Leo: Online Traffic Classification at Multi-Terabit Line Rate

Syed Usman Jafri
Purdue University

Sanjay Rao
Purdue University

Vishal Shrivastav
Purdue University

Mohit Tawarmalani
Purdue University

Abstract

Online traffic classification enables critical applications such as network intrusion detection and prevention, providing Quality-of-Service, and real-time IoT analytics. However, with increasing network speeds, it has become extremely challenging to analyze and classify traffic online. In this paper, we present Leo, a system for online traffic classification at multi-terabit line rates. At its core, Leo implements an online machine learning (ML) model for traffic classification, namely the decision tree, in the network switch's data plane. Leo's design is fast (can classify packets at switch's line rate), scalable (can automatically select a resource-efficient design for the class of decision tree models a user wants to support), and runtime programmable (the model can be updated on-the-fly without switch downtime), while achieving high model accuracy. We implement Leo on top of Intel Tofino switches. Our evaluations show that Leo is able to classify traffic at line rate with nominal latency overhead, can scale to model sizes more than twice as large as state-of-the-art data plane ML classification systems, while achieving classification accuracy on-par with an offline traffic classifier.

1 Introduction

In recent years, there have been increasing calls for a "self-driving" network [8–10], i.e., a network driven by real-time analytics performed on data at line-rate guided by programmatic control. Self-driving networks can help in tasks such as anomaly detection (e.g., identifying and isolating malicious traffic), and performance monitoring and repair (e.g., identifying flows that see sub-optimal traffic, and rerouting them). A key component of such self-driving networks is the need to run machine learning (ML) inferencing algorithms. For example, network security applications rely on ML algorithms for classifying applications and devices (e.g., IoT device) [19], and detecting anomalous patterns [13, 17].

Traditionally, ML inference algorithms today are run off the network path. For instance, Intrusion Detection Systems (IDS) typically require exporting network data to an off-path IDS device, which runs necessary ML models and flags anomalies. On the one hand, the bandwidth needed to export data from routers is significant. On the other hand, reaction times to take action to resolve security issues is slow and of the order of minutes. Other systems such as Intrusion Prevention require individual packets to be sent off-path and only packets deemed safe are then forwarded. Doing so can incur significant latencies for all packets, which limits the wide-spread adoption of such intrusion prevention devices.

The recent advent of programmable switches offers the unique opportunity of running ML inference algorithms directly on the data plane [24]. For instance, a lighter weight ML model run on a switch could let the vast majority of traffic pass through the switch without delays, with only a smaller portion of the traffic routed to the control plane, where the traffic could be inspected with more sophisticated traffic models. While programmable switches offer promise, there are several challenges as well. First, switches are significantly resource constrained in terms of computation (ALU resources), memory, and pipeline stages. Second, while ML models change frequently, switches offer limited runtime programmability, with many changes requiring a switch reboot.

In this paper, we take a key step towards running ML inference algorithms in the data plane. We focus on decision trees given they are widely used in traffic classification, are interpretable, and since they only require features (e.g., comparisons, conditions) that are already available on existing programmable switches. Further, although there is ongoing research on new hardware support for switches [20, 23, 25], our focus is on realizing decision trees on existing switches for ease of deployability considerations.

We make the following **contributions:**

- We analyze initial proposals [5,12,22,24,26], notably [5,24] to support decision trees on programmable switch pipelines. The majority of these approaches [5, 12, 22] follow the natural dependencies of a decision tree, implying they are bottlenecked by the number of stages in a switch for larger trees. While [24, 26] decouples the number of switch stages from tree dependencies, we show that its memory requirements are prohibitive whether SRAM or TCAM is used (§3).
- We present Leo, a system that provisions a programmable switch pipeline to support an entire class of trees in a runtime programmable fashion (e.g., all trees with depth $\leq D$ and at most L leaves). Leo exploits the fact that although the decision tree itself could be large, any individual data packet only encounters a small subset of tree nodes. Further, it selectively flattens portions of a decision tree to judiciously trade off memory and ALU requirements on the one hand, and the number of stages on the other hand. To achieve these goals, we have developed sub-tree multiplexing as part of Leo which allows multiple sub-trees in a decision tree to be multiplexed into an execution layer with ALUs whose features and constraints can be programmed at runtime. This simultaneously allows programmability across trees, and reduces resource requirements for a given tree.
- We present worst-case bounds on resource requirements

(e.g., table sizes) with Leo, to ensure it can sufficiently provision for all trees in a class. We show that unlike IIsy [24, 26] Leo achieves attractive worst-case bounds on memory requirements, while requiring significantly fewer stages than [5, 12, 22].

• Our evaluations on Intel Tofino [2] switches show (i) Leo support much larger models than existing approaches –e.g., it supports complete trees of depth 10 with SRAM, while existing approaches are limited to depth 5 trees; with TCAMs, Leo supports trees of depth 22 with upto 1024 leaves; (ii) on real IDS datasets, Leo achieves classification accuracies comparable to a control plane solution, and significantly better than accuracies achieved with prior work [5, 12, 26] and (iii) evaluations on a real programmable switch testbed show that classification latencies with Leo are 500X lower than performing classification in the router control plane. Overall, the results show Leo is a viable approach to support packet classification in the data plane.

2 Background and Motivation

ML for traffic classification. Consider Intrusion Detection and Prevention systems (IDS and IPS) systems, which match network flows to patterns to identify potential anomalies. IPS systems inspect individual data packets synchronously and take necessary preventive action on malicious traffic (e.g., block traffic). In contrast, IDS systems operate asynchronously over traffic exported from the data plane. While many IDS/IPS systems rely on rules based on packet payload, the increasing trend towards encrypted traffic has spurred interest in behavioral systems that do not rely on packet payloads. Instead these systems only rely on models of network flow statistics (e.g., packet size distributions, inter-arrival times etc.). Behavioral models may also serve as an initial coarse filter that flags potentially malicious traffic to a rule-based systems that performs more detailed inspection. Although we use intrusion detection and prevention as our primary motivating example, many other use cases such as IoT device identification, application classification and QoE inference [13, 17, 19] benefit from ML classification models.

Why traffic classification in data plane? With increasing network speeds, the amount of data that needs to be analyzed per unit time has also increased. For example, a single state-of-the-art switch can receive multi-terabits of data per second. Unfortunately, implementing traffic classification in the control plane or remote servers incurs high response latency, which might be unacceptable for synchronous systems such as IPS that require packet processing on the critical path. Our measurements on a real router testbed show that while it only takes few hundreds of nanoseconds to process a packet in the switch's data plane, it could take hundreds of microseconds to simply send a packet to the switch's local control plane and back. Further, the bandwidth of the data path between a switch's local control and the data plane is typically a few 10s of Gbps (implemented using PCIe bus). However, the data plane of state-of-the-art switches runs at multi-Tbps. Thus, even for an asynchronous system such as an IDS, it is not possible to analyze and classify each packet going through the network using the control plane. The alternative is to heavily sample packets, or report digests over longer epochs, which can reduce accuracies and responsiveness. Using a remote server for analysis and classification suffers from the same limitations, namely high latency for classification and server bandwidth acting as bottleneck. These factors motivate us to explore running traffic classification in the data plane.

Programmable switches. Network data plane support for ML inference is facilitated by the recent emergence of programmable switches which allow the architect to deploy programs that are executed on each data packet. Many programmable switches today follow the Protocol Independent Switch Architecture (PISA) data plane model and comprise a parser, processing pipeline, and deparser, which are each programmable [4]. The processing is done by a pipeline of a fixed number of *stages* [6, 7] that execute on every clock cycle. Each stage consists of:

• *Match tables,* which specify the packet header fields to match against, and the corresponding action (e.g., rewrite a packet header field). Match tables may be supported using (i) *SRAMs* which only support exact matches; or (ii) *TCAMs* which support wild card matches (e.g, all packets with source port 80, and arbitrary destination port). TCAMs are more flexible, but they are power hungry and more expensive.

• *Registers and ALUs.* Registers store small amounts of state that persists across packets (e.g., to implement packet counters). Stateful ALUs involve registers and allow computations across packets. Stateless ALUs only take other packet headers as inputs. Computation is limited to fields within a packet.

In addition, each switch consists of a Packet Header Vector (PHV) which contains packet header fields, and metadata used to communicate intermediate results across stages.

3 Design goals and prior work limitations

In this paper, we focus on decision trees for ML classification given that they are widely used, and are easy to interpret unlike "black-box" approaches such as neural networks. Further, the simplicity of a decision tree model (e.g., unlike neural networks, decision trees do not require complex ALU operations such as multiplication) makes them amenable for implementation on existing high-speed data plane architectures. In the rest of this section, we discuss our design goals in mapping decision trees to programmable switch data planes, and why prior attempts [5, 12, 22, 24, 26] fall short.

In mapping an ML structure to the data plane, our **goals** are:

Support a class of models in a runtime programmable fashion. ML policy changes over time as new data is available and models are retrained. Unfortunately, existing switches offer limited runtime programmability (i.e., ability to make changes without rebooting a switch). For instance, while the

	Runtime Prog	Not limited by tree dependency	Implementable in ASIC switch	Low ALU usage	Low memory usage
Infocom [22]	✗	✗	✓	✗	✓
pForest [5]	✓	✗	✗	✓	✓
SwitchTree [12]	✓	✗	✗	✓	✓
IIsy [24,26]	✓	✓	✓	✓	✗
Leo	✓	✓	✓	✓	✓

Table 1: Comparing `Leo` to prior work. §3 presents analysis to show the high memory requirements of `IIsy`, and we analyze `Leo`'s resource requirements in §6.

entries of match tables can be modified at runtime by the control plane, the inputs of an ALU cannot. Ideally, any approach must support an entire class of decision trees without requiring a router reboot (e.g., all trees with a particular depth).

Support sufficiently large models for high accuracy. The accuracy of classification depends on factors such as tree depth and the number of leaves. Figure 1 shows for a real IDS dataset (details in §7) that deeper trees achieve higher accuracy despite using the same number of decision nodes.

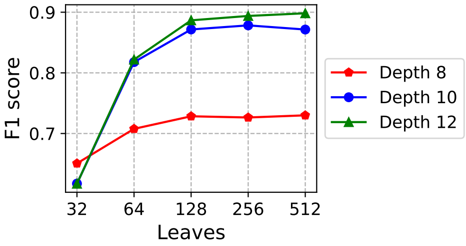
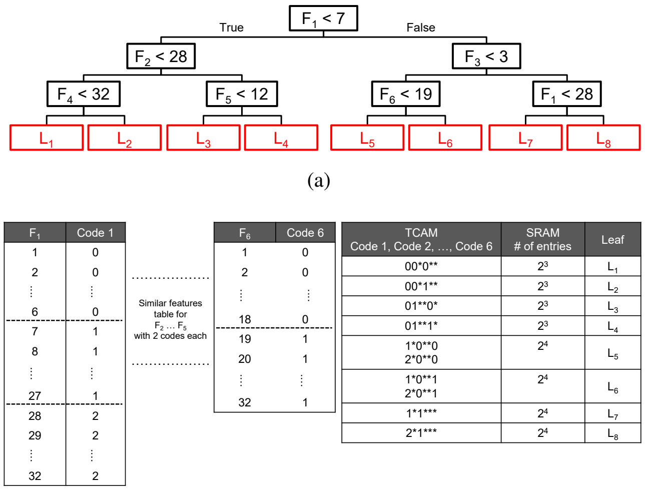


Figure 1: Impact of increasing depth for the same number of leaves using CICIDS-2017 dataset.

Resource efficient for all ML models in a class. The resources required in programmable switches must be acceptable in terms of resources such as ALUs, SRAM, TCAM, and the number of stages. Further, since we are provisioning for a class of ML models (e.g., all decision trees with a particular depth), the mapping must be such that the **worst-case** bounds for mapping for all models within the class is acceptable in terms of resource requirements. The worst-case bounds must scale acceptably -e.g., with the number of leaf nodes in a decision tree, or with tree depth.

Prior work and limitations. Unfortunately, existing approaches [5, 12, 22, 24, 26] to support decision trees in the data plane fall short of the above goals. First, the majority of these works [5, 12, 22] follow a natural strategy, which involves mapping the hierarchical structure of decision trees to the programmable switch pipeline. This requires at least one (and possibly more) stages per tree level. Consequently, tree depth is bottlenecked by the number of pipeline stages. The results are further exacerbated given our analysis above which indicates that sparse and deeper trees are more important to improve classification accuracy. Besides this central limitation, some of these works are not runtime programmable (e.g., [22] uses simple conditionals, and even a minor change would require rebooting the switch to deploy recompiled code), and have high ALU usage (e.g., [22] uses an ALU per decision tree node), while others [5, 12] have only been implemented on software BMv2 switches.



(a)

F_1	Code 1
1	0
2	0
⋮	⋮
6	0
7	1
8	1
⋮	⋮
27	1
28	2
29	2
⋮	⋮
32	2

F_6	Code 6
1	0
2	0
⋮	⋮
18	0
19	1
20	1
⋮	⋮
32	1

TCAM Code 1, Code 2, …, Code 6	SRAM # of entries	Leaf
00*0**	2^3	L_1
00*1**	2^3	L_2
01**0*	2^3	L_3
01**1*	2^3	L_4
1*0**0 2*0**0	2^4	L_5
1*0**1 2*0**1	2^4	L_6
1*1***	2^4	L_7
2*1***	2^4	L_8

(b)

Figure 2: Illustrating `IIsy`'s approach for an example decision tree with 6 features each with a maximum value of 32. Feature F1 appears twice and hence its values are mapped to three code words, while other features are mapped to two code words. The final table has entries for all code word combinations. The rules are shown for a TCAM, but the number of entries needed with an SRAM is also shown.

While `IIsy` [24, 26] addresses many of these shortcomings, we analyze it extensively below, and show that its memory requirements can be prohibitive, whether SRAM or TCAM is used. Table 3 summarizes existing schemes. `Leo` addresses these limitations with a design that is runtime programmable, not constrained by tree dependencies, has acceptable memory and ALU usage, and implemented on a hardware switch. In the rest of the section, we show `IIsy` has unacceptably high memory requirements.

Analyzing memory requirements with `IIsy`. `IIsy` [24,26] seeks to break intrinsic tree dependencies by (i) a table per feature which maps feature values into a smaller set of code words. The mapping is such that the final classification result is the same for all values of a feature that share the same code word, no matter what the values of other features are. A final classification table looks at every possible combination of code words across features, and maps them to a classification result. We present two results to show that `IIsy`'s memory requirements grow exponentially with the number of features, whether SRAM or TCAM is used.

Proposition 1 *Consider the requirement that `IIsy` must sup-*

port all trees with a depth D or lower, involving any subset of N pre-determined features with each feature taking values in the range $[0,K]$ in a runtime programmable fashion. Then, the total SRAM that must be provisioned to achieve this goal grows exponentially with N.

Proof sketch. To derive conservative bounds on the size of the combination table, consider a complete decision tree of depth D where each feature appears in the same number of decision tree nodes. Let $I = 2^D - 1$ denote the number of internal nodes. The total number of decision nodes that involve each feature is $\frac{I}{N}$, requiring $\frac{I}{N}+1$ codewords per feature. Since the combination table includes combinations of all possible codewords associated with each feature, the total size is $(\frac{I}{N}+1)^N$ which is exponential in N. Further, each feature table requires K entries since each table explicitly enumerates all values of every feature. Thus, NK total entries is needed across the tables. In practice, some savings is possible owing to default rules (see Appendix A.1). Accounting for this, we obtain the following conservative bound on the the total number of SRAM entries, where the first term is the requirement for feature tables, and the second term the requirement for the combination table.

$$N*(K-\lceil\frac{K}{2^D}\rceil)+\frac{2^D-1}{2^D}*(\frac{2^D-1}{N}+1)^N \qquad (1)$$

We next show `IIsy`'s memory requirements grow exponentially even with TCAMs. focusing on the combination table since this is most crucial to the analysis.

Proposition 2 *There exist a family of decision trees with $O(N^2+NK)$ leaves which require at least $O(\lg^{N-1}(K-1))$ TCAM rules with* `IIsy`*, where N is the number of features, and each feature has values that could range from* $1\ldots K$

Proof sketch. The proof is based on a family of decision trees shown in Figure 15 in the Appendix for the general case with N features $(F_1\ldots F_N)$ with each feature having values ranging from $1\ldots K$. To provide more intuition, Figure 3 shows a geometric representation for the special case with $N=3$, and $K=4$, where the highlighted cubes correspond to the leaves of the corresponding decision tree.

The intuition behind the tree construction is as follows. First, the decision tree nodes has leaves for each value of a feature F_j when all other features are at their maximum value K. This forces `IIsy` to use a distinct code word for each value of every feature. Next, the decision tree has leaf nodes which correspond to regions where some of the features can take multiple possible values. These nodes will require a large number of code word combinations with `IIsy` since it is forced to use a distinct code word for each feature value. For example, in Figure 3, `IIsy` requires $K=4$ code words for each feature (owing to the decision tree nodes shown by the small cubes). While the inner cube (A_2) which captures the region where all features are $< K$ corresponds to a single decision tree node, it would requires $(K-1)^3$ combinations with `IIsy`. Likewise, each of the regions $A_{1,2}$, $A_{1,1}$, and B_1 (where exactly one feature is K) correspond to single decision tree nodes but would each require $(K-1)^2$ different code word combinations. Even if a default rule were used for the inner cube, $3*(K-1)^2$ code word combinations must still be explicitly covered, which would require at least $3m^2$ TCAM entries where $m=\lceil\log(K-1)\rceil$. We defer the details of the proof for the general case to the Appendix §A.2 along with models for estimating the size of the feature and combination tables that must be provisioned for a TCAM implementation.

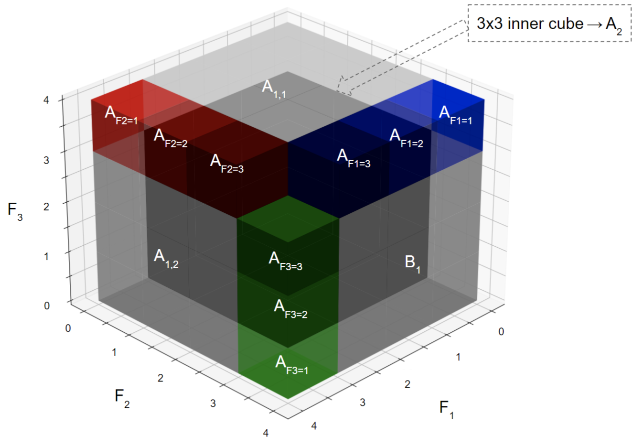


Figure 3: Visualizing the general decision tree for three features.

Implications. Using equation (1), a complete tree of Depth 6 using 9 16-bit features requires over 10^8 SRAM entries with `IIsy`. Using the analysis in Appendix §A.2. we can construct a tree with only 200 leaves that needs over 10^6 TCAM entries

4 `Leo` Design

In this section, we present the design of `Leo`, a resource-efficient and runtime programmable system that enables data plane traffic classification on PISA switch pipelines.

4.1 Abstraction

Having a well-defined abstraction for a decision tree model is crucial to provide the right balance between flexibly switching between different decision trees at runtime and worst-case resource and performance overhead. The two extreme abstractions of supporting a *specific* decision tree and supporting *all* possible decision trees compromises either on flexibility or on resource constraint respectively. To that end, `Leo` provides the following abstraction for a decision tree model: ***to support any tree from a "class" of decision trees specified by the three tuple (D, L, F) where D is the maximum tree depth, L is the maximum number of leaves, and F is the set of features that the tree nodes can use, subject to the switch resource constraints while allowing for switching between trees of the same class at runtime without switch downtime.***

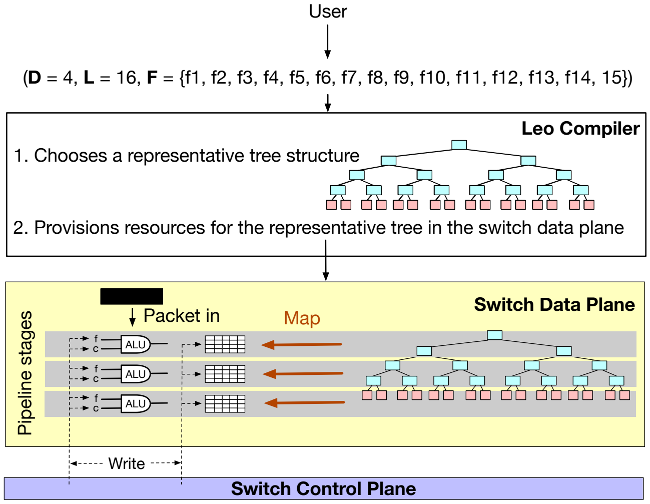

User
(D = 4, L = 16, F = {f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, 15})
Leo Compiler
1. Chooses a representative tree structure
2. Provisions resources for the representative tree in the switch data plane
Packet in
Map
Switch Data Plane
Pipeline stages
f
c
ALU
Write
Switch Control Plane

Figure 4: Workflow of Leo. Given a user specified (D, L, F) class, Leo chooses a representative decision tree for that class and maps it to the switch pipeline using multiplexed ALUs and SRAM/TCAM tables (Figure 9). The table inputs are configured at runtime by the switch control plane to implement different decision trees in the class (D, L, F) (Figure 16).

The user can specify a decision tree model to map to the switch pipeline using the tuple (D, L, F). Leo's compiler will try to find a mapping that can express *any* tree in the given class, subject to the switch resource constraints. However, the compiler will return a failed mapping if one or more trees in the given class could not be mapped within the resource constraints of the switch pipeline. If the mapping is successful, the user can choose any tree from the given class to run on the switch, and can switch between trees within the same class at runtime. Note that at any given time, the switch runs exactly one tree from the given class. The above abstraction is general enough to also express other common abstractions for a decision tree model. For example, $(D, 2^D, F)$ can express all decision trees with depth D or smaller.

4.2 Design Overview

The workflow for Leo's design is shown in Figure 4.

Representative decision tree. Given a (D, L, F) specification, Leo provisions for a *single* decision tree structure at compile time that can be multiplexed at runtime to implement any tree in the class. We refer to this tree as a *representative decision tree* A natural choice for a representative decision tree structure would be a complete tree of depth D (and 2^D leaves), as any tree structure with depth $\leq D$ would simply be a sub-tree of the complete tree. For example, in Figure 4, for the $(D = 4, L = 16, F)$ class, Leo's compiler chooses a complete decision tree of depth 4 as the representative tree structure. However, for classes where $L < 2^D$, this approach over-provisions the resources. In §6, we discuss how Leo reduces the resources to provision by also considering the maximum number of leaves L within the (D, L, F) class.

Mapping Representative Tree to Switch. Next, the compiler efficiently maps the tree structure to the switch pipeline. In doing so, it addresses two key challenges.

Challenge 1. The ability to update the features and constraints of the representative decision tree nodes at runtime to implement different trees in a (D, L, F) class.

Challenge 2. The ability to support decision tree classes with large D (depth) and L (leaves), subject to the resource constraints of the switch.

Leo solves these challenges with two novel techniques it introduces called *decision tree node multiplexing*(§4.3) and *sub-tree flattening and multiplexing* (§4.4)

Runtime reconfiguration. Finally, at runtime, the switch control plane configures the feature and constraint values in each node of the representative decision tree structure to implement a specific decision tree in the class (D, L, F) (§4.5).

4.3 Decision Tree Node Multiplexing

An internal node of a decision tree is implemented using an ALU whose inputs are a feature value $f \in F$ and a constraint c. The output is the result of a comparison (e.g., $f < c$). The set of features F are defined by the user as part of Leo's abstraction (D, L, F), and are stored either in the packet header vector (for stateless features, e.g., TCP SYN flag) or inside the switch registers (for stateful features, e.g., TCP packet count). The constraints are stored in the switch memory and are populated (and updated at runtime) by the control plane.

To support different decision trees in a given (D, L, F) class, Leo allows that *any* internal node in the decision tree can take *any* of the F features and *any* constraint value as inputs for the conditional statements. To achieve this, Leo implements an internal node of the representative decision tree using a **multiplexed ALU**, an abstraction of which is shown in Figure 5. Conceptually, this comprises a (i) *feature multiplexer,* that can select any of the features at runtime from the set F; (ii) a *constraint multiplexer,* that can select any of the stored constraints; and (iii) an ALU that operates on the selected feature and constraint. We discuss concrete implementation in §5.

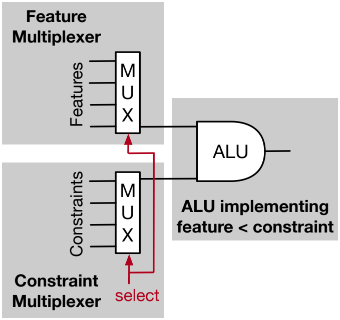


Figure 5: An abstract multiplexed ALU.

Supporting all conditional statements. At compile time, each node in the representative decision tree implements a statement of the form $f < c$. However, a decision tree can have other conditionals (e.g., $f > c$ or $f \leq c$). Unfortunately,

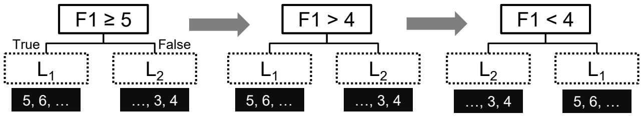


Figure 6: Transforming a $\geq$ node to an equivalent $<$ node.

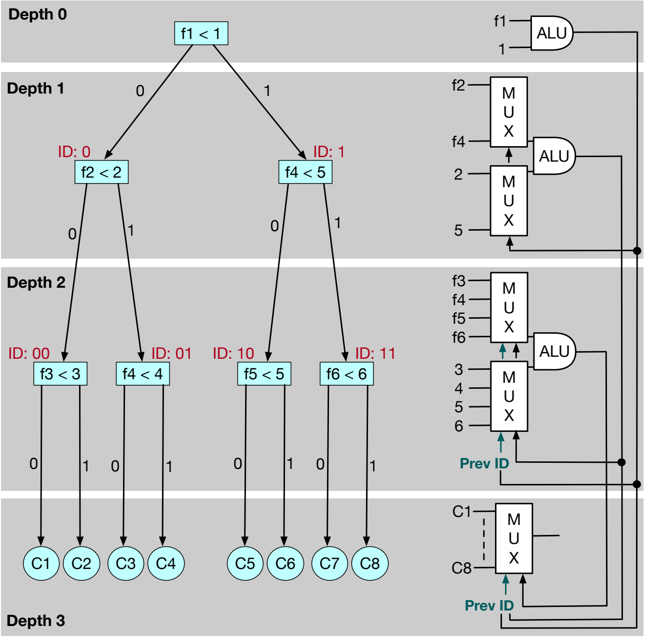
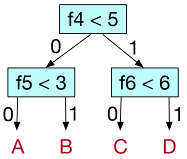


Figure 7: Mapping a decision tree to the switch pipeline using only one multiplexed ALU per tree depth.

the ALU operations in PISA pipelines cannot be changed at runtime. Instead, Leo transforms a more general decision tree into an equivalent version with only $f < c$ conditions offline (Figure 6). Specifically, (i) An $x \leq C$ condition is converted into $x < C+1$ while an $x \geq C$ is converted into $x > C-1$. (ii) $x > C$ is transformed into $x < C$ by simply swapping the left and right sub-trees of the node. Both these transformations maintain the behavior of the original conditions.

4.4 Sub-Tree Flattening and Multiplexing

A naive mapping of a representative decision tree structure to the switch pipeline would require a multiplexed ALU for each tree node implementing the conditional decision at that node. This can be as high as $2^D - 1$ ALUs to support a complete representative decision tree of depth D. However, we observe that at runtime a given packet would access *exactly one* tree node at a given tree depth. The tree node that the packet accesses at depth d would be decided by the path the packet takes through the tree, which, in turn, can be decided by the combination of the **{ID, decision output}** of the decision tree node accessed at depth $d-1$. For example, in Figure 7, if the ID of the node accessed at depth 1 was 0 and the decision output of that node was 1, then the node that will be accessed at depth 2 will have ID 01. To implement this (Figure 7), Leo only needs to provision one multiplexed ALU per tree depth d. The combination of the node output and node ID at depth $d-1$ (shown as **Prev ID** in Figure 7) is used as the select value for the MUX at depth d to configure the ALU with the correct decision tree node parameters (feature and constraint). Hence, for a D depth tree, only D ALUs are needed.



O1 = f4 < 5 **O2** = f5 < 3 **O3** = f6 < 6

O1	O2	O3	ID
0	0	0	A
0	0	1	A
0	1	0	B
0	1	1	B
1	0	0	C
1	0	1	D
1	1	0	C
1	1	1	D

Boolean SRAM Table

O1	O2	O3	ID
0	0	*	A
0	1	*	B
1	*	0	C
1	*	1	D

Boolean TCAM Table

Figure 8: Encoding a decision sub-tree using a boolean table. The boolean table could be stored either in SRAM or TCAM.

While the above design requires significantly fewer ALUs compared to the naive design, unfortunately, it does not scale to trees with larger depths. This is because we require one ALU for each tree depth, and the ALU at a given depth could be configured only once we have the outputs of the ALUs from the lower tree depths. This inherent dependency means that ALU at each depth must be mapped to a different switch pipeline stage, thus requiring $D+1$ pipeline stages to implement a tree of depth D (Figure 7). In programmable switches, the number of pipeline stages can be as few as $\sim$10. To overcome this bottleneck, Leo uses two key insights.

First, Leo generalizes the unit of mapping from a single decision tree node as above to a sub-tree of size k nodes. Further, Leo notes that a sub-tree of size k within a decision tree can be represented using a **boolean table** by encoding all possible combinations of the outputs of the decision nodes within the sub-tree. This is illustrated in Figure 8. Hence, assuming sufficient resources per switch pipeline stage, one could implement an entire sub-tree of size k by calculating the output of each decision node in the sub-tree in parallel (using k multiplexed ALUs), and next matching those outputs against the boolean table(s) to figure the output of the entire decision sub-tree. We call this **sub-tree flattening**.

Figure 9 illustrates the idea where the representative tree is partitioned into layers, each layer comprised of similar sub-trees. Using similar insights as earlier, we observe that for any layer, exactly one sub-tree is executed at runtime as decided by the path the packet takes through the tree. For example, in Figure 9, exactly one of the four sub-trees (of size $k=3$) in layer 2 will be executed. Leo generalizes the idea of node multiplexing to **sub-tree multiplexing**, by provisioning for exactly one sub-tree per layer. Given a sub-tree multiplexing unit of size k nodes (which is always a complete binary tree

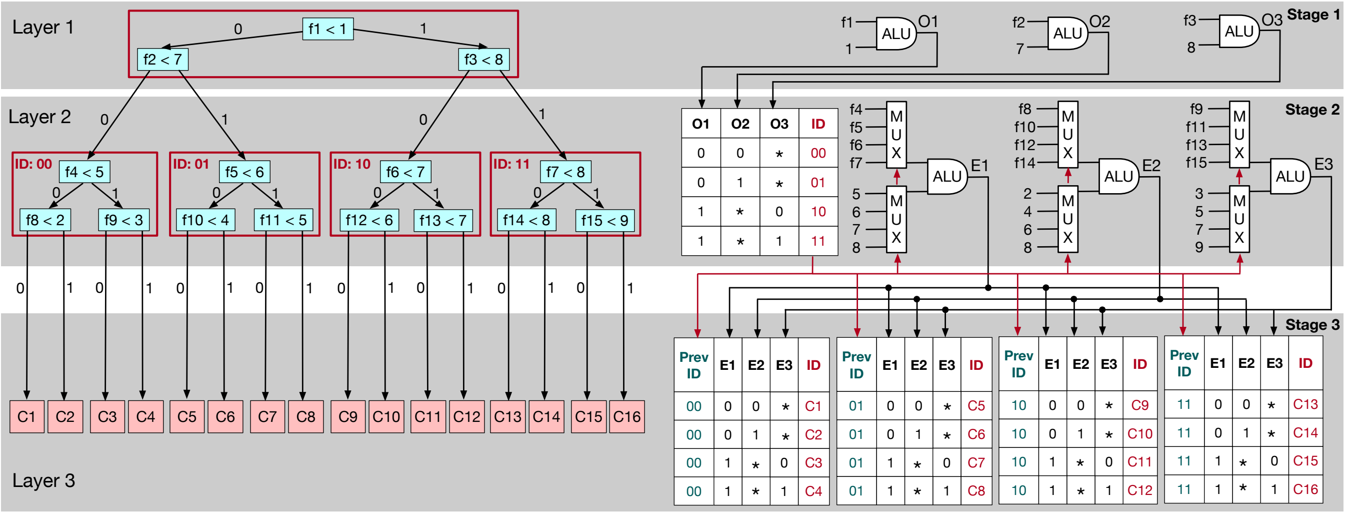


Figure 9: Illustrating sub-tree flattening and multiplexing (with sub-tree size $k = 3$) for a representative (complete) decision tree of depth $D = 4$ from Figure 4. As mentioned in §4.4, it requires $\lceil D / \lceil log_2(k) \rceil \rceil + 1 = 3$ switch pipeline stages to map the tree.

in Leo) and n such blocks of sub-trees at layer i, Leo only provisions for one sub-tree of size k (i.e., k multiplexed ALUs). In Figure 9, there are $n = 4$ blocks of sub-trees of size $k = 3$ in layer $i = 2$. Hence, Leo only provisions $k = 3$ multiplexed ALUs for $i = 2$. While the figure shows all layers having $k = 3$, Leo allows for different layers to be associated with different sized sub-trees (and hence k may vary across layers).

Finally, similar to Figure 7, to decide which sub-tree (or leaf node) at layer i to choose, Leo requires the id and the outputs of the sub-tree at layer $i-1$. The combination of the **{ID, decision outputs}** from layer $i-1$ is fed to the feature and constraint multiplexers at layer i to choose the right feature and constraint value for each multiplexed ALU. Note that in Figure 9, we don't need the ID of the node at layer 1 to configure the multiplexers at layer 2; only the decision outputs of the sub-tree at layer 1 (shown as **O1, O2, O3** in the figure) are needed. This is because there is only one sub-tree at layer 1. However, at layer 2, we have 4 sub-trees, and hence in Figure 9, the leaf node at layer 3 is selected using both the ID of the sub-tree accessed at layer 2 (shown as **Prev ID** in the figure) plus the decision outputs from that sub-tree.

Using complete sub-trees of size $k > 1$ nodes as the unit of flattening in every layer, Leo only requires $\lceil D / \lceil log_2(k) \rceil \rceil + 1$ pipeline stages to implement a complete decision tree of depth D, as illustrated in Figure 9.

4.5 Runtime Programmability

The inputs to the building blocks of a representative decision tree in Leo, namely the feature and constraint multiplexers and the boolean table, can be configured and updated at runtime via the switch control plane without any switch downtime. Thus, by changing the inputs to these building blocks appropriately at runtime, one could implement any decision tree within a given (D, L, F) class. (Figure 16 in the Appendix presents an example of a different tree mapped to the same representative structure in Figure 9).

Handling transient state during runtime tree updates. One key issue with using the control plane for runtime tree updates is that it can take several clock cycles to update all the tree nodes while transitioning from one decision tree to another. Hence, packets arriving during the update might encounter an inconsistent tree state. To handle this, Leo maintains two copies of the representative decision tree. At any given time, exactly one of those trees is marked as active, and all incoming packets are directed through the active tree. In order to transition to a new decision tree, the control plane configures the nodes of the inactive decision tree, and once the new tree is configured, it is marked as active, while the previously active tree is marked inactive. Switching the status of a tree from active to inactive and vice-versa is an atomic operation, as it only requires writing to a single register entry storing the tree id of the active tree. Thus, future incoming packets smoothly transition to the new decision tree without encountering any inconsistent state. The approach doubles the switch resources required to support a representative decision tree, but we show in §7 that the costs are acceptable.

5 Leo Implementation

We present details of our implementation in PISA ASICs.

Implementing Multiplexed ALUs. Figure 9 shows the feature and constraint multiplexers, and the Boolean SRAM/TCAM table as distinct components. We however implemented all of them using the same Match/Action table (MAT), with the comparison performed using the stateless ALU in the action field. For concreteness, layer 2 of Figure 9 is implemented as 3 MATs, one per Multiplexed ALU, which respectively produce the results $E1$, $E2$, and $E3$. The MAT which outputs $E1$ has $F+2$ possible actions. F of the actions (one per feature) involve (i) executing a condition of the form $E1 = f < c$, where f is an appropriate feature in the packet's header or metadata, and c is the relevant constraint loaded in the table memory as an action parameter; and (ii) assigning the ID field for the next layer based on another action parameter. The other two actions include a (i) `NoOp,` which simply disables the multiplexed ALU when a tree of smaller depth is installed; and (ii) `SetLeaf,` which terminates a path in the tree by writing the class label to the packet header. The MATs that compute $E2$ and $E3$ are similar, except that they do not compute the ID field (it suffices one MAT in each layer computes the ID field for the next layer).

Ideally, the operation $E1 = f < c$ is realizable in the stateless ALU. However, we were unable to compare a memory-read parameter (i.e. the constraint) with a header/metadata (i.e. a feature) in a single processing stage. We instead implemented the comparison as $E1 = f - c$ in a single stage. The condition is met if E1 is negative which involves checking that the Most Significant Bit (MSB) of $E1$ is 1. We achieve this with TCAMs using wildcard matching (of the form $1***$) for $E1$ in the next layer Boolean Table, realizing each layer in a single switch stage. With SRAMs, we used an extra stage to extract the MSB of E1, needing two stages per layer.

Choice of k. While larger k can reduce the number of stages, we typically used $k = 3$. Increasing k beyond 3 is constrained by the limit that PISA switches place on the number of header fields within each container group. Two header fields can be used together in an ALU operation only if they are in the same group (e.g., in Figure 9, the different features, and the outputs O and E of each layer are part of the same group). We reuse output headers with $k = 1$. When $k > 1$, neighboring layers must use different output headers as multiple MATs in a layer rely on the output headers of the previous layer. Hence, we reuse across alternate layers (odd layers use inputs E and outputs O, with the opposite for even layers), and our implementation requires |F|+2k header fields where |F| is the number of features. An increase in k reduces the number of features possible. A potential trade-off that we defer for future investigation is to implement each layer in two stages: (i) a first that determines an ID; and (ii) a second that uses the ID for the multiplexed ALUs. This would reduce the number of header fields in each container group to $|F| + k$ and allow more features, at the expense of more stages per layer.

Handling stateful features. Stateful features requires allocating per-flow memory. We augment the `Leo` abstraction to include a target on the number of flows M that are to be supported. `Leo` maps stateful features to 8-bit or 16-bit registers based on the feature. Features may be discretized to fit the desired budget – e.g., packet lengths are divided by 64 (using bit shifts) which allows lengths ranging from 64 bytes to 16 KB to be represented in 64 byte units using a 8-bit budget. We discuss how other features are discretized in §A.3.3).

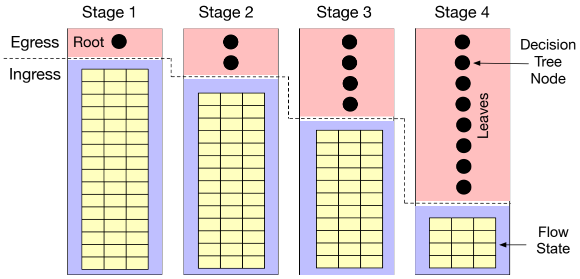


Figure 10: Illustration of how `Leo` maximally utilizes the resources in each physical stage by mapping the feature tables to the ingress pipeline and decision tree to the egress pipeline.

We store stateful features in separate SRAM register tables indexed by flow ids. The flow id is calculated as a hash over its 5-tuple. In PISA switches, each packet first traverses through an ingress pipeline followed by an egress pipeline. Each physical pipeline stage serves as both an ingress and an egress stage, with its resources shared between the ingress and egress. In `Leo`, we map the tables containing flow state to the ingress pipeline and the decision tree to the egress pipeline (Figure 10) to maximally utilize pipeline resources. This is motivated by the fact that when mapping the decision tree, many stages consume minimal memory resource (especially the first few stages corresponding to the top tree levels with fewer internal nodes). Thus `Leo` could use the remaining resources in those physical stages to store flow state in the ingress. Alternatively, if we had mapped both the flow state and the decision tree to the ingress pipeline, then the inherent dependency that the flow state must be accessed before the decision tree logic would mean that `Leo` could not use the resources left in each stage after mapping the decision tree nodes. This optimization is useful in scaling `Leo` to a large number of flows with stateful features (§7).

6 `Leo` Analysis

We next discuss how to provision a representative tree in `Leo` that can support all trees in the (D, L, F) class. This requires getting an upper bound on the requirements for every resource across *all possible trees* in that class.

The number of rules in each `Leo` table depends on the number of tree states that are multiplexed. For example, in Figure 9, each Mux in layer 2 (resp. layer 3) would multiplex across 4 (resp. 16) possible states. Let k_i denote the subtree multiplexing unit size in layer i (generalizing the above discussion which assumed $k_i = k$ for all layers). Then, each execution layer produces $k_i + 1$ possible outcomes for each of the states multiplexed by the layer. Thus, the maximum number of states multiplexed in layer i (which we denote by R_i) is at most $R_{i-1} * (k_i + 1)$. Given a (D,L,F) specification, the total number of leaves, and hence the number of internal nodes at each tree level does not exceed L. Thus, the number of states multiplexed by any layer is at most L, reducing the entries to be provisioned. We now have:

$$R_i = \begin{cases} 1 & \text{if } i \leq 1 \\ min(L, R_{i-1} * (k_i + 1)) & \text{otherwise} \end{cases} \quad (2)$$

If TCAM is used, the number of entries needed in each mux in layer i is simply R_i, the number of multiplexed states. If an SRAM is used, the number of entries is $R_{i-1} * 2^{k_{i-1}}$ as it depends on the multiplexed states in the previous layer, and the possible combinations of the k_{i-1} outputs of layer $i-1$. Since k values are typically small, this is just a small constant factor of the TCAM rules.

Total memory requirement. When $k_i = k$ in each layer, and we provision for complete trees, the total entries to provision for all tables across all layers (there are k tables per layer) is $(k+1)I+1$ with TCAM and $2^k I+1$ with SRAM. Here, I is the number of internal nodes = $2^D - 1$. When $k = 3$ as in Figure 9, the total entries are $4I+3$ with TCAM and $8I+3$ with SRAM. In contrast, Equation (1) shows for IIsy, the SRAM entries goes as $(I/N+1)^N$, where N is the number of features, while TCAM entries are also exponential in N when I is polynomial. Further, when given a limit L on the number of leaves, and for $k = 3$, a comfortable upper bound in memory requirements is $1+DL/2$ with TCAM, and $1+4DL$ with SRAM, with the more general expression being $1+\frac{DL}{log(K+1)}$ with TCAM, and $1+\frac{2^k DL}{log(k+1)}$ with SRAM.

7 Evaluation

Our evaluations address several questions: (i) What classes of decision trees can be supported by Leo, and how does this compare to other approaches in terms of tree size, the number of stages and rules? (§7.1). (ii) What classification accuracies are achievable on real traffic data-sets using decision trees that Leo can support? (§7.2). (iii) How does the decision tree accuracy vary with the number of flows and stateful features? (§7.3). (iii) What are the benefits of implementing decision trees in the data plane? (§7.4).

To answer these questions, we implement Leo, and other schemes using P4 [3], and analyze resource requirements on a Tofino switch. We also evaluate Leo with a control plane solution in a real tested involving a Tofino switch. We evaluate accuracies on publicly available intrusion detection datasets.

7.1 Leo vs. Other Data plane Tree Schemes

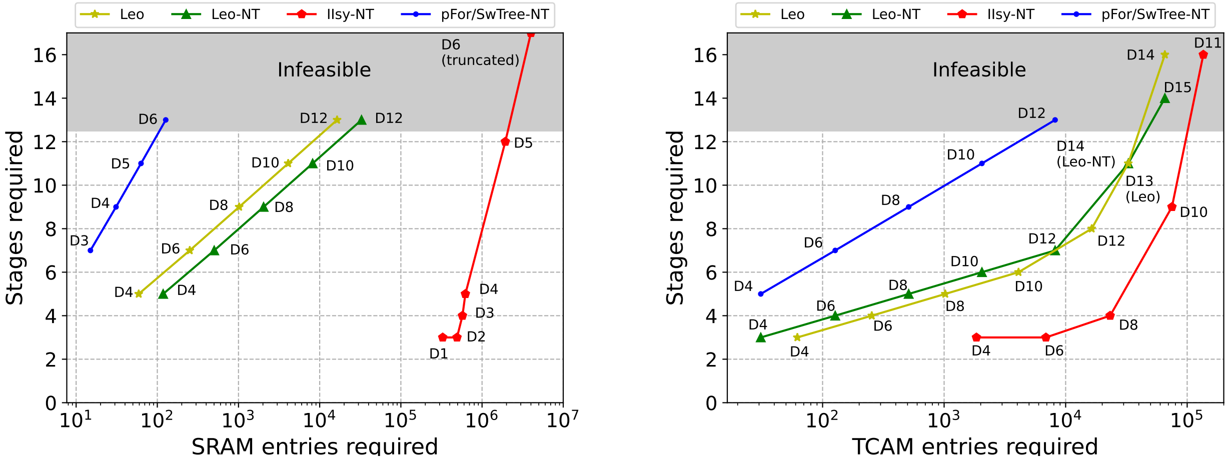
Schemes compared. We compare Leo with tree-based approaches [5, 12, 22] and IIsy [24, 26]. Unfortunately, as discussed in §3, existing tree-based approaches are either not runtime programmable [22], or have only been implemented on software switches. For instance, we experimented with the publicly available code of SwitchTree [12] (an extension of pForest [5]) and found it does not compile on the TNA [2] hardware switch. Instead, we modify Leo to mimic these approaches. Our implementation contains several optimizations not present in [5, 12] and is an optimistic bound on the performance of these approaches. Unlike Leo, none of the prior works guarantee correct decision tree execution during a transient period when a tree update is in progress. Hence, we refer to them as pFor/SwTree-NT and IIsy-NT. We also implement a variant of Leo where we disable the mechanisms to ensure correct transient performance, which we call Leo-NT.

Methodology and metrics. We compare schemes with respect to their ability to support all trees within a given class (D, L, F) that constrains the maximum depth D, maximum number of leaves L, and feature set F. We vary the class specifications across our experiments. For any given class, and for all schemes, we must estimate the memory to be provisioned at compile time for each table used in the scheme. This is needed since the goal is to support all trees within a class in a runtime programmable manner and since table sizes cannot be dynamically changed. This requires estimating the worst-case requirement across all decision trees within a class for each table with every scheme. For Leo and pFor/SwTree, we obtain these requirements using the analysis in §6. IIsy does not discuss how to assign sizes to its tables. Thus to guarantee programmability within each class, we utilize the analysis that we presented in §3 (and detailed in §A.1 and §A.2).

Our main metrics include (i) whether a given specification can be realized on an actual hardware switch (we focus on the Tofino Native Architecture (TNA) [2] used in Intel's Tofino line of switch ASICs); (ii) when the specification can be realized, the total number of SRAM or TCAM entries and the number of stages that the scheme requires. We implement all schemes using P4$_{16}$ targeting TNA and consider a specification met on successful compilation. All P4 code was compiled using version 9.11.1 of the Intel Barefoot SDK.

We focus on the resource requirements of the decision tree alone, and explore per-flow state requirements in §7.3. For Leo and pFor/SwTree-NT, the number of features is limited by the number of headers within each container group (§5). pFor/SwTree-NT can support a maximum of 15 features, while Leo can support upto 10 features for $k = 3$, and 15 features for $k = 1$. With Leo and pFor/SwTree-NT, the memory requirements for the decision tree logic does not grow with the number of features (§6). In contrast, the memory requirements of IIsy-NT grows exponentially with the number of features (§3). In the rest of this section, we compare resource requirements by keeping the number of features fixed with all schemes. Appendix §A.3.1 analyzes the maximum tree depth that can be supported as the number of features change. Finally, our classification accuracy experiments (§7.2) explore different combinations of features and depth for each scheme, and report the best operating point for that scheme.

Results with SRAM. Figure 11(a) presents results for classes of the form $(D, 2^D, F)$, which indicates that all trees of depth D or lower must be supported. We varied D while keeping the number of features fixed at 10. All schemes were constrained to only using SRAM. First, pFor/SwTree-NT can only meet



(a)

(b)

Figure 11: Table entries required to support a programmable class of complete decision trees using SRAM and TCAM. Designs needing more than 12 stages are infeasible on a Tofino switch [7]. Note the logarithm scale on x-axis.

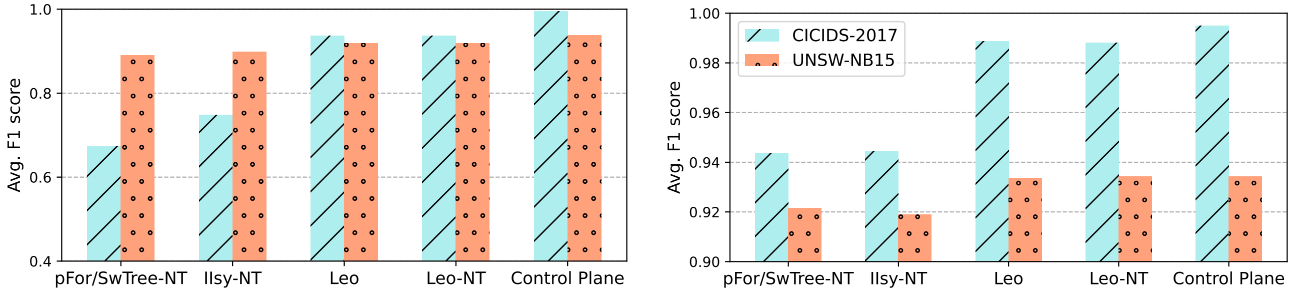
the specification upto $D = 5$, and it is unable to support depth 6 trees. This is because the approach is constrained by the number of switch stages. Second, Leo is able to support trees twice as deep as those supported by pFor/SwTree-NT, and can support all depth 10 trees. Further it requires half as many stages for the same depth. While the total memory requirement is higher, it is modest in an absolute sense, and this is an acceptable trade-off. Third, the memory requirements with IIsy-NT [24, 26] are several orders of magnitude more than Leo. Supporting all depth 5 trees requires over 1.9M entries, which is 10000× more than Leo (which requires 182 entries). IIsy is unable to support trees greater than depth 5 owing to the memory requirement. Finally, although Leo uses twice as much SRAM than Leo-NT, the table sizes are still small, and there is no impact on the classes of trees supported.

Results with TCAM. Figure 11(b) presents similar results as above but now implementing all tables in all schemes with TCAM. First, all schemes can support larger trees with TCAM, but Leo still outperforms – pFor/SwTree and IIsy can support depth 10 trees, but Leo can support any depth 13 tree. Second, for the same specification, Leo requires half as many stages as pFor/SwTree, and an order of magnitude fewer TCAM entries (for depth 10, the requirement with IIsy is 18× that of Leo). Third, the higher memory requirements of Leo limit it to depth 13 trees (versus depth 14 for Leo-NT), owing to both lower TCAM memory available per stage, and the larger memory requirements at higher tree depth. However, Leo can still support larger trees with a bound on the number of leaves with negligible impact on classification accuracy as we will see later. Finally, Leo and pFor/SwTree-NT require fewer stages for the same depth with TCAM relative to SRAM. This is because of our optimizations with TCAMs to avoid an extra computation stage for each layer (§5)

Exploiting bounds on the number of leaves. While the earlier results indicate Leo (resp. Leo-NT) can support all complete trees upto depth 13 (resp. 14), we next explore specifications that bound both the number of leaves and tree depth. This is beneficial given our analysis in Figure 1 which shows that accuracy greatly improves with tree depth, but is less sensitive to the number of leaves for a given depth. Figure 13 shows the maximum number of leaves that can be supported for different tree depths, and indicates that Leo (and Leo-NT) can support deeper trees given a constraint on the number of leaves. For instance, Leo can support depth 22 trees given a specification that the tree has at most 1024 leaves This is because Leo provisions fewer TCAM entries in each layer exploiting knowledge of the leaf constraint. The maximum number of leaves reduces with tree depth because for larger depths there are fewer available stages (and hence TCAM table space). In contrast, IIsy, and pFor/SwTree can support at most depth 10 trees as discussed earlier. While Leo allows fewer leaves than Leo-NT (see Appendix for more discussion), the number of leaves is still high and there is practically no impact on classification accuracy (§7.2).

7.2 Classification Accuracy on Real Datasets

Intrusion detection datasets. We used two publicly available and widely used intrusion detection datasets: **(i) CICIDS-2017 (CICIDS):** The dataset [17] consists of multiple attack classes along with a single benign class. Since some attack classes have very little data, we merge the 9 least populated attack classes into a single class resulting in a total of 7 classes. The dataset consists of 78 flow and packet level features. However, not all of these can be deployed on the hardware (e.g. means and percentiles that require division, etc.). Thus, we



(a)

(b)

Figure 12: Classification accuracy of the best performing, hardware-supported tree class for each scheme when implemented using (a) SRAM and (b) TCAM. We assume all switch stages are available for the decision tree logic, and do not account for memory requirements of stateful flow features for all schemes (we explore impact of flow state in 7.3.) The control plane implementation is independent of SRAM and TCAM, and hence the accuracy is the same across figures (a) and (b) – Note the y-axis has different scale in figures (a) and (b).

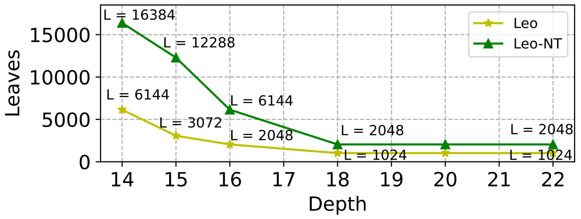


Figure 13: `Leo` can support tree classes beyond depth 13 with a bound on the number of leaves.

only select 42 switch-compatible features for our evaluation; and **(ii) UNSW-NB15(`NB15`):** This dataset [13] also consists of real user traffic interspersed with generated attacks. We use this dataset to classify traffic as malicious or benign. It contains 49 flow and packet level features, out of which we use 22 switch-compatible features.

Training. We use Python3's scikitlearn library to train the decision trees. The CART algorithm was used along with the Shannon entropy loss function. We enabled class weighting to alleviate issues due to imbalance. To find the best set of features for an experiment, we use the Mean Decrease in Impurity (MDI) score and eliminate one least-scored feature recursively until we arrive at the target number of features for an experiment. We experimented with Permutation Importance score and found no difference in accuracy compared to MDI. 75% of the dataset was used for training, and 25% for testing. All results are reported on the test set.

Metrics. We report the average of the per class F1 scores which ensures the accuracy for all classes including those with relatively low samples is considered. Given the highly imbalanced nature of the dataset (most samples are benign), alternative metrics that weigh by the number of samples provide a highly optimistic view of accuracy for all schemes. For each scheme, we evaluated a range of (D, L, F) tree classes that are supported on the switch, we then selected the best performing class for each scheme and present them in Figure 12. Our comparisons include an idealized scheme (Control Plane) which gives an upper bound on the accuracy of the decision tree model for each dataset (by allowing large trees that use all dataset features).

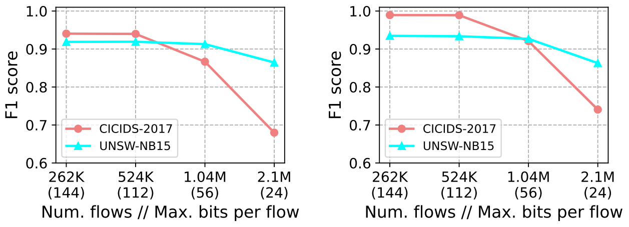
Results. For `CICIDS` and SRAM, `Leo` outperforms both `IIsy-NT` and `pFor/SwTree-NT` achieving an average F1 score of 0.94 which is much higher than the 0.75 of `IIsy-NT` and 0.67 of `pFor/SwTree-NT`. With TCAM, `IIsy-NT` and `pFor/SwTree-NT` support depth 10 trees and thus improve to a 0.94 F1 score, while `Leo` outperforms both by managing a 0.98, due to its ability to support a depth 22 tree.

For the `NB15` dataset, using SRAM `Leo` outperforms both `pFor/SwTree-NT` and `IIsy-NT` by achieving an average F1 score of 0.92(versus 0.88). Using TCAM, `Leo` continues to outperform the others while managing to achieve identical accuracy as the control plane. Notice, that all schemes perform better with the `NB15` dataset because of the smaller number of classes. The Appendix presents a breakdown of the F1 score per class for all schemes. We also show results for the `NB15` dataset with multiple classes. Our results show similar trends – `Leo` outperforms `IIsy-NT` and `pFor/SwTree-NT` for each class while performing close to the control plane solution.

7.3 Number of concurrent flows supported

Stateful per-flow features must fit into the memory budget of the switch (§5). Supporting more flows reduces the bits per flow (and which features can be supported), thereby impacting classification accuracy. We explore these issues next.

Methodology and metrics. We experimentally determine the maximum switch memory that can be configured as registers. Given a target M on the number of flows, we determine the budget for per-flow state, and then find different config-



(a)

(b)

Figure 14: Impact of per-flow state on Leo classification accuracy using (a) SRAM and (b) TCAM.

urations of stateful features that can meet the budget. For example, a 24-bit budget can be configured as (16-bit * 1 + 8-bit * 1) OR (16-bit * 0 + 8-bit * 3). For each valid configuration, we rank features by their MDI feature importance, and greedily select the top features that fit the configuration. We compute the F1 score achieved for each valid configuration and report the F1 score of the best performing configuration.

Results. Figure 14(a) and 14(b) summarize how classification accuracy varies with the number of flows for SRAM and TCAM respectively. The X-Axis corresponds to the number of flows to be supported, and the corresponding budget on per-flow state (e.g., supporting 1.04 M flows imposes a budget of 56 bits per flow). The leftmost points match accuracy levels shown in Figures 12(a) and 12(b). Leo can support 1.04M concurrent flows with modest degradation, with average F1 score decreasing from 0.94 to 0.86 for SRAM, and decreasing from 0.98 to 0.92 for TCAM and CICIDS. For NB15, F1 scores decrease from 0.92 to 0.91 for SRAM, and are practically unchanged for TCAM. This is because we use the dataset for binary classification of traffic as benign or malicious, an easier classification problem. Table 2 in the Appendix summarizes the features used when supporting 1.04M flows for all 4 configurations.

7.4 Leo vs. Control Plane

To evaluate the benefits of implementing decision trees in the data plane versus control plane, we deploy Leo on an Intel Tofino [2] EdgeCore Wedge-100BF-32x [15] switch. We train decision trees of varying depths using the CICIDS dataset allowing the control plane to utilize the complete set of 78 features while restricting Leo to 10. We use tcpreplay to replay packets from select flows from the dataset to ensure similar characteristics of the original traffic are maintained. Besides classification, the switch also implements L2 forwarding. We capture timestamps at the switch ingress and egress to accurately measure the processing delay.

The control plane experiment involves forwarding the packet to the switch CPU for inference with no other functionality in the data plane (besides IPv4 match-based forwarding) while the data plane experiment uses Leo.

We find (see Figure 23 in the Appendix) that Leo is on average 500× faster than the control plane implementation of the same decision tree class. Leo takes about 500 nanoseconds to completely apply a 1024-leaf tree for inference to every packet. This re-asserts the point that a control plane inference scheme is not capable of per-packet classification at multi-terabit line rates. On the other hand, Leo can not only do per-packet classification, but also achieve accuracy comparable to an idealized control plane scheme for real datasets (§7.2).

8 Related Work

Beyond [5, 12, 22, 26]. N3IC [18] implements a binary neural network in the NICs with the goal of accelerating traffic analysis at the edge rather than per-packet in-network traffic analysis. Taurus [20] extends PISA pipelines with a new hardware module implementing a map-reduce abstraction to run neural networks. Unlike Taurus, Leo focuses on efficiently mapping ML inference to existing PISA pipelines, thus enabling data plane ML inference on commercially available hardware. Given training data, a recent work, Homonculus [21], trains an ML model that achieves a given accuracy while meeting the constraints of a switch. Neither Taurus nor Homonculus support runtime programmability. In contrast, our focus is on runtime programmability, and provisioning sufficient resources so *all models in a given class* can be supported. Recent work [23] explores support for runtime programmability for a general class of packet processing programs. In contrast, we only focus on providing runtime programmability for ML inference programs, without hardware changes.

ATP [11] and SwitchML [16] do in-network aggregation in the data plane to accelerate ML training. In contrast, Leo focuses only on ML inference in the data plane, and assumes that ML training is done offline using standard techniques. Finally, there have also been recent works that use a hybrid of control and data plane for traffic analysis. FastFE [1] and DAD [14] both implement ML inference in the control plane and extract feature values in the data plane. As shown in §7.4, such an approach cannot do per-packet traffic analysis.

9 Conclusion

We have presented Leo, a system for online decision tree classification in the data plane. Given a specification of a class of trees. Leo supports any tree in that class at runtime. Leo reduces resource requirements for a given tree through sub-tree multiplexing, and supports decision trees to be changed at runtime by allowing features and constraints to be re-programmed. Leo support trees with 2× the depth of prior data plane solutions, resulting in much better classification accuracies comparable to a control plane solution. Evaluations on a real programmable switch testbed show classification latencies with Leo are 500× lower than a control plane approach. Overall, the results show Leo is a viable approach to support packet classification in the data plane.

References

[1] Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, and Hongxin Hu. FastFE: Accelerating ML-Based Traffic Analysis with Programmable Switches. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure (SPIN)*, 2020.

[2] Intel Barefoot Networks. P4-16 Intel Tofino Native Architecture. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed: 05/04/2023.

[3] 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. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[4] 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. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[5] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pForest: In-Network Inference with Random Forests. https://arxiv.org/abs/1909.05680, 2019.

[6] The P4 Language Consortium. P4-16 Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.2.2.pdf. Accessed: 05/04/2023.

[7] Vladimir Gurevich and Andy Fingerhut. P4-16 Programming for Intel® Tofino™ using Intel P4 Studio™. Open Network Foundation. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf. Accessed: 05/04/2023.

[8] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Patrick Kalmbach, Johannes Zerwas, Péter Babarczi, Andreas Blenk, Wolfgang Kellerer, and Stefan Schmid. Empowering Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 8–14, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Wolfgang Kellerer, Patrick Kalmbach, Andreas Blenk, Arsany Basta, Martin Reisslein, and Stefan Schmid. Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. *Proceedings of the IEEE*, 107(4):711–731, 2019.

[11] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[12] Jong-Hyouk Lee and Kamal Singh. SwitchTree: In-network Computing and Traffic Analyses with Random Forests. *Neural Computing and Applications*, 11 2020.

[13] Nour Moustafa and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2015.

[14] Francesco Musumeci, Valentina Ionata, Francesco Paolucci, Filippo Cugini, and Massimo Tornatore. Machine-learning-assisted DDoS attack detection with P4 language. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020.

[15] EdgeCore Networks. EdgeCore Wedge 100BF-32X. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335. Accessed: 05/04/2023.

[16] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[17] Iman Sharafaldin, Arash Habibi Lashkari, and Ali Ghorbani. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. pages 108–116, 01 2018.

[18] 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 *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[19] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices

in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2019.

[20] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. Taurus: A Data Plane Architecture for per-Packet ML. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1099–1114, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 329–342, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Bruno Missi Xavier, Rafael Silva Guimarães, Giovanni Comarela, and Magnos Martinello. Programmable Switches for in-Networking Classification. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[23] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. Runtime Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 651–665, Renton, WA, April 2022. USENIX Association.

[24] Zhaoqi Xiong and Noa Zilberman. Do Switches Dream of Machine Learning? Toward In-Network Classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 25–33, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.

[26] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. IIsy: Practical In-Network Classification. https://arxiv.org/abs/2205.08243, 2022.

A Appendix

A.1 `IIsy` with SRAM.

We present a detailed explanation of Equation 1 elaborating on how default rules are handled. We obtain conservative lower bounds on the size of feature tables and combination tables. To obtain a lower bound on requirements, it suffices to construct a single tree in the class of trees that are to be supported that needs a given amount of memory. Since the combination table, and each of the feature tables are provisioned independently, we present below separate example trees that trigger a minimum memory requirement for each of the tables.

Feature Table. Consider a decision tree of depth D based on only one of the N features, which could take values in the range $[0,K]$. For an SRAM, `IIsy` must explicitly enumerate every value, and map the value to a code word which requires a table of size K. Some savings could be obtained with a default rule since the values in the largest interval need not be explicitly enumerated. A complete tree with depth D has $I = 2^D - 1$ internal nodes, and partitions the possible feature values into $I+1$ intervals. Consider a tree where all intervals have the same length $\lceil \frac{K}{I+1} \rceil$. The number of SRAM rules required in the feature table for this tree is $K - \lceil \frac{K}{I+1} \rceil$ indicating at least so much memory is required.

Combination Table. To derive conservative bounds on the size of the combination table, consider a complete decision tree of depth D where each feature appears in the same number of decision tree nodes. Let $I = 2^D - 1$ denote the number of internal nodes. The total number of decision nodes that involve each feature is $\frac{I}{N}$, requiring $\frac{I}{N} + 1$ codewords per feature. Since the combination table includes combinations of all possible codewords associated with each feature, the total size is $(\frac{I}{N} + 1)^N$ which is exponential in N. A *default* rule could take away entries corresponding to one leaf. Since there are 2^D leaves in a depth D tree, taking away combinations corresponding to one of the leaves will still require at least a fraction $\frac{2^D-1}{2^D}$ of the remaining combinations. Thus, the total number of entries in the combination table is at least:

$$\frac{2^D - 1}{2^D} * (\frac{2^D - 1}{N} + 1)^N \qquad (3)$$

Equation 1 follows by combining the two terms.

A.2 `IIsy` with TCAM

A.2.1 Proof sketch of Proposition 2

We complete the discussion of the proof sketch for the general case. The proof is based on a family of decision trees shown in Figure 15 for the general case with N features $(F_1 \dots F_N)$ with each feature having values ranging from $1 \dots K$.

As discussed earlier, the intuition behind the tree construction is as follows. First, the decision tree nodes has leaves for each value of a feature F_j when all other features are at their

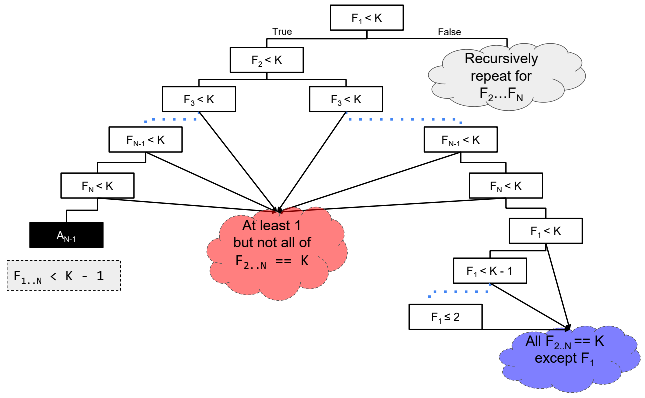


Figure 15: A decision tree that requires exponential TCAM entries with `IIsy`.

maximum value K. This forces `IIsy` to use a distinct code word for each value of every feature. Next, the decision tree has leaf nodes which correspond to regions where some of the features can take multiple possible values. These nodes will require a large number of code word combinations with `IIsy` since it is forced to use a distinct code word for each feature value.

In more detail, consider the left sub-tree of the root (feature $F_1 < K$). We have three cases based on other features:

(i) All other features are K (Blue sub-cloud). Here, a different action is chosen depending on the value of F_1, forcing `IIsy` to pick a different codeword for each value of $F1$.

(ii) All other features are $< K$ (left most node A_{N-1}). Although a single decision tree node, `IIsy` is forced to represent this using $(K-1)^N$ distinct combinations because each value of each feature is associated with a different codeword.

(iii) All other nodes which together capture the case that at least one, but not all of $F_2, F_3 \ldots F_N$ are K. An action distinct from the above is chosen for these nodes. Consider one such node where $F_N = K$ and all other features are $< K$. Encoding this alone will require $(K-1)^{N-1}$ combinations.

While a default rule can be used to cover the combinations in either Case (ii) or Case (iii), both cases cannot be covered. Thus, `IIsy` requires at least $(K-1)^{N-1}$ distinct combinations. With a TCAM, this would require at least m^{N-1} rules, where $m = log(K-1)$, which again grows exponentially with N.

It remains to estimate the total number of leaves in the tree. The left sub-tree has $K+2(N-2)$ leaf nodes (the three cases above are associated with $K-1$ leaves, 1 leaf, and $N-2+N-2$ leaves respectively), The right sub-tree expands in a recursive fashion with the same structure on (N-1) features [i.e., the features $F2 \ldots F_N$]. Setting and solving a recurrence, the total number of leaf nodes is $N^2 + N*(K-3)+2$. This leads to the proposition.

A.2.2 Estimating TCAM memory requirements

We present models to obtain conservative lower bounds on the size of feature tables and combination tables when TCAM is used with `IIsy`. To obtain a lower bound on requirements, it suffices to construct a single tree in the class of trees that are to be supported that needs a given amount of memory. Since the combination table, and each of the feature tables are provisioned independently, we present below separate example trees that trigger a minimum memory requirement for each of the tables.

Estimating feature table size. Consider again a decision tree where all nodes correspond to the same feature which takes values in the range $[0,K]$. With a TCAM, the feature table need not explicitly enumerate all possible feature values. For instance, an interval $[1,15]$ can be represented with 4 TCAM entries. In general, any interval requires at most log K entries, and there are $I+1$ intervals, where I is the number of internal tree nodes. While $(I+1)*logK$ serves as an upper bound on the entries needed by the feature table, this is not a conservative lower bound since not all intervals may need $logK$ entries (e.g., the interval $[0,15]$ requires 1 TCAM rule, while $[1,15]$ requires 4). We have devised a simple algorithmic procedure, which partitions the space $[0,K]$ into $I+1$ intervals so the total number of TCAM rules to encode all intervals is maximized. The algorithm works by iteratively partitioning each interval into two such that the total rules across the two intervals is maximized. The iterations proceed until the number of internal nodes matches the desired target or when no further splits are possible. While we do not have a simple closed form expression for this estimate, we denote the total feature table rules required using this procedure as $TCAMFTRules(K,I)$. This is a minimum size `IIsy` needs to provision.

For example, for the numeric range $[0,31]$ using 4 internal decision nodes, the recursive split is as follows:

$$\begin{gathered}[0,31]\\ [0,0],[1,31]\\ [0,0],[1,16],[17,31]\\ [0,0],[1,8],[9,16],[17,31]\\ [0,0],[1,8],[9,16],[17,24],[24,31]\end{gathered}$$

This example would allocate $1+4+4+4+3 = 16$ TCAM rules per feature table.

Combination table rules estimation. We use the number of table rules that `IIsy` requires for the special tree construction (Figure 15) as a conservative lower bound on the size of the combination table. Below, we analyze the rules required for this tree.

Let T_N denote the total number of combinations that `IIsy` must handle in its combination table for this tree. Then, we set up the following recursion. Let $K1$ denote $K-1$. Then,

$$T_N = \begin{cases} \sum_{n=2}^{N}(K1)^n + \sum_{n=2}^{N-1}(K1)^2 + K1 + T_{N-1} & N > 1 \\ K1 & N = 1 \end{cases} \tag{4}$$

The first term models leaves including (i) the leftmost A_{N-1} node where all features are $< K$; and the right child of $F_n < K$ nodes in the pink cloud. These nodes correspond to the case the first n features are $\leq K-1$, and the feature $n+1$ is K. The second term models leaves which are left sub-children of nodes of the form $F_n < K$ in the pink cloud. These correspond to the case $F_1 < K$, $F_2 \ldots F_{n-1} == K$, and $F_n < K$. The third term models leaves where all features except F_1 are K, with one leaf for F_1 taking each of the values in $1 \ldots K-1$.

Note, that this expression represents the total number of combinations i.e. the resource requirement for this example under SRAM. With TCAM, the combinations corresponding to the same leaf are collapsed using wildcards. Thus we rewrite the expression as:

$$T_N = \begin{cases} \sum_{n=2}^{N} \lg^n(K1) + \sum_{n=2}^{N-1} \lg^2(K1) + K1 + T_{N-1} & N > 1 \\ K1 & N = 1 \end{cases} \tag{5}$$

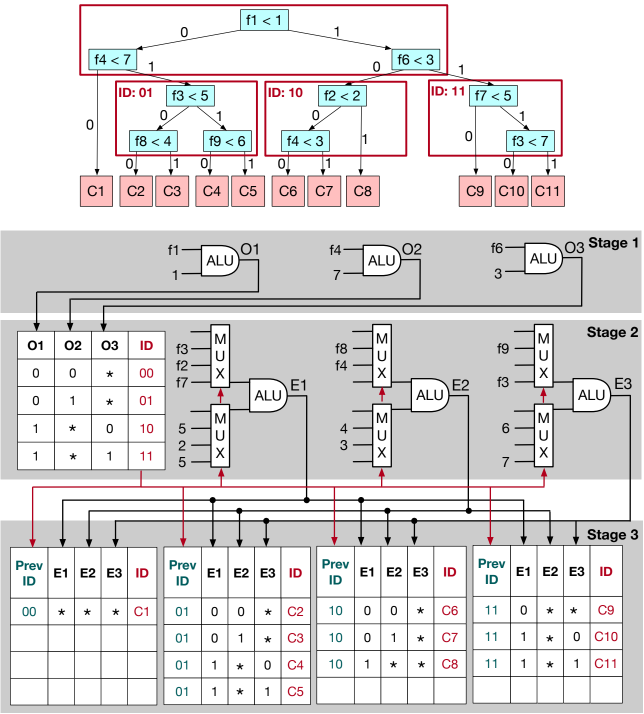
Solving the recurrence gives the total number of rules that must be provisioned.

A.3 Evaluation

A.3.1 Impact of number of features

Figure 11(a) and Figure 11(b) had shown the resource usage for different schemes with 10 features. We next vary the number of features, and show the maximum tree depth that can be supported with different schemes. Results are presented in Figure 17 for both SRAM and TCAM.

We make several points. First, Leo can support up to 10 features for $k = 3$, and 15 features for $k = 1$ (§5). Since the memory requirements for implementing the decision tree logic do not grow with the number of features supported, the choice of k determines the maximum depth. While we have not explored, other optimizations are possible with Leo– e.g., by alternating layers with $k = 1$ and $k = 3$, there is potential to achieve lower depth when the number of features is higher than 10. pFor/SwTree-NT can support up to 15 features, and the maximum depth is the same across features. In contrast, while IIsy-NT is not subject to the constraint on header fields (§5) as it does not perform ALU operations, its memory requirements do grow exponentially with the number of features. Consequently, the tree depth that it can support becomes smaller as the number of features is increased. Leo typically supports higher depth trees than IIsy-NT with SRAM, and comparable or higher depth trees with TCAM up to about 15 features. Beyond 15 features, and if TCAMs were used, IIsy-NT may be viable, while Leo and pFor/SwTree-NT are not – however, in our accuracy experiments, Leo already performs very well with TCAM. Further, per-flow state requirements are the likely bottleneck (§7.3) before hitting 15 features. Finally, IIsy-NT does not handle transients, and the additional memory overheads if transients were addressed may further limit its performance.



O1	O2	O3	ID
0	0	*	00
0	1	*	01
1	*	0	10
1	*	1	11

Prev ID	E1	E2	E3	ID
00	*	*	*	C1

Prev ID	E1	E2	E3	ID
01	0	0	*	C2
01	0	1	*	C3
01	1	*	0	C4
01	1	*	1	C5

Prev ID	E1	E2	E3	ID
10	0	0	*	C6
10	0	1	*	C7
10	1	*	*	C8

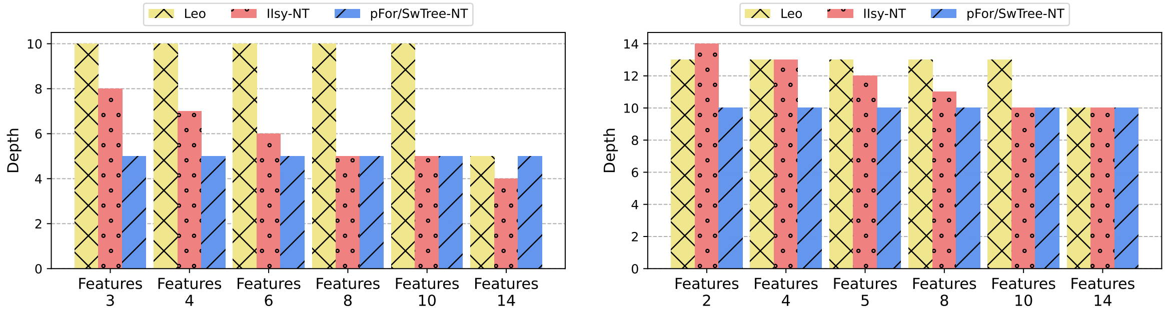
Prev ID	E1	E2	E3	ID
11	0	*	*	C9
11	1	*	0	C10
11	1	*	1	C11

Figure 16: Illustrating the inputs of the mapped representative decision tree from Figure 9 can be configured by the control plane at runtime to implement a different decision tree within the same class.

Figure 18 presents a sweep of classification accuracy with IIsy-NT for different combinations of $(|F|, D)$ for the CICIDS dataset for SRAM and TCAM respectively. Here $|F|$ is the number of features supported, and D is the maximum depth achievable for $|F|$. For SRAM, the best accuracy is obtained with 6 features (depth 6), while with TCAM, the best accuracy is obtained with 14 features (depth 10). These values are reported in Figure 12 in the main text.

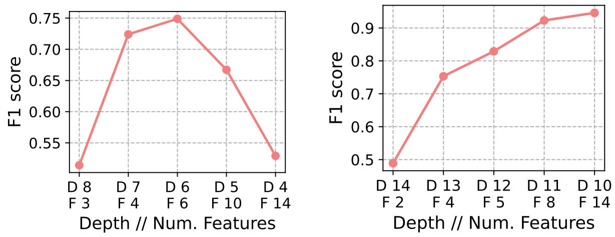
A.3.2 Breakdown of F1 scores per class

CICIDS dataset. Figure 19(a) presents a breakdown of F1 score per class for different schemes for the CICIDS dataset for SRAM. For all classes, Leo outperforms IIsy-NT and pFor/SwTree-NT. While Leo sees an F1 score higher than 0.95 for all classes besides OtherMalicious, IIsy-NT sees F1 scores falling to 0.24 and 0.28 for two classes while pFor/SwTree-NT sees even lower minimum F1 scores. The performance is particularly poor for the OtherMalicious class – this is because the tree depths supported by IIsy-NT and



(a) (b)

Figure 17: Maximum tree depth with different schemes as number of features is varied using (a) SRAM and (b) TCAM.



(a) (b)

Figure 18: F1 accuracies for different settings with IIsy-NT for (a) SRAM; and (b) TCAM for CICIDS. The best result is shown in Figure 12

pFor/SwTree-NT are too small to allow effective categorization of traffic into this class. Leo is close to the Control Plane solution for all but the OtherMalicious class which tends to be challenging because of a mix of different attack patterns – but even here, Leo significantly outperforms other schemes.

Figure 19(b) presents a similar breakdown with TCAM implementations, and Figure 22 presents a zoomed in version. Although all schemes perform much better with TCAM (as the trees supported are larger), Leo still performs better than IIsy-NT and pFor/SwTree-NT, while performing close to Control Plane. The benefits are particularly strong for the OtherMalicious category.

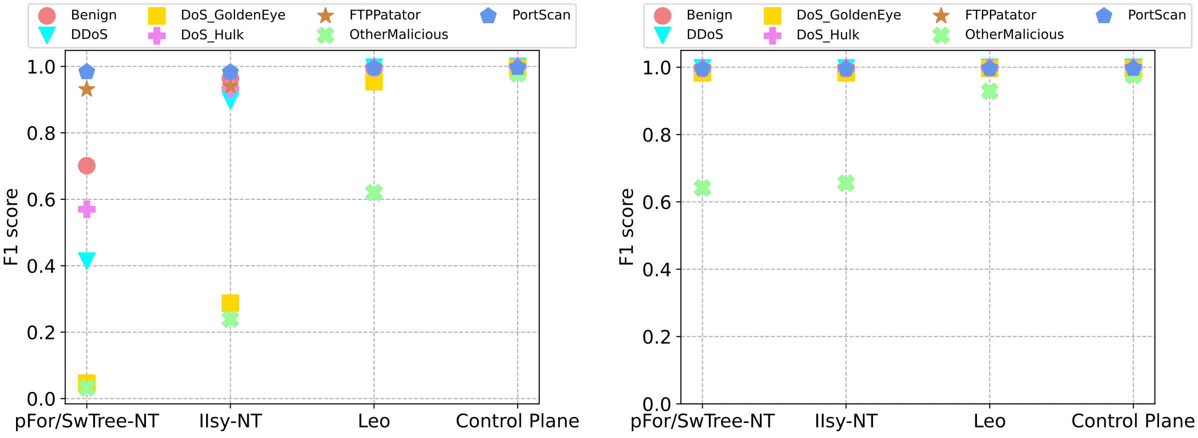
NB15 dataset (binary classification) Figures 20(a) and 20(b) show a similar breakdown for UNSW-NB15 for SRAM and TCAM respectively. All schemes perform better since this experiment focuses on a binary classification (and much better with TCAM since they all achieve larger tree depths) – however, Leo continues to perform better than pFor/SwTree-NT and IIsy-NT, while performing close to Control Plane.

NB15 dataset (multi-class classification) We have also taken the UNSW-NB15 dataset, and trained all approaches for a classification problem involving multiple classes. Figure 21 present a breakdown of the F1 scores per class. Once again, Leo performs better than IIsy-NT and pFor/SwTree-NT (with the benefits stronger in the SRAM case), and performs close to Control Plane.

A.3.3 Other Results/Discussion

Features used in classification. Table 2 presents a set of features used in classification. To implement stateful features, and fit them into 8 and 16 bit budgets. we employed discretization. Packet length metrics were divided by 64 (using bit shifts) which allowed for sizes between 64 bytes and 16K to be represented with 8 bits. Flow size metrics were also divided by 64. Using 16 bits, this lets us represent flow sizes between 64 bytes and 4 MB. Inter Arrival Time is represented as two 8 bit values per flow, one to track the last timestamp, and the other to keep track of the running minimum. The timestamp itself involves extracting an appropriate 8 bits based on the granularity of measurements. Taking the last 8 bits (bits 0 to 7) would capture timestamps up to 255 nanoseconds (ns) in 1 ns granularity. However, extracting a different set of 8 bits allows a different granularity – e.g., extracting bits 2 to 9 would capture timestamps from 4 ns to 1024 ns in 4 ns granularities.

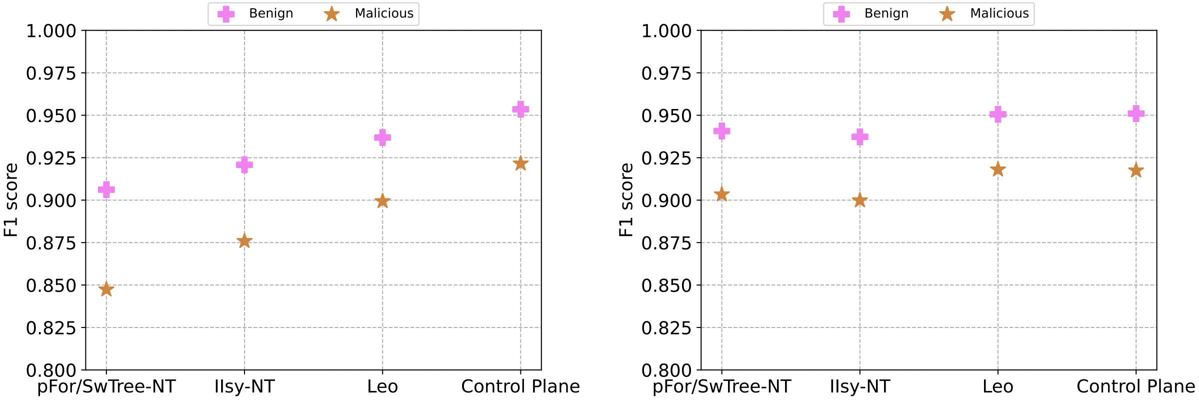
Comparing the number of leaves with Leo and Leo-NT (Figure 13). The impact on the number of leaves is initially more than a factor of 2 before converging to exactly 2. This is because, where the leaf constraint is higher, the tables handling the deeper levels span multiple stages (Leo-NT). Doubling these tables for Leo would require more stages than what are available. However, as the leaf constraint reduces, the tables shrink, freeing up more stages and thus allowing the double-sized tables to grow.



(a)

(b)

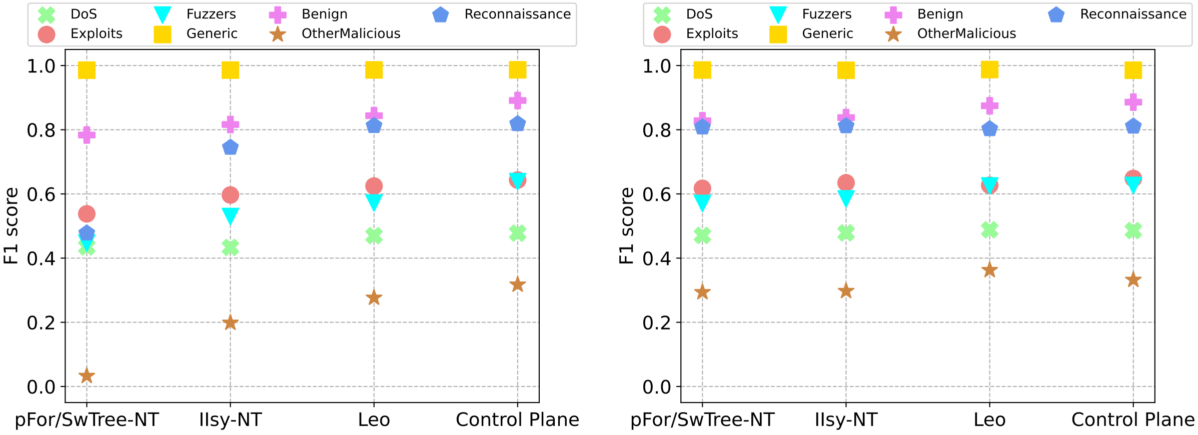
Figure 19: CICIDS-2017 classification accuracy broken down per class for (a) SRAM and (b) TCAM.



(a)

(b)

Figure 20: UNSW-NB15 classification accuracy broken down per class in a benign vs malicious scenario with (a) SRAM and (b) TCAM.



(a)

(b)

Figure 21: UNSW-NB15 classification accuracy broken down per class (for 7 classes) with (a) SRAM and (b) TCAM.

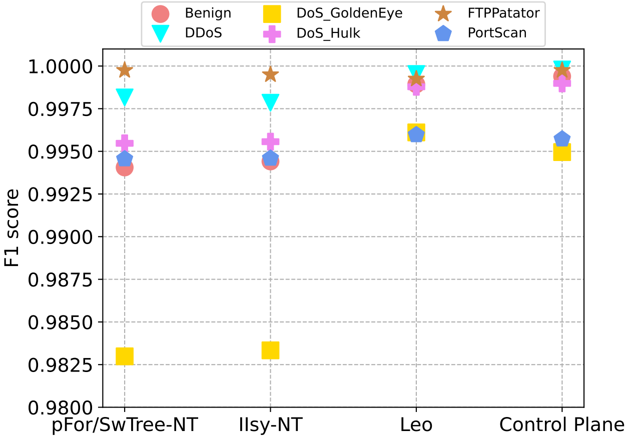


Figure 22: Zooming into the top 6 classes in figure 19(b).

	Feature Name	CICIDS-2017		UNSW-NB15	
		SRAM	TCAM	SRAM	TCAM
Stateless	Destination Port	✓	✓		
	Forward TTL			✓	✓
	Backward TTL			✓	✓
Stateful 8-bit	Backward packet length min.	✓	✓		
	Backward packet length max.		✓		
	Forward segment size min.		✓		
	Forward initial. window advertisement			✓	✓
Stateful 16-bit	Forward flow size	✓		✓	✓
	Forward initial bytes	✓			
	Backward initial bytes	✓			
	Flow IAT min.		✓		
	Backward IAT min.		✓		
	(Dst. IP, Src. Port) count			✓	✓
	(Src. IP, Dst. Port) count			✓	✓
Total (stateful)		56			

Table 2: A breakdown of the features used for the 1.04M flows data points in figure 14.

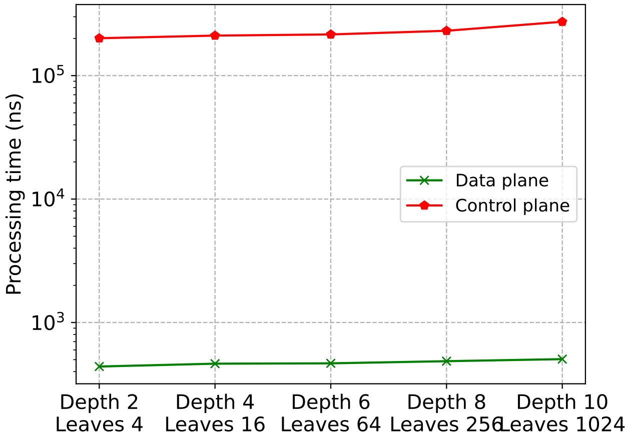


Figure 23: Average time to classify packets when routed through the data plane versus via the control plane.