4	4	28	308	8,192	587	128	128	160
	500K	0.59	0.64	0.94	5	7	23 / 3	3	3	16	354	2,048	578	96	96	96
	1M	0.51	0.34	0.73	8	3	10 / 4	2	2	9	403	2,048	174	64	64	32
D7	100K	0.87	0.86	0.99	5	10	26 / 5	6	6	17	251	8,192	191	192	192	160
	500K	0.82	0.76	0.96	8	7	10 / 6	4	3	10	495	2,048	106	128	96	64
	1M	0.47	0.41	0.76	5	3	10 / 6	2	2	7	71	2,048	76	64	64	32

Table 3: Model performance vs. resource usage tested against Tofino1 switch (6.4 Mbits TCAM budget, 12 stages) [45].

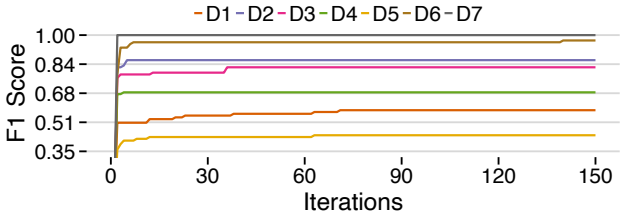

Figure 7: Number of BO search iterations to reach peak F1 score. All datasets converge within 150 iterations.

Intel XL710 [44] 10/40G NICs and Nvidia ConnectX-6 [58] 100G NICs, and use MoonGen [26] for traffic generation and reception. The same servers run our Bayesian Optimization (BO) experiments, with 500 iterations and 16 parallel evaluations per iteration for each dataset.

Baselines, and Real-World Applications and Environments. Our baselines include Leo [47] and NetBeacon [89], two state-of-the-art data plane DT implementations. Both baselines support stateful (top-k) DTs and improve flow scalability with respect to tree depth via efficient TCAM rule generation. To ensure fairness, we allocate the full switch pipeline (all stages) to each baseline, including ours, and evaluate the best-performing model each can support using all available hardware resources. Table 2 summarizes the seven real-world datasets (D1–7) used in our evaluation, spanning use cases such as intrusion detection, traffic classification, and modern attack detection (e.g., DoS, botnets, infiltration). Together, these datasets allow for a robust evaluation of the performance of SPLIDT DTs against baselines across diverse network security tasks. We also evaluate the volume of resubmitted traffic under two representative data center workloads—E1: Webserver [65], with many long-lived flows, and E2: Hadoop [65], characterized by short, bursty mice flows.

Dataset Generation. To generate per-flow packet windows for training SPLIDT DTs, we extended the widely-used CICFlowMeter tool [32,34,51]. By default, CICFlowMeter identifies flows by 5-tuples, computes 78 flow-level features, and emits statistics only at the FIN packet, discarding intermediate data. We modified it to output statistics at every window boundary (e.g., each quarter of a flow for 4 partitions) and reset flow state after each window. While NetBeacon's *phases* resemble SPLIDT's windows, they differ in two key aspects: (a) phase intervals grow exponentially (i.e., 2, 4, 8, 16, ...), and (b) flow statistics are retained across phases, resulting in the same top-k features being reused. For each dataset in Table 2, we generate up to 7 partitions[7] for SPLIDT. For NetBeacon [89], we adopt the same phase intervals as their public artifact.

[7]Classification accuracy significantly drops beyond 7 partitions.

Stages	D1	D2	D3	D4	D5	D6	D7
Fetch	0.9s	0.32s	0.01s	0.07s	0.91s	0.24s	0.18s
Training	556s	228s	10s	84s	725s	163s	111s
Optimizer	33s	45s	43s	43s	30s	32s	37s
Rulegen	0.8s	0.99s	0.91s	1.08s	0.71s	0.71s	0.91s
Backend	42μs	45μs	43μs	42μs	46μs	44μs	47μs
Time	589s	273s	54s	128s	756s	196s	148s

Table 4: Average time per iteration across different stages of the SPLIDT framework.

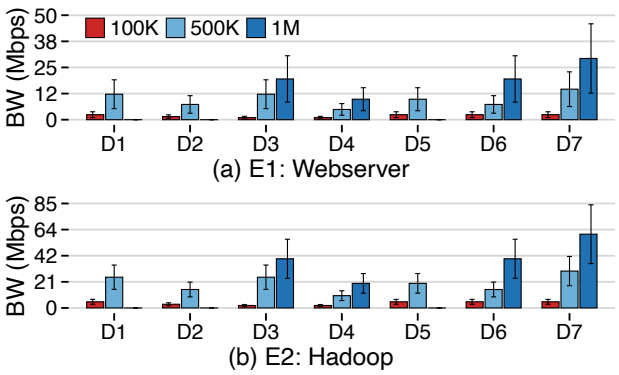

Figure 8: Maximum recirculation bandwidth (Mbps) of SPLIDT partitioned trees when processing datasets (D1–7) for the two datacenter environments, E1: Webserver and E2: Hadoop, with a varying number of flows. A model with a single partition does not recirculate packets (0 Mbps).

5.2 End-to-End Analysis

Pareto Frontier. Across all datasets, SPLIDT DTs consistently outperform the baselines by achieving a superior tradeoff—delivering higher accuracy at the same flow count. This consistent ability to balance model performance (i.e., accuracy) with scalability (i.e., number of flows) allows SPLIDT to define the Pareto frontier across all evaluated datasets (Figure 6). The resulting tradeoff curves are monotonically decreasing: models yield higher accuracy when supporting fewer flows and progressively trade off feature coverage and model complexity to scale to larger flow counts.

Feature Scalability and Resource Utilization *(TCAMs, Registers, and Recirculation Bandwidth).* As shown in Table 3, SPLIDT DTs consistently deliver the highest accuracy across datasets, balancing tree depth and partitioning effectively. They typically use less register space while maintaining competitive accuracy, and optimize TCAM usage with manageable register overhead, well within the capabilities of modern programmable switches [45, 46] and SmartNICs [1, 50, 60]. For instance, in D6, SPLIDT supports more features[8] than both baselines across all flow sizes (within 160-, 96-, and 32-bit budgets for 100K, 500K, and 1M flows, respectively), while also deploying deeper trees. As shown in Figure 8 and Figure 11, SPLIDT's packet recirculation remains well within the 100 Gbps resubmission buffer capacity and does not degrade model responsiveness, matching NetBeacon [89] and Leo [47] in per-flow time-to-detection (TTD). These results underscore the efficiency of SPLIDT DTs in maximizing accuracy under the same resource constraints as competing baselines.

Offline Software Overhead. SPLIDT's offline design search reaches maximum accuracy for all datasets within 150 iterations (Figure 7). As shown in Table 4, training dominates the per-iteration cost (88% on average), followed by the BO stage (12%), which leverages HyperMapper [57] to guide the search toward high-performing models. Training is computationally intensive due to the recursive nature of SPLIDT DTs, which involves deeper trees, more subtrees, and repeated dataset partitioning. Despite this, the full search completes in an average of 3.8 hours per dataset. Crucially, this process is performed offline and only once per use case; as in large-scale ML systems, training may be expensive but incurs no data-plane packet-processing overhead, yielding fully optimized, deployment-ready models [35, 42].

5.3 Microbenchmarks

SPLIDT *framework finds optimal model parameters for partitioned decision trees.* The SPLIDT DSE framework navigates the parameter space for tree partitioning to identify high-performing models for a given number of flows. Figure 9a shows the Pareto frontiers for fixed tree depths (10, 20, 30) as we vary the features per subtree and partition count. For instance, in D2, a depth of 30 generally yields a better frontier than depth 10—except at high flow counts (e.g., 1M flows), where the simpler model performs comparably. Each dataset benefits from a different tree depth depending on its feature complexity and class distribution, and SPLIDT effectively adapts to these differences. Figure 9b fixes the number of partitions (1, 3, 5), showing that fewer partitions often yield better frontiers due to more packets per window, enhancing model accuracy. Lastly, Figure 9c varies the number of features per subtree (1, 2, 3): using more features improves accuracy but reduces scalability; fewer features increase flow support at the cost of classification performance.

SPLIDT *DTs outperform baselines for any given TCAM budget.* Figure 10 shows that SPLIDT DTs consistently achieve higher accuracy than the baselines across different TCAM budgets. This improvement stems from reducing the match key size by decreasing the number of features (k) used in the model table for the active subtree.

SPLIDT *DTs recirculate limited traffic without impacting end-to-end flow-level time-to-detection (TTD).* Figure 8 shows that in both environments: E1 and E2, packet recirculation overhead of SPLIDT DTs stays well within the resubmission buffer capacity of 100 Gbps. Additionally, since packet processing latency is fixed in RMT-based switches [45, 46], we evaluate the per-flow time-to-detection (TTD)—defined as the time from the start of tree traversal to the final infer-

[8] We describe the per-model features in Section A.

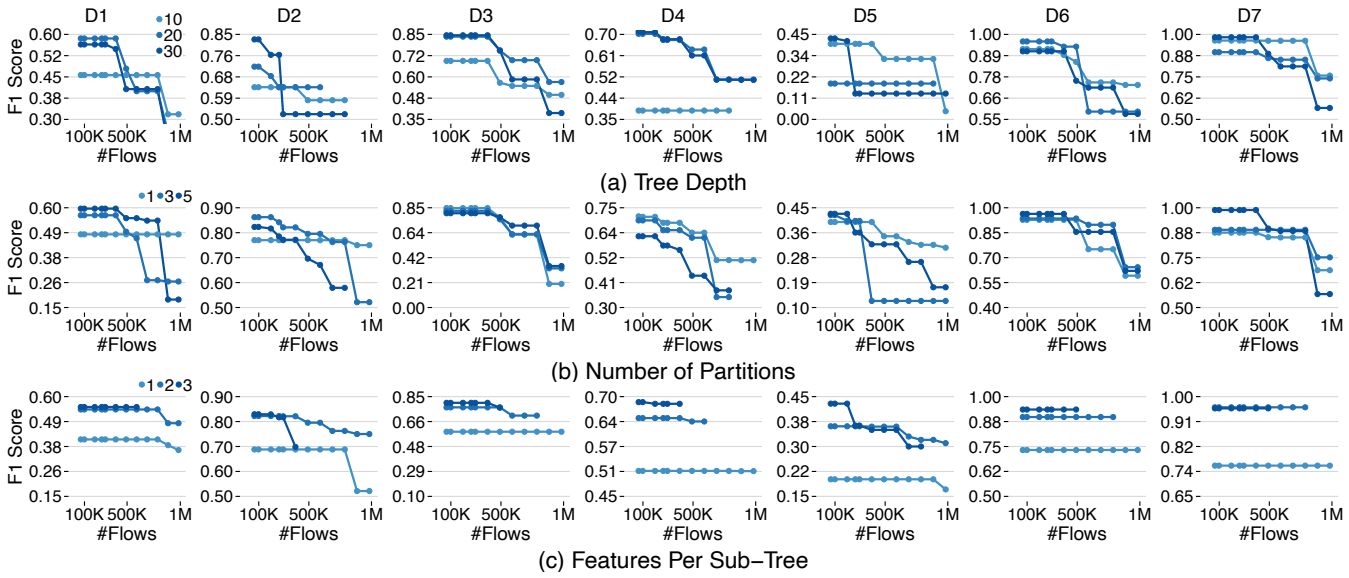

Figure 9: Pareto frontiers for SPLIDT partitioned trees under varying constraints (top to bottom): (a) fixed tree depth, (b) fixed number of partitions, and (c) fixed number of features per subtree.

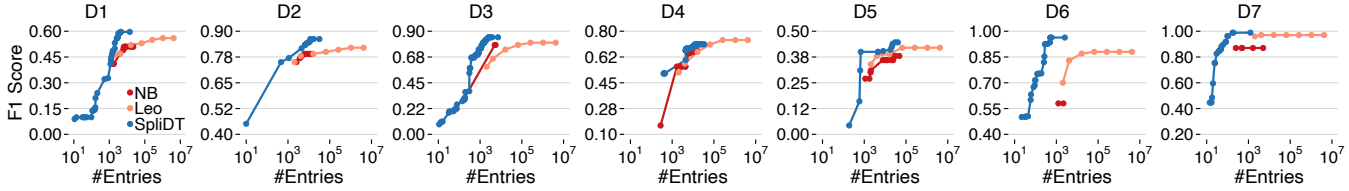

Figure 10: Comparison of #TCAM entries against F1 score for SPLIDT versus baselines.

ence decision—as shown in Figure 11 for D3 which closely matches that of NetBeacon and Leo, while sustaining 9% and 16% higher F1 score, respectively. This indicates that recirculation overhead in SPLIDT DTs does not negatively impact model performance or responsiveness.

SPLIDT *DTs scale feature count with constant register space.* SPLIDT DTs maintain a constant register footprint, based solely on k features per subtree, regardless of the total number of unique features used across the entire tree (Figure 12). As shown in Table 3, SPLIDT can store up to 30 distinct 32-bit features within a 128-bit register budget. This demonstrates that by dynamically multiplexing features across partitions and subtrees, SPLIDT minimizes register overhead. This efficient use of stateful memory enables the deployment of deeper, more complex DTs in the data plane.

SPLIDT *DTs can accommodate a larger number of flows with reduced feature bit precision while maintaining higher F1 scores.* We lower the bit precision of all features from 32 bits to 16 and 8 bits, respectively, and measure the resulting impact on accuracy and flow scalability. Figure 13 shows that this reduction affects all models similarly, as they are all decision trees. On average, we observe a 7% (14%) drop in accuracy with 16-bit (8-bit) precision, while the maximum number of supported flows increases to 2M (4M). In both cases, SPLIDT continues to yield a better Pareto frontier than the baselines, due to its enhanced feature coverage.

6 Discussion & Limitations

Windowing Strategy and Extensions. Although SPLIDT adopts a fixed partitioning strategy, the effective window size varies across flows based on flow size, while remaining consistent across partitions within a given flow. Our evaluation shows that using equal-sized, per-flow windows across partitions is sufficient to achieve robust performance (§5), while avoiding the exponential expansion of the design search space required to explore unequal window sizes within a flow (§3.2).

Deployment Assumptions and Threat Model. SPLIDT targets trusted data center environments in which programmable switches are assumed to be secure, packet headers are not adversarially spoofed, and attackers lack physical access to networking hardware. Under these assumptions, flow-level metadata maintained in registers and packet headers is treated as trustworthy during inference, consistent with the threat models adopted by prior in-network ML systems.

7 Related Work

Decision Tree (DT)-Based Inference in the Data Plane. Decision tree inference has been widely studied in programmable data planes. Prior work has explored optimized match–action table (MAT) representations and encodings (Leo [47], NetBeacon [89]), resource-efficient binary DTs via knowledge

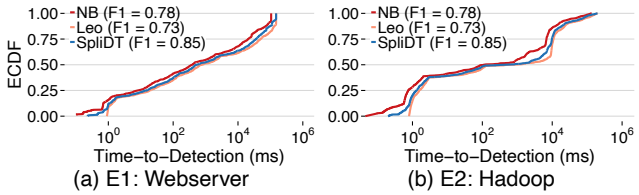

Figure 11: Time-to-detection (TTD) of D3 for environments: E1 and E2. Other datasets show a similar trend.

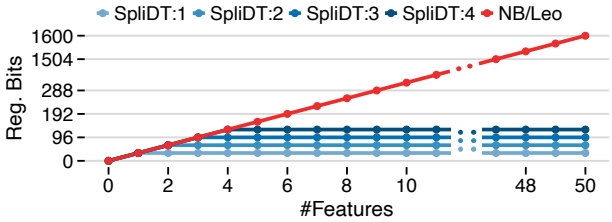

Figure 12: Register sizes (in bits) versus number of features supported by each model. SPLIDT:*k* is a partitioned tree with *k* features per subtree.

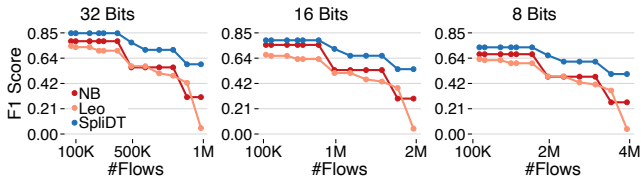

Figure 13: Pareto frontier of D3 versus bit precisions.

distillation (Mousika [79]), classification using packet-level summaries (Planter [88]), and general DT-to-MAT mappings (IIsy [83]). Extensions to ensemble models further support random forests with dynamic, traffic-driven feature selection (pForest [13]). However, these approaches typically rely on static top-*k* feature constraints and single-pass DT execution. In contrast, SPLIDT enables dynamic feature allocation across tree partitions and introduces incremental inference via subtree transitions, optimizing stateful feature storage and scaling to millions of flows with reduced TCAM overhead.

Neural Network (NN)-Based Inference in the Data Plane. NN-based approaches offer high expressiveness but face resource constraints in programmable data planes. Prior work spans deployment frameworks (Taurus [71], Homunculus [72]), expressive sequential models (Brain-on-Switch (BoS) [87]), and efficiency-oriented designs such as binary or accelerator-assisted NNs (N3IC [70], ServeFlow [54], ACDC [48]). To support these workloads, recent platforms integrate dedicated inference hardware, including SmartNICs (NVIDIA BlueField-3 [60]) and on-chip neural engines (Broadcom Trident 5's NetGNT [11]). While effective, these approaches rely on specialized hardware, limiting model flexibility, resource sharing, and online adaptation, and often require careful hardware–model co-design. In contrast, SPLIDT targets resource-constrained, line-rate execution on existing data planes, enabling scalable, high-accuracy inference without external accelerators by partitioning decision trees and reusing MAT stages and registers via recirculation.

Lookaside Accelerators for ML Inference. Modern ML inference increasingly relies on high-performance accelerators such as GPUs [22], FPGAs [81, 82], TPUs [19], and SmartNICs [1, 60]. These platforms excel at offline or near-real-time inference, motivating *lookaside* architectures that offload packets or flow metadata to external co-processors for ML inference [37, 68, 78]. However, while effective for sampled traffic and background analytics, lookaside inference is ill-suited for high-speed networks requiring line-rate processing with microsecond-level latency [11, 47, 89]. External accelerators cannot consistently meet data-plane throughput and latency demands, often forcing packet sampling and reducing real-time detection coverage. In contrast, SPLIDT enables scalable, line-rate DT inference entirely within the programmable data plane, eliminating lookaside offload and dependence on external accelerators while ensuring low-latency execution on the switch fast path.

8 Conclusion

SPLIDT recasts decision tree (DT) deployment in programmable data planes through partitioned, window-based inference. Unlike prior approaches that rely on static top-*k* feature sets and one-shot execution, SPLIDT enables incremental inference via subtree transitions and dynamic feature reuse. Combined with a Bayesian optimization (BO)-based training pipeline, SPLIDT balances model accuracy and hardware efficiency, scaling to millions of flows at line rate.

SPLIDT shows that complex and expressive in-network ML inference is not only feasible but practical—without compromising model accuracy or performance. As network traffic grows in volume and speed, SPLIDT offers a scalable and efficient path to real-time, high-throughput inference under tight hardware constraints. By enabling informed in-network decision-making, SPLIDT contributes to ongoing efforts toward a more responsible use of ML in networking.

Acknowledgements

We sincerely thank our shepherd, Hamed Haddadi, and the anonymous reviewers for their valuable feedback, and Sripath Mishra for his assistance during the early stages of this work. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by NSF Awards CAREER 2521510 and 2443777 and CNS 2521196 and 2323229; by Israel Science Foundation Grant 980/21; by a Google Research Scholar Award; and by research gifts from Cisco and Google. The opinions, findings, conclusions, and recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of NSF, DARPA, NIKSUN, or our collaborators' companies (including AMD/Xilinx and Intel/Barefoot). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] AMD. Pensando. https://www.amd.com/en/accelerators/pensando, last accessed: 06/18/2025.

[2] Blake Anderson and David McGrew. Machine Learning for Encrypted Malware Traffic Classification: Accounting for Noisy Labels and Non-Stationarity. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[4] Alireza Bahramali, Amir Houmansadr, Ramin Soltani, Dennis Goeckel, and Don Towsley. Practical Traffic Analysis Attacks on Secure Messaging Applications. In *NDSS*, 2020.

[5] Diogo Barradas, Nuno Santos, Luis Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *NDSS*, 2021.

[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review (CCR)*, 2010.

[7] J. Bergstra, D. Yamins, and D. D. Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *ICML*, 2013.

[8] Abhik Bose, Diptyaroop Maji, Prateek Agarwal, Nilesh Unhale, Rinku Shah, and Mythili Vutukuru. Leveraging Programmable Dataplanes for a High Performance 5G User Plane Function. In *Proceedings of the 5th Asia-Pacific Workshop on Networking*, 2021.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM Computer Communication Review (CCR)*, 2014.

[10] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[11] BROADCOM. Trident 5 / BCM78800 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800, last accessed: 06/18/2025.

[12] BROADCOM. Trident4/BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, last accessed: 06/18/2025.

[13] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. *arXiv preprint arXiv:1909.05680*, 2022.

[14] Canadian Institute for Cybersecurity. CIC IDS 2017 Dataset. https://www.unb.ca/cic/datasets/ids-2017.html, last accessed: 06/18/2025.

[15] Canadian Institute for Cybersecurity. CIC IDS 2018 Dataset. https://www.unb.ca/cic/datasets/ids-2018.html, last accessed: 06/18/2025.

[16] Canadian Institute for Cybersecurity. CIC IoMT 2024 Dataset. https://www.unb.ca/cic/datasets/iomt-dataset-2024.html, last accessed: 06/18/2025.

[17] Canadian Institute for Cybersecurity. CIC IoT 2023 Dataset. https://www.unb.ca/cic/datasets/iotdataset-2023.html, last accessed: 06/18/2025.

[18] Canadian Institute for Cybersecurity. CIC VPN Dataset. https://www.unb.ca/cic/datasets/vpn.html, last accessed: 06/18/2025.

[19] Google Cloud. Tensor Processing Units (TPUs). https://cloud.google.com/tpu, 2025.

[20] Intel Corporation. Intel P4 Insight. https://p4.org/onf-product/intel-p4-insight/, last accessed: 06/18/2025.

[21] Intel Corporation. Intel® P4 Studio. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html, last accessed: 06/18/2025.

[22] NVIDIA Corporation. NVIDIA T4 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/tesla-t4/, 2025.

[23] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-Architecting Congestion Control for Consistent High Performance. In *USENIX NSDI*, 2015.

[24] Yutao Dong, Qing Li, Kaidong Wu, Ruoyu Li, Dan Zhao, Gareth Tyson, Junkun Peng, Yong Jiang, Shutao Xia, and Mingwei Xu. HorusEye: A Realtime IoT Malicious Traffic Detection Framework using Programmable Switches. In *USENIX Security*, 2023.

[25] R. Doriguzzi-Corin, S. Millar, S. Scott-Hayward, J. Martínez-del Rincón, and D. Siracusa. Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection. *IEEE Transactions on Network and Service Management*, 2020.

[26] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, 2015.

[27] Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *ICML*, 2018.

[28] Open Networking Foundation. Stratum OS. https://www.opennetworking.org/stratum/, last accessed: 06/18/2025.

[29] Chuanpu Fu, Qi Li 0002, and Ke Xu 0002. Detecting Unknown Encrypted Malicious Traffic in Real Time via Flow Interaction Graph Analysis. In *NDSS*, 2023.

[30] Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai. Sirius: Composing Network Function Chains into P4-Capable Edge Gateways. In *USENIX NSDI*, 2024.

[31] Saikrishna Garlapati. Network Programming Language (NPL) Specification. https://www.scribd.com/document/430082948/Network-programming-Language-NPL, last accessed: 06/18/2025.

[32] Gerard Drapper Gil, Arash Habibi Lashkari, Mohammad Mamun, and Ali A Ghorbani. Characterization of encrypted and VPN traffic using time-related features. In *Proceedings of the 2nd international conference on information systems security and privacy*, 2016.

[33] Github. SpliDT Decision Trees. https://splidt-decision-trees.github.io/, last accessed: 02/06/2026.

[34] Arash Habibi Lashkari (GitHub). CICFlowMeter. https://github.com/ahlashkari/CICFlowMeter/tree/master, last accessed: 06/18/2025.

[35] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*. MIT Press, Cambridge, 2016.

[36] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org, last accessed: 06/18/2025.

[37] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven network telemetry. In *ACM SIGCOMM*, 2018.

[38] Satyandra Guthula, Roman Beltiukov, Navya Battula, Wenbo Guo, Arpit Gupta, and Inder Monga. netFound: Foundation Model for Network Security. *arXiv preprint arXiv:2310.17025*, 2025.

[39] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 2008.

[40] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *ACM SIGCOMM*, 2017.

[41] He3 Team. Understanding the CRC32 Hash: A Comprehensive Guide. https://he3.app/blogs/understanding-the-crc32-hash-a-comprehensive-guide/, last accessed: 06/18/2025.

[42] ibm. What are Large Language Models (LLMS)? https://www.ibm.com/think/topics/large-language-models, last accessed: 06/18/2025.

[43] MongoDB Inc. MongoDB. https://www.mongodb.com, last accessed: 06/18/2025.

[44] Intel. Intel Ethernet Network Adapter X710. https://www.intel.com/content/www/us/en/products/details/ethernet/700-network-adapters/x710-network-adapters/products.html, last accessed: 06/18/2025.

[45] Intel. Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, last accessed: 06/18/2025.

[46] Intel. Tofino2: Second-generation P4-programmable Ethernet Switch ASIC that Continues to Deliver Programmability without Compromise. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, last accessed: 06/18/2025.

[47] Syed Usman Jafri, Sanjay Rao, Vishal Shrivastav, and Mohit Tawarmalani. Leo: Online ML-based Traffic Classification at Multi-Terabit Line Rate. In *USENIX NSDI*, 2024.

[48] Xi Jiang, Shinan Liu, Saloua Naama, Francesco Bronzino, Paul Schmitt, and Nick Feamster. AC-DC: Adaptive Ensemble Classification for Network Traffic Identification. *arXiv preprint arXiv:2302.11718*, 2023.

[49] Nicolas Knudde, Joachim van der Herten, Tom Dhaene, and Ivo Couckuyt. GPflowOpt: A Bayesian Optimization Library Using TensorFlow. *arXiv preprint arXiv:1711.03845*, 2017.

[50] Patricia Kummrow. The IPU: A New, Strategic Resource for Cloud Service Providers. https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/The-IPU-A-New-Strategic-Resource-for-Cloud-Service-Providers/post/1335081, last accessed: 06/18/2025.

[51] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of tor traffic using time based features. In *International Conference on Information Systems Security and Privacy*, 2017.

[52] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[53] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. SMAC3: A versatile Bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research (JMLR)*, 2022.

[54] Shinan Liu, Ted Shaowang, Gerry Wan, Jeewon Chae, Jonatas Marques, Sanjay Krishnan, and Nick Feamster. ServeFlow: A Fast-Slow Model Architecture for Network Traffic Analysis. *arXiv preprint arXiv:2402.03694*, 2024.

[55] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*, 2017.

[56] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*, 2018.

[57] Luigi Nardi, Bruno Bodin, Sajad Saeedi, Emanuele Vespa, Andrew J Davison, and Paul HJ Kelly. Algorithmic Performance-accuracy Trade-off in 3D Vision Applications using Hypermapper. In *IEEE IPDPSW*, 2017.

[58] Nvidia. ConnectX-6 Network Adapters. https://www.nvidia.com/en-in/networking/ethernet/connectx-6-dx/, last accessed: 06/18/2025.

[59] Nvidia. DOCA Documentation. https://docs.nvidia.com/doca/archive/2-9-2/doca+p4+developer+tools/index.html, last accessed: 06/18/2025.

[60] Nvidia. Nvidia BlueField Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, last accessed: 06/18/2025.

[61] NVIDIA Corporation. NVIDIA Spectrum-X: Ethernet Networking Platform for AI. https://www.nvidia.com/en-us/networking/spectrumx/, last accessed: 06/18/2025.

[62] pandas. pandas. https://pandas.pydata.org/, last accessed: 06/18/2025.

[63] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 2011.

[64] Proxmox. Proxmox. https://www.proxmox.com/en/, last accessed: 06/18/2025.

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM*, 2015.

[66] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *APNet*, 2019.

[67] Muhammad Shahbaz and Nick Feamster. The case for an intermediate representation for programmable data planes. In *SOSR*, 2015.

[68] Chaofan Shou, Rohan Bhatia, Arpit Gupta, Rob Harrison, Daniel Lokshtanov, and Walter Willinger. Query planning for robust and scalable hybrid network telemetry systems. *Proceedings of the ACM on Networking*, 2024.

[69] Sebastian Simon, Nikolay Kolyada, Christopher Akiki, Martin Potthast, Benno Stein, and Norbert Siegmund. Exploring Hyperparameter Usage and Tuning in Machine Learning Research. In *IEEE/ACM 2nd International Conference on AI Engineering–Software Engineering for AI (CAIN)*, 2023.

[70] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. Re-architecting Traffic Analysis with Neural Network Interface Cards. In *USENIX NSDI*, 2022.

[71] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for Per-Packet ML. In *ASPLOS*, 2022.

[72] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *ASPLOS*, 2023.

[73] Tensorflow. Tensorflow. https://www.tensorflow.org/, last accessed: 06/18/2025.

[74] Wei Wang, Ming Zhu, Xuewen Zeng, Xiaozhou Ye, and Yiqiang Sheng. Malware traffic classification using convolutional neural network for representation learning. In *International Conference on Information Networking (ICOIN)*, 2017.

[75] Feng Wei, Hongda Li, Ziming Zhao, and Hongxin Hu. xNIDS: Explaining Deep Learning-based Network Intrusion Detection Systems for Active Intrusion Responses. In *USENIX Security*, 2023.

[76] Wikipedia. Bayesian Optimization. https://en.wikipedia.org/wiki/Bayesian_optimization, last accessed: 06/18/2025.

[77] Keith Winstein and Hari Balakrishnan. TCP ex machina: Computer-generated Congestion Control. In *ACM SIGCOMM Computer Communication Review (CCR)*, 2013.

[78] Wenji Wu and Phil Demar. A GPU-accelerated network traffic monitoring and analysis system. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013.

[79] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable General In-Network Intelligence in Programmable Switches by Knowledge Distillation. In *IEEE INFOCOM*, 2022.

[80] Renjie Xie, Jiahao Cao, Enhuan Dong, Mingwei Xu, Kun Sun, Qi Li, Licheng Shen, and Menghao Zhang. Rosetta: Enabling Robust TLS Encrypted Traffic Classification in Diverse Network Environments with TCP-Aware Traffic Augmentation. In *USENIX Security*, 2023.

[81] Xilinx. Alveo SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/sn1000-product-brief.pdf, last accessed: 06/18/2025.

[82] AMD Xilinx. Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html, last accessed: 06/18/2025.

[83] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *ACM HotNets*, 2019.

[84] Xsight Labs. X2 Programmable Ethernet Switch. https://xsightlabs.com/products/, last accessed: 06/18/2025.

[85] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: A Randomized Experiment in Video Streaming. In *USENIX NSDI*, 2020.

[86] Francis Y Yan, Jestin Ma, Greg D Hill, Deepti Raghavan, Riad S Wahby, Philip Levis, and Keith Winstein. Pantheon: The Training Ground for Internet Congestion-Control Research. In *USENIX ATC*, 2018.

[87] Jinzhu Yan, Haotian Xu, Zhuotao Liu, Qi Li, Ke Xu, Mingwei Xu, and Jianping Wu. Brain-on-switch: towards advanced intelligent network data plane via NN-driven traffic analysis at line-speed. In *USENIX NSDI*, 2024.

[88] Changgang Zheng, Mingyuan Zang, Xinpeng Hong, Liam Perreault, Riyad Bensoussane, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. Planter: Rapid prototyping of in-network machine learning inference. *ACM SIGCOMM Computer Communication Review (CCR)*, 2024.

[89] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. An Efficient Design of Intelligent Network Data Plane. In *USENIX Security*, 2023.

A Candidate and Selected Switch Features For Each Model Per Dataset.

Table 5 lists the set of candidate switch features that can be offloaded in programmable RMT-like data planes (e.g., packet counts, aggregate sizes, and inter-arrival times). From this space, SPLIDT's custom training/search framework (§3.2) automatically selects a subset of features to use in the models for each dataset (D1–7). The table reports these selected features, with marked entries indicating which features were chosen by each model.

B Artifact Appendix

B.1 Artifact Overview

This artifact contains the full implementation of SPLIDT, including the training framework, partitioned decision tree generation, HyperMapper-based design space exploration, and evaluation pipelines used to produce all experimental results in the paper. It is publicly available on our project website at https://splidt-decision-trees.github.io/ and on GitHub at https://github.com/SpliDT-Decision-Trees/SpliDT-Artifact-NSDI26.

B.2 Contents

The artifact is provided as a self-contained repository available at https://github.com/SpliDT-Decision-Trees/SpliDT-Artifact-NSDI26.

- The `dse-and-training-framework` repository contains the SPLIDT's training framework, experimental pipelines, and scripts necessary to reproduce Figures 6–13 and Tables 3–4.
- The `README.md` provides detailed instructions for running the training pipeline, executing the experiments, and reproducing our microbenchmarks.

B.3 Requirements

The `dse-and-training-framework` is implemented in Python `v3.8.10`. All required libraries and dependencies are specified in `environment.yml`, which defines the complete Conda environment used by the framework. This file ensures that the same package versions and runtime environment can be reproduced consistently across different systems. The environment can be created by running `conda env create -f environment.yml`, which installs all required dependencies automatically.

Feature	D1			D2			D3			D4			D5			D6			D7		
	100K	500K	1M	100K	500K	1M	100K	500K	1M	100K	500K	1M	100K	500K	1M	100K	500K	1M	100K	500K	1M
Destination Port	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓
Flow Duration	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓				
Total Forward Packets	✓	✓		✓	✓		✓	✓		✓	✓		✓	✓			✓			✓	✓
Total Backward Packets	✓	✓		✓	✓			✓		✓	✓		✓	✓		✓	✓				
Forward Packet Length Total	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	
Backward Packet Length Total	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓			✓	✓
Forward Packet Length Min.	✓	✓		✓	✓			✓	✓	✓	✓		✓	✓			✓		✓	✓	
Backward Packet Length Min.	✓	✓		✓	✓		✓	✓		✓	✓		✓	✓		✓		✓	✓		✓
Forward Packet Length Max.	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	
Backward Packet Length Max.	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓		✓		
Flow IAT Max.				✓	✓		✓	✓		✓	✓					✓			✓		
Flow IAT Min.				✓	✓		✓	✓		✓	✓					✓					
Forward IAT Min.				✓	✓		✓	✓		✓	✓					✓			✓		
Forward IAT Max.				✓	✓		✓	✓		✓	✓					✓					
Forward IAT Total				✓	✓		✓	✓		✓	✓					✓			✓		
Backward IAT Min.				✓	✓		✓	✓		✓	✓					✓					
Backward IAT Max.				✓	✓		✓	✓		✓	✓					✓					
Backward IAT Total				✓	✓		✓	✓		✓	✓					✓					
Forward PSH Flag		✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓				✓	
Backward PSH Flag																					
Forward URG Flag																					
Backward URG Flag																					
Forward Header Length	✓	✓		✓	✓			✓		✓	✓		✓	✓		✓	✓	✓	✓		
Backward Header Length	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓			✓		
Min. Packet Length	✓	✓		✓	✓			✓	✓				✓	✓		✓			✓		
Max. Packet Length	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
FIN Flag Count	✓	✓		✓	✓			✓		✓	✓			✓		✓					
SYN Flag Count	✓	✓		✓	✓	✓							✓	✓	✓	✓	✓				
RST Flag Count				✓	✓					✓						✓					
PSH Flag Count	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓		
ACK Flag Count	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓
URG Flag Count																					
CWR Flag Count																			✓		
ECE Flag Count																					
Forward Act Data Packets	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓		✓	✓				
Forward Segment Size Min.	✓	✓		✓	✓		✓	✓	✓				✓	✓		✓	✓	✓	✓	✓	✓

Table 5: Feature coverage across datasets (D1–7) with varying number of flows (100K, 500K, 1M).