Re-architecting Traffic Analysis with *Neural* Network Interface Cards

Giuseppe Siracusano
NEC Laboratories Europe

Salvator Galea
University of Cambridge

Davide Sanvito
NEC Laboratories Europe

Mohammad Malekzadeh
Imperial College London

Gianni Antichi
Queen Mary University of London

Paolo Costa
Microsoft Research

Hamed Haddadi
Imperial College London

Roberto Bifulco
NEC Laboratories Europe

Abstract

We present an approach to improve the scalability of online machine learning-based network traffic analysis. We first make the case to replace widely-used supervised machine learning models for network traffic analysis with *binary neural networks*. We then introduce **N**eural **N**etworks on the **NIC** (`N3IC`), a system that compiles binary neural network models into implementations that can be directly integrated in the data plane of SmartNICs. `N3IC` supports different hardware targets, and it generates data plane descriptions using both micro-C and P4 languages.

We implement and evaluate our solution using two use cases related to traffic identification and to anomaly detection. In both cases, `N3IC` provides up to a 100x lower classification latency, and 1.5-7x higher throughput than state-of-the-art software-based machine learning classification systems. This is achieved by running the entire traffic analysis pipeline within the data plane of the SmartNIC, thereby completely freeing the system's CPU from any related tasks, while forwarding traffic at line rate (40Gbps) on the target NICs. Encouraged by these results we finally present the design and FPGA-based prototype of a hardware primitive that adds binary neural network support to a NIC data plane. Our new primitive requires less than 1-2% of the logic and memory resources of a VirteX7 FPGA. We show through experimental evaluation that extending the NIC data plane enables more challenging use cases that require online traffic analysis to be performed in a few microseconds.

1 Introduction

Online traffic analysis is a fundamental building block in today's networks, as it enables traffic classification [2, 5, 14, 26], security [10, 25, 31] and application-specific traffic forwarding strategies [40]. The complexity of network traffic patterns and the use of encrypted communications are driving the widespread adoption of traffic analysis based on Machine-Leaning (ML), implemented on commodity servers [13]. However, it is challenging to meet the throughput and latency requirements of modern networks while performing ML-based traffic analysis [47]. Current high-performance solutions use programmable network interface cards (NICs) [12, 29, 48] to offload parts of the traffic analysis (e.g., flow statistic collection [1, 3, 28]) directly in their data plane, while still performing machine learning inference on a separate executor, e.g., the host's CPU. Unfortunately, moving the collected flow statistics across sub-systems introduces an important bottleneck [30], forcing high throughput solutions to send collected data to the ML executor in batches, thus sensibly increasing the processing latency (§ 2).

Recognizing that running ML inference *within* the network data plane would avoid data movements and solve the issue, state-of-the-art solutions implement widely used techniques, i.e., Decision Trees and their ensembles (Random Forests), using match-action tables, which are available within a NIC data plane [8, 55]. However, these solutions rely on expensive TCAM memories, and fitting Decision Trees in match-action tables requires restricting their depth to a few levels, thus impacting their accuracy. More specifically, [55] reports a maximum of five levels implemented on the NetFPGA, while [8] supports only Decision Trees of depth four on the Barefoot Tofino. Therefore, currently, network operators have to compromise between throughput, latency, or accuracy.

In this paper, we propose a new approach that efficiently leverages programmable NICs' hardware (and can achieve high throughput and low latency) while maintaining comparable accuracy with respect to existing ML-based traffic-analysis solutions implemented in software. The key insight is to exploit binary neural networks (BNNs) [15], a recently-proposed ML model targeting battery-powered edge devices. We show that BNNs can provide better classification accuracy than Decision Trees and Random Forests on the tested traffic analysis tasks (§ 3). Importantly, BNNs use single bits to represent inputs and weights, which provides two critical properties: (i) they exhibit a very compact memory footprint even for larger models; (ii) unlike mainstream Deep Neural Networks (DNNs), BNNs require only simple operations such as XOR and population count. This enables the implementation of efficient BNNs executors in a NIC's data plane,

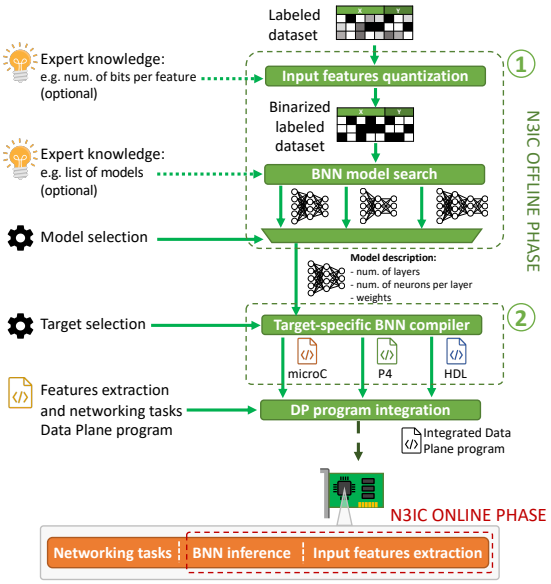

Figure 1: N3IC overview. Users provide a labeled dataset. N3IC uses it to generate a binary neural network model, which is then compiled to a data plane program for a target NIC.

without requiring expensive resources, such as TCAMs.

Building on this insight, we developed N3IC, a complete solution to perform network traffic analysis using BNNs with commodity programmable NICs. N3IC comprises two key components (Figure 1, § 4): (1) a framework to train a BNN using a labeled dataset provided by the user, and (2) a compiler that translates the trained model into target-specific executable code. To show the generality of our approach, we implement two compiler backends: one targeting micro-C, a subset of the C language used by Netronome SoC-based NICs [29], and one targeting the P4 language [6]. The latter enables compiling to a growing set of P4-enabled NICs [21], including FPGA-based NICs using the P4->NetFPGA toolchain [16].

Furthermore, we evaluate the cost of providing BNN execution as a *native* hardware primitive that can be exposed to high-level programming languages (e.g., using P4's *extern*). We prototype this on the NetFPGA using RTL description language and show it only needs a modest 1-2% of a Xilinx Virtex7 FPGA's logic resources. While prior work has shown the potential of implementing ML models on FPGA [24, 51], they target ML models for application-level data processing, which has millisecond-scale latency requirements (as opposed to microsecond), and they are typically based on FPGA monolithic implementations. To the best of our knowledge, we are the first ones integrating a streamlined BNN executor, tailored to network traffic analysis models, *within* the NIC data plane.

We evaluate N3IC across different hardware platforms using traffic classification, security anomaly detection and network tomography as use cases (§ 6). Results show that N3IC can perform traffic analysis with high accuracy and with latency in the microseconds, for millions of network flows per second, while processing packets at NICs' line rate. Compared to a similar system that implements the traffic analysis on a general-purpose CPU (with packet forwarding and feature extraction still offloaded to the NIC), N3IC provides up to 7x higher throughput and up to 100x lower latency.

Contributions. In this paper, we:

- demonstrate that BNNs provide high accuracy and low memory footprint for the selected traffic analysis use cases.
- design and implement an end-to-end system that performs traffic analysis in programmable NICs' data plane: this includes a framework to train BNNs and a compiler that translates models into both P4 and Netronome's micro-C.
- develop a new hardware primitive that enables BNN inference as first-class-primitive for next-generation programmable NICs.
- evaluate our solution on three traffic analysis use cases: (i) traffic classification, (ii) anomaly detection, and (iii) network tomography.
- Source code to reproduce key results of our work is at: https://github.com/nec-research/n3ic-nsdi22

2 Motivation and Challenges

Motivation. Modern data-center networks comprise a variety of network appliances, e.g., traffic classifiers, load balancers, and security middleboxes [34, 36]. They need to handle over a million of flows per second while only incurring a few tens of microseconds of processing latency per packet to avoid affecting the end-to-end latency [11, 23].

To meet these tight requirements, mainstream systems offload the packet capture and feature extraction steps to a programmable NIC [1, 28]. Periodically, the host system polls the extracted features from the NIC, and performs the analysis step. This approach relieves the load on the host's CPU and achieves higher throughput but at the cost of higher processing latency. To illustrate this trade-off in practice, we set up an experiment in which we offload the feature extraction on the Netronome NFP4000 NIC while we execute the analysis on an Intel E5-1630 v3 CPU. The results in Figure 2 (*NIC+CPU* line) show that as the throughput increases, the processing latency scales super-linearly. For instance, at 0.2M flows per second, the latency is 42µs but if we increase the throughput to 1M flows per second, the latency grows beyond 800µs.

There are two reasons for this. First, having the feature extraction and analysis steps running on two different subsystems requires moving data, e.g. crossing the PCIe bus, which can take up to a few microseconds [30]. Second, and most critically, CPUs require input data batching to improve the per-core processing efficiency. Batching improves data locality, avoiding stalls in the CPU pipeline due to data read delay, and it allows to fill the CPU's vector processing registers, thereby increasing the overall throughput but at the expense of much higher latency. This trade-off also applies to GPUs, which extensively rely on batching to achieve high through-

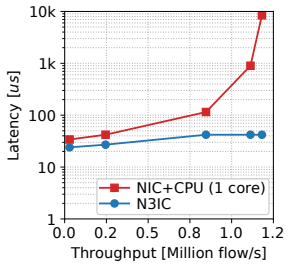

Figure 2: Processing latency when increasing throughput (flows per second) for a baseline performing feature extraction on the NIC and analysis on the CPU (*NIC+CPU*) and our system N3IC.

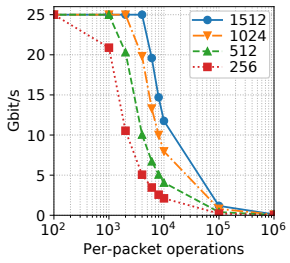

Figure 3: Forwarding throughput on a Netronome NFP4000 NIC, for different packet sizes when increasing the number of operations per packet.

put, and it explains why even network-attached GPUs [32] are not well-suited for low-latency packet processing.

A way to address the above issues is to perform the analysis directly within the subsystem that collects the data to be analyzed, i.e., within the NIC data plane. This would allows us to *(i)* avoid data movements from one subsystem to the other, and *(ii)* leverage the architectures of programmable NICs tailored to perform latency-efficient per-packet processing. As we detail in the rest of the paper, this indeed enables maintaining low latency (<40us) even at high throughput, as shown by the performance of N3IC in Figure 2.

Challenges. Existing solutions advocating for performing ML inference in the data plane of packet-processing hardware [8, 55] strictly rely on match-action tables that support ternary-matching. These resources are *(i)* not always available in a NIC data plane; *(ii)* costly, when available, since they use ternary content-addressable memory (TCAM), which is about 6x more expensive in terms of silicon area than SRAM [7]; *(iii)* limited; thus enabling only Decision Trees with small depth with an impact on the inference accuracy.[1] While using exact-matching tables may be a workaround, it would require enumerating the values to match on. For instance, to handle a single 16b feature, we may need to add 65k entries.

Enabling ML inference without the use of match-action tables resources and doing so while guaranteeing high-throughput, low latency, and high-accuracy requires solving three key challenges. First, existing programmable NICs have at most few 10s of MBs of fast on-chip SRAM memory [29, 48, 49]. Most of this memory, though, is needed to store forwarding and policy tables, leaving little space available for application data. This makes it hard to implement ML models within the NIC, often requiring trading-off model complexity for memory utilization. Second, to achieve high throughput the application logic needs to be highly parallelizable in order to fully utilize all compute resources on a NIC.

[1] Both IISY [55] and pforest [8] report an ability to run Decision Tree models with depth capped to five and four layers, respectively.

In fact, the NIC may provide a good amount of available processing resources if its architecture parallelism is leveraged. We show this in Figure 3, which plots the throughput achieved on the Netronome NFP4000 SmartNIC for different packet sizes as we increase the number of operations performed per packet. The larger the average packet size (and, hence, the less packets per second need to be processed), the higher is the number of operations that can be performed, before the forwarding throughput is negatively impacted. Finally, some ML models require complex arithmetic functions, such as multiplications or floating-point operations, which usually are not available on programmable NICs [45]. This limitation does not only affect the implementation of the ML model, but it also impacts the ability to perform pre-processing on the input features, as required by some models such as Support-Vector-Machine or K-Nearest-Neighbor.

3 Traffic Analysis with BNN

In this section, we show that binary neural networks (BNNs) are a promising option to address these challenges. Originally proposed for energy-efficient image processing on battery-powered devices, BNNs are an extreme quantized version of traditional DNNs in which each weight is encoded in just one bit rather than the typical 8-, 16- or 32-bit values. This makes them particularly appealing for our goals due to the following reasons. First, the single-bit input and weights drastically reduce their memory footprint. Second, the BNN's neurons perform a XOR between the input and weight vectors, and use as activation function the sign function on the population count (popcnt) performed on the bit vector resulting from the XOR. Therefore, they can be implemented efficiently (and with high performance) in hardware since XOR and popcnt operations are commonly supported by most platforms.

Unsurprisingly, for complex tasks such as image recognition, BNNs exhibit 3-10% points lower prediction accuracy than fully-fledged DNNs [20]. However, as we illustrate in the rest of this section, network traffic analysis models are usually much simpler and this enables BNNs to achieve an accuracy comparable (if not better) than existing implementations relying on decision trees and random forests.

3.1 Use cases

We introduce two typical traffic analysis uses cases that we use as running examples throughout the paper: IoT Traffic Classification and Security Anomaly Detection. Both use cases are general machine learning classification tasks, and therefore they are representative of common analysis use cases performed on network traffic. Further, they have open datasets, which helps making our results reproducible.

IoT Traffic Classification assigns an IoT device type to an observed network flow. For instance, this can be used in edge networks by operators to assign IoT traffic to specific Quality-of-Service classes. We focus on a 10-classes classification

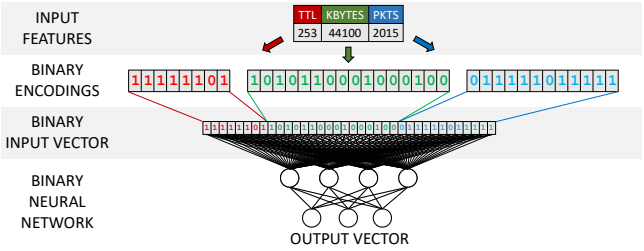

Figure 4: The binary input vector is a concatenation of features' binary encodings. Each feature is represented using the minimum required number of bits to represent its values range. The number of bits per feature has to be fixed at training time.

task, where each flow is assigned to 9 possible device categories, such as home assistants, IoT cameras, sensors, or to a 10-th class that includes *anything else*, e.g., smartphones or laptops network traffic. We use 17 flow-level features to perform the classification. Examples of features are the number of packets and bytes being transferred, the mean packet interarrival time or the mean packet sizes. We remark that the selected features are not specific to this use case: they are widely supported in open source tools and used in production settings [35]. We use the dataset published by [44].

Security Anomaly Detection is about flagging network flows that are related to security issues, such as Denial-of-Service attacks, port scans, etc. This is a network analysis task widely applied in networks of any size, and in different scenarios including telecom operators networks, datacenters, and enterprises. For this task, flows are classified into two classes, i.e., good or bad. Usually, this kind of classification is used to potentially trigger more expensive downstream analysis on the traffic, and it has the goal to capture the large bulk of potentially malicious network interactions, rather than guaranteeing complete protection. For this task, in addition to the 17 flow-level features reported earlier, we add 3 additional features that look at the behavior of multiple flows. For instance, we consider the number of flows from a single source IP address. Like in the previous case, these features are well-known and widely adopted in operational settings. We use the dataset published by [27].

3.2 BNN Analysis Pipeline

To apply BNNs, we have to define the input features quantization strategy, to convert float and integer numbers into BNN's binary features. Then, we perform the training of the binary neural network using the labeled dataset. Finally, we evaluate the classification performance with previously unseen data.

Input preparation Previous work on BNNs introduces a first regular non-binary network layer that is trained together with the remaining binarized layers. This enables "learning" the quantization strategy for the features, but at the same time, it introduces multiplication operations within the first layer. We cannot afford to perform such operations in the data plane. Therefore, we designed a different quantization approach (Figure 4): we use as input to the BNN the concatenation of the flow feature values' binary representations. For instance, an input feature in the range 0-255 can be represented by an 8b vector. Our approach has two advantages: it does not require any additional processing since we reuse the hardware representation of the features; and it allows to assign to the features the number of bits their value ranges require. For instance, a single vector of 64b can be used to represent 4 features on 16b, or 3 features on 16b and 2 features on 8b; and so on.

BNN Training Like other ML models, BNNs need to be trained offline on a training dataset, in order to define the values of the weights that will be used during the online analysis phase. We perform training using the technique from Courbariaux and Bengio [9], which is based on a canonical back-propagation algorithm. This solution trains the network using float values, but it ensures that the BNN's weights converge to values included in the [-1, 1] range, and that they are normally distributed around 0. This helps in reducing the loss of information when the float weight values are mapped to just two values, i.e., 0 and 1 [9].

BNN traffic analysis performance We test three different BNNs architectures, each with 256 input binary features and three fully-connected layers. The three models differ by the number of neurons in the hidden layers: [32, 16, 10]; [64, 32, 10]; [128, 64, 10]. For both datasets, we use a 256b input vector. Although we have 17 and 20 features for the two cases, respectively, we can represent different number of features with the same binary input vector size by changing the number of bits used to represent each feature, as mentioned earlier (cf. Figure 4). We compare the BNNs to Decision Trees (DT) and Random Forests (RFs). For DTs, we vary the depth of the tree, between 3 and 10. RFs are an ensemble of DTs, therefore they have as an additional hyperparameter the number of trees, which we vary between 3 and 5. For readability, since the trends are similar, we only plot a subset of the results in the figures, i.e., three depth values of 3, 6, and 9, and always 5 trees for the RF. In all the tests, we perform 5-fold cross-validation, and report averaged results.[2]

In Figure 5 and Figure 6 we plot the classification accuracy vs the amount of memory required by the ML models, for the IoT and Security use cases, respectively. In the top plots, we do not make distinction between memory of type SRAM, used by BNN implementations, and of type TCAM, required by DT and RF implementations. Here, we can observe that the two larger BNNs achieve an accuracy that is closer to that of DTs and RFs of at least depth 6. The two larger BNNs achieve 96% and 97.4% using 2.5KB and 5.5KB of SRAM, vs 97% and 96.9% accuracy of DT6 and RF6, using 1.3KB and 6.4KB of TCAM, respectively. The smaller BNN achieves 92.4% accuracy using 1.2KB of SRAM. In the Security dataset, the classification is harder, and only the

[2]The IoT dataset is balanced across the 10 classification categories, with each category having 43k distinct flows. In the Security dataset, we have a binary classification with 164k anomalous and 90k normal flows.

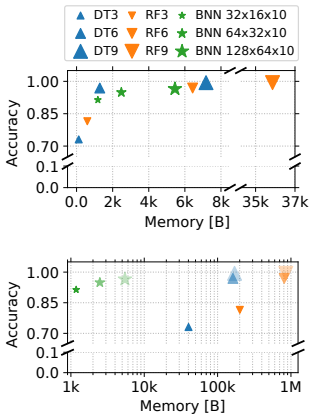

Figure 5: Accuracy vs Memory (bytes) scatter plot for DT and RF on the IoT dataset. BNNs always use SRAM-based implementations. DTs and RFs use TCAM (top) and SRAM (bottom) implementations.

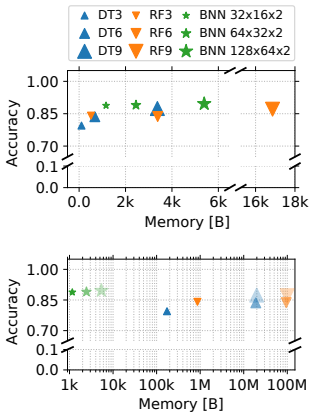

Figure 6: Accuracy vs Memory (bytes) scatter plot for DT and RF on the Security dataset. BNNs always use SRAM-based implementations. DTs and RFs use TCAM (top) and SRAM (bottom) implementations.

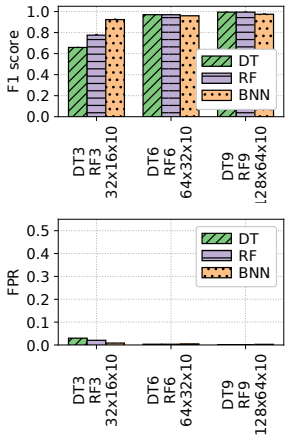

Figure 7: F1 score (top) and FPR (bottom) of BNN models on the IoT dataset, compared to DTs and RFs.

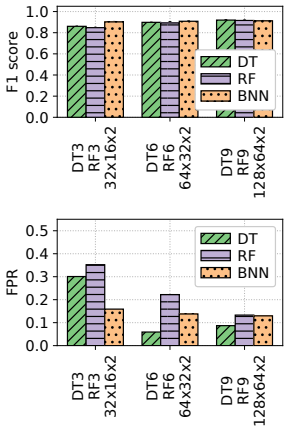

Figure 8: F1 score (top) and FPR (bottom) of BNN models on the Security dataset, compared to DTs and RFs

larger DT9 and RF9 achieve accuracy above 90%, using respectively 3.4KB and 16.9KB of TCAM. The smallest BNN achieves 91.1% accuracy using just 1.2KB of SRAM. However, it should be noted that ternary matching with TCAM is roughly 6x more expensive than binary matching with SRAM, in terms of required silicon resources [7] and TCAM is often not available on NICs.

SRAM implementations: To compare the memory requirements when targeting similar hardware, in the bottom plots of Figure 5 and Figure 6 we show the memory consumption of DTs and RFs when using SRAM-based implementations. As described in [55], in the absence of TCAM support from the hardware target, all the values of the features selected by model fitting have to be enumerated as appearing in the data. Given that some of our features are flow statistics, the values they can potentially assume range from the minimum to the maximum observed from the data. In fact the memory requirements for DTs and RFs grow orders of magnitude larger (in this case, the plots have the x axis in log scale). Even the smallest DT3 model requires at least 40.2KB of SRAM for the IoT case, and 173.3KB for the Security case.

F1-score and FPR: We now look more carefully at the classifier performance, reporting F1-score and False Positive Rate (FPR) for the tested models. The F1-score is a harmonic mean of Precision and Recall, whereas the False Positive Rate tells the quota of negative samples mis-classified as positive, in a two-classes classifier. For this metric, in the IoT case that has 10 classes, we use a 1-vs-all strategy. The BNN models achieve always better F1-score when compared to the smallest DT and RF models, in both use cases. For larger models, the F1-score is in the range 88-91.6 in all cases, showing relatively small variations among classifiers.

For the FPR, it is important to consider this metric in relation to the *Recall* of the classifier. In fact, a low FPR may be a symptom of a classifier assigning very few samples to the positive class. We can see this in Figure 7. DT3 and RF3 appear to have a relatively good FPR (3.0% and 2.1%). However, these low FPRs are due to the classifiers inability to identify the positive class. In particular, as captured also by F1-scores, DT3 and RF3 have a low Recall of 73.1% and 81.5% for the IoT case, whereas the smallest BNN has Recall at 92.4% with an FPR of 0.8%. We can see a similar issue in the Security use case (Figure 8). For instance, DT6 has FPR at 5.9% but Recall at 88.2%, whereas the smallest BNN has a higher FPR of 15.9% with a better Recall at 95.1%. Here, it should be noted that in this use case a reasonably higher FPR is not necessarily an issue. The anomaly detection is often used as a filter, before performing more expensive analysis on the flows classified as suspicious, e.g., diverting the traffic to a Scrubbing Center [33]. We provide more results in Appendix.

4 System design and implementation

We now present the design and implementation of `N3IC`, our end-to-end solution that enables to perform traffic analysis within a NIC data plane using BNNs (cf. Figure 1).

`N3IC` operations `N3IC` takes a training labeled dataset as input, and outputs programs that can be integrated into a target device's data plane. Currently, we support outputs in micro-C and P4 languages, targeting SoC-based Netronome SmartNICs and PISA-based architectures, respectively. `N3IC` entirely automates the generation of the BNN model and its implementation in the target data plane programming language. However, programmers need to perform the final integration step, to connect the input features extracted from the network packets with the programs generated by `N3IC`. In fact, feature extraction may happen in different ways, and

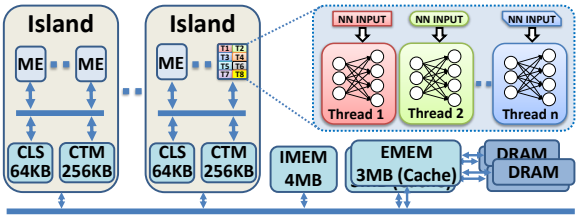

Figure 9: The architecture of a Netronome NFP4000's programmable blocks and the BNN processing with N3IC-NFP.

it is generally dependent on the implemented data plane features [3, 8]. Furthermore, since N3IC leverages the same hardware in the switching chips' data plane used also for other tasks, the networking and traffic analysis functionality can usually be intertwined, e.g., in the case of programs targeting the PISA architecture the same pipeline stages may take forwarding decisions and compute BNN's neurons. This is a process that in the future may be automated too, as data plane composability technology matures [46].

BNN model generation We described in § 3 the input feature quantization and training processes for BNNs. N3IC applies these processes on the provided labeled dataset. For input quantization, N3IC takes *hints* from the programmer, who can provide the number of bits that should be used to represent each feature. For instance, the programmer may have expert knowledge about what value ranges a given feature may have. Otherwise, N3IC can perform an automatic assignment of features to the binary input features vector, using the range of values observed in the dataset as a guide. Once the feature quantization strategy is fixed, N3IC starts a model search task. During this task several models are trained, and their performance on the provided data set is tested using K-fold cross validation. Also in this case the programmer can guide the process, providing a list of models to test or limitations on the maximum model size. Our current implementation performs a simple exhaustive search over a predefined (or programmer-provided) set of models, however, this step can also be enhanced with techniques that implement more sophisticated ML architecture search solutions [38].

At the end of these two steps, N3IC generates a BNN model implementing an MLP architecture, described by the number of layers, number of neurons per layer, and the corresponding weights. This description is finally passed to the target-specific BNN compilers, which generate the data plane programs that implement the BNN executors. We describe these implementations next.

4.1 SoC NIC: Netronome NFP4000

The NFP4000 architecture, shown in Figure 9, comprises tens of independent processing cores, which in Netronome terminology are named micro-engines (MEs). MEs are programmed with a high-level language named *micro-C*, a C dialect. Each ME has 8 threads, which allow the system to efficiently hide memory access times, e.g., context switching

Algorithm 1: BNN layer processing function. Weights and inputs are in groups of `block_size`.

Input : x input vector, w weights matrix, n num. of output neurons;
Output : y output vector

```
1  block_size ← 32;
2  assert(n % block_size == 0);
3  sign_thr = (len(x) * block_size)/2;
4  y[n/block_size] ← {0};
5  for neur ← 0 to n − 1 by 1 do
6      tmp ← 0;
7      for i ← 0 to len(x) − 1 by 1 do
8          tmp += popcnt(w[neur][i] ⊙ x[i]);
9      end
10     if tmp >= sign_thr then
11         tmp_out |= (1 << (neur % block_size));
12     end
13     if (neur + 1) % block_size == 0 then
14         y[neur] ← tmp_out;
15         tmp_out ← 0;
16     end
17 end
```

between threads as they process different packets. MEs are further organized in islands, and each island has two shared SRAM memory areas of 64KB and 256KB, called CLS and CTM, respectively. Generally, these memory areas are used to host data required for the processing of each network packet. Finally, the chip provides a memory area shared by all islands, the IMEM, of 4MB SRAM, and a memory subsystem that combines two 3MB SRAMs, used as cache, with larger DRAMs, called EMEMs. These larger memories generally host forwarding tables, access control lists, and flow counters. The BNN executor implementation has to share the MEs and memory resources with packet processing tasks, thus, it has to strike the right balance between the needs of quickly forwarding network packets and running BNN inference. For both processing tasks the main bottleneck is the memory access time. Therefore, selecting the memory area to store BNN's weights plays a major role in our design.

If the BNN is small, like in our cases, it is worth considering the fastest available on-chip memories, i.e., the CTM and CTS, with an access time of less than 100ns [29]. However, the CTM memory is usually dedicated to packet processing tasks, being the memory used by the NFP to store incoming packets and making them available to the MEs. Thus, using the CTM may impact packet processing and should be avoided. Because of this, our implementation loads the NN's weights at configuration time in the CLS memory. Then, to run the BNN, N3IC outputs a function that can be run within an ME's thread, and which performs Algorithm 1. This function implements the BNN executor, with input and weights packed in 32b integers (i.e., `block_size` is 32). As a consequence, multiple threads can perform BNN executions in

Algorithm 2: *popcount* implementation. $\texttt{X}|_y$ is the y-times concatenation of the binary number `X`; $\texttt{Z}||\texttt{W}$ is the concatenation of the binary numbers `Z` and `W`.

Input : n input number;
Output : c output counter

$B \leftarrow \lceil log_2(n+1)/8 \rceil * 8$;
$L \leftarrow log_2 B$;
$bits[L] \leftarrow \{1,2,4,...,B/2\}$;
$masks[L] \leftarrow$
$\{01|_{B/2}, 0011|_{B/4}, 00001111|_{B/8}, ..., 0|_{B/2}||1|_{B/2}\}$;
$c \leftarrow n$;
for $i \leftarrow 0$ **to** $L-1$ **by** 1 **do**
  $c \leftarrow (c \,\&\, masks[i]) + ((c >> bits[i]) \,\&\, masks[i])$;
end

parallel (Figure 9), and it is up to the programmer to decide when and how many threads to use for the BNN execution.

For example, a typical implementation would have, at boot time, each of the MEs' threads registering itself to be notified of packets reception. The NFP takes care of distributing packets to threads on a per-flow basis. This is a standard approach when programming the NFP. Thus, whenever a new packet is received, the NFP copies its content in an island's CTM, and notifies one of the island's threads to start packet processing. The notified thread can perform regular packet processing tasks, such as parsing, counters update, forwarding table lookups. The programmer can include in this context a trigger condition to start the processing of the BNN executor, by calling the function provided by `N3IC`. An example of triggering condition is the the reception of a predefined number of packets for a given flow.

4.2 BNN->P4->NetFPGA

P4 [6] is a domain-specific, platform-agnostic language for the programming of packet processing functions. `N3IC` implements a compiler that transforms BNN descriptions into BNN executors described with P4, targeting a PISA architecture. In principle, a P4-based implementation allows us to separate the `N3IC`'s BNN executors from the underlying hardware-specific details, thus it should make the executor portable to any PISA architecture. However, as we will discuss at the end of the section, the target hardware architecture has still an important impact on the final implementation.

Compiling BNN to P4. The `NNtoP4` compiler takes as input the BNN description created by the model generation step, and generates $P4_{16}$ code for a generic P4 target based on the PISA architecture. PISA is a spatial forwarding pipeline architecture, with a number of match-action units (MAUs) in series. A packet header vector (PHV), containing both the input packet and metadata information, is passed through the MAUs to perform the programmed processing tasks. Each MAU combines a table memory structure, for quick lookups using the PHV fields, with arrays of ALUs that perform operations on such fields. The code generated by `NNtoP4` implements a function, on top of the PISA architecture, which reads the input value from the PHV, performs the NN execution and writes back to a PHV's field the result of the computation. The NN weights are stored in the MAUs' fast memories to enable runtime reconfiguration. The generated P4 code also includes headers definition, parser, de-parser and control blocks. The code can therefore be easily extended to integrate with any other required packet processing function.

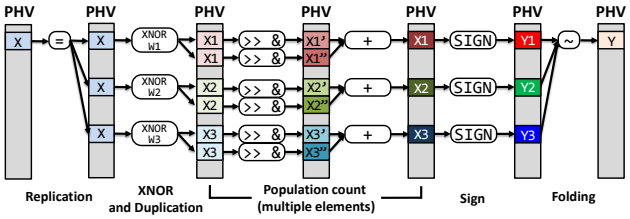

Figure 10: The logical steps required to implement a BNN using a PISA architecture.

The basic operations needed to implement Algorithm 1 are (1) XNOR, (2) popcount and (3) SIGN function. Executing a XNOR and a comparison (SIGN) is readily supported by the P4 language. Unfortunately, the popcount operation is not. The main issue is that its execution time depends on the input size, which makes popcount difficult to implement in networking hardware, and therefore not supported in the PISA architecture. To overcome this issue using only current P4 primitives, we adapted the solution proposed in [4] (Item 169), as shown in Algorithm 2. The idea is to implement the popcount by combining basic integer arithmetic and logic operations in a tree structure whose depth is dependent on the input size.[3] A tree structure can be easily pipelined, with the processing of different tree's levels assigned to different pipeline's stages, thus achieving pipeline-level parallelism.

Overall, the processing includes five steps, each one mapped to a logical pipeline stage, except for the popcount which requires multiple stages, depending on the input size (cf. Figure 10). First, the NN input is replicated in as many PHV fields as the number of neurons to exploit the parallel processing on multiple packet header fields. Specifically, this corresponds to an unrolling (or partial unrolling) of the first `for` cycle of Algorithm 1. Second, each field, containing a copy of the NN input, is XNORed with the corresponding weight. The resulting value is further duplicated to additional fields to implement the shift, AND and sum as described in Algorithm 2. The outcome of each popcount is then compared with a threshold to implement the SIGN function, whose result is the output of each neuron. Finally, the resulting bits, stored in one PHV field for each neuron, are folded together in a single field. Depending on the NN depth, `NNtoP4` replicates and concatenates the described operations as many times as the number of layers to obtain the complete MLP execution.

For hardware targets, it is worth noticing that the PHV

[3] See [52], chapter 5, for a longer description of the algorithm.

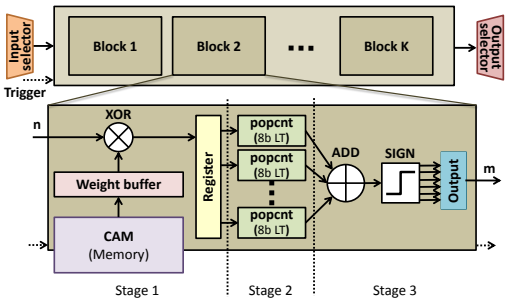

Figure 11: Hardware design of the BNN Executor module.

size limits the number of neurons the pipeline can execute in parallel. This is due to the need to replicate the input in the PHV to enable parallelism at the MAU level.

Generating P4 code for the NetFPGA. The NetFPGA is a 4x10GbE FPGA NIC, incorporating a Xilinx Virtex-7 FPGA. We integrate `N3IC` in the reference NIC project provided with the NetFPGA-SUME code base. We used the P4->NetFPGA workflow [16] to port the generated target-independent P4 code to the NetFPGA platform. The P4->NetFPGA workflow is built upon the Xilinx P4-SDNet [54] compiler and the NetFPGA-SUME code base. It translates P4 code to Verilog, and integrates it within the NetFPGA pipeline.

The P4->NetFPGA workflow required several adaptations to the `NNtoP4` compiler, in order to meet the FPGA resources and timing constraints. First, the P4-SDNet compiler does not support `if statements` among the operations of a MAU. Thus, we replaced all the if statements required by the SIGN function using a combination of bitwise logic operations and masks. Second, MAUs use the CAM IP core from Xilinx to implement lookup tables, which restricts the maximum width size that can be used for each entry. Consequently, a maximum of 32B can be fetched from memory every time a table is called, limiting the number of neuron weights that could be loaded in parallel by each table. To overcome this issue we had to write the weights as constant values in the MAU's operations code, effectively trading the possibility to perform runtime reconfiguration with the ability to compute more neurons in parallel. Finally, P4-SDNet is capable of performing a large number of operations on a field in a single MAU. This is in contrast with ASIC targets, which are instead usually constrained to execute a single operation per MAU [45]. This allowed us to describe several steps of a BNN computation in a single MAU, thus reducing the number of MAUs required to implement the BNN computation.

5 Hardware support for BNNs

While `N3IC` can generate data plane programs that implement a BNN executor, a native support for BNNs could enable more challenging use cases. In this section, we present the implementation of a data plane's hardware primitive to run BNN, and an example of a use case that can benefit from it.

5.1 BNN inference primitive

BNN executors have been presented in the past, however, their implementations were more generally targeted to applications within devices dedicated to AI and ML workloads, e.g., cameras. Instead, our target is to design a BNN executor integrated within the data plane of a NIC. This changes the implementation constraints. Most notably, our executor targets smaller models, and it is designed to fetch input data from the internal data plane data buses. We target the NetFPGA prototyping platform, and design our BNN executor in HDL.

Figure 11 shows the architecture of our BNN executor. The module is composed of multiple blocks. Each of them performs the computation of a single NN layer, and can be parametrized providing the sizes n and m for the input and output vectors, respectively. Together, the blocks build a BNN Executor for specific BNN architectures. For instance, three of these blocks are required to build a 3 layers MLP. The NN layer weights are stored in the FPGA on-chip memories, i.e., Block RAM (BRAM). The BRAMs are organized as tables with a number of rows dependent on the number of neurons, and with a width of 256b. Each row can be read in 2 clock cycles and, depending on the size n of the input vector, can store one or multiple weights, e.g., 1x256b or 16x32b. The BRAMs are shared by all the blocks of a BNN module.

A single block is a pipeline of three stages. The first reads the weights from the BRAM and performs the XNOR with the input. The second performs the first step of the popcount. Here, we create Lookup-Tables (LTs) of 256 entries each, in order to associate one 8b integer (address) to the corresponding population count value. Each block has $n/8$ of these LTs. As a consequence, for a 256b input we create 32 LTs that operate in parallel. In the last stage, the LTs outputs are summed together, the sign function is applied on the final sum and its result is stored in one of the m bits of the output register. If multiple weights are placed in a single BRAM's row, the module performs the execution of several neurons in parallel.

5.2 Enabling more challenging use cases

The BNN inference primitive can enable more challenging applications that have very low processing latency requirements. To highlight this, we look at a recently presented network tomography solution: SIMON [14]. SIMON periodically sends probe packets to measure network path delays, and then it uses the collected delay measurements to infer congestion points and the size of the related queues. The analysis of probe delays is performed offline with neural networks (MLPs). The high processing latency only enables post-mortem analysis. Therefore, in its current implementation, SIMON cannot be used to create a measurement and control loop, i.e., for path selection. The probe periodicity defines the processing latency constraint and it depends on the fastest link speed [14]. For instance, probes have to be sent every 250µs and 100µs for 40Gb/s and 100Gb/s links, respectively. As a consequence, to work at modern datacenters' link speeds and in real-time, the

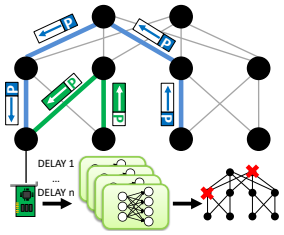

Figure 12: N3IC enables the real-time implementation of SIMON [14], using BNNs in the NIC to identify congested queue from probes' one-way delays. This can be used to implement new traffic steering policies in the data plane.

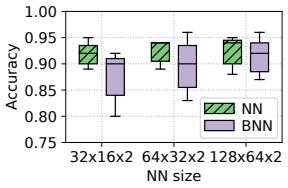

Figure 13: Box plot of the accuracies for the predicted queues in the network tomography use case. BNNs makes our approach practical while trading just a tiny amount of accuracy with respect to non-binarized NNs.

execution latency has to be lower than few tens of µs.

We tested the use case simulating a CLOS-like Fat Tree datacenter network with ns3 [50], using different link speeds and traffic workloads. Following the methodology suggested by [14], we split the problem of inferring queue sizes in multiple sub-problems, each targeting a subset of the queues. This allows us to run smaller MLPs on each of the NICs. Unlike SIMON, our approach does not infer the actual size of a queue, but it only infers which queues are bigger than given thresholds levels. This information is usually sufficient for the control plane to take a flow-steering decision (See Figure 12).

We implement SIMON with N3IC, providing as input features 19 probes' one-way delays per BNN. A NIC can run multiple BNNs, since each of them infers the congestion status of a specific queue. We show the accuracy of prediction for each of the network queues in Figure 13, comparing the BNNs accuracy to that of non-binarized neural networks. For a BNN with three layers and 128, 64, 2 neurons per layer, across all the queues of the simulated network, we achieve a median accuracy in predicting a congested queue above 92%, which is comparable with the non-binarized neural network accuracy. As we will see in § 6, the introduced BNN hardware primitive will enable running these BNNs within the processing latency required for links faster than 400Gb/s.

6 System-level Evaluation

In this section, we present the experimental evaluation of N3IC's BNN executors. We report and discuss the end-to-end performance of the use cases presented in § 3, and of the the network tomography use case from § 5. Furthermore, we report results for micro-benchmarks and resource requirements.

Testbed. Unless stated otherwise, the system-under-test (SuT) uses a machine equipped with an Intel Haswell E5-1630 v3 CPU and either a Netronome Agilio CX, with an NFP4000 processor, or a NetFPGA-SUME[4]. The Haswell is clocked at 3.7GHz, the NFP at 800MHz, and the NetFPGA at 200MHz for both the N3IC-P4 and N3IC-FPGA (i.e., using the hardware primitive) implementations. The host system runs Linux, kernel v.4.18.15. The SuT is connected back-to-back to a second machine that hosts the traffic generators and receivers. For stress tests, we use a 40Gb/s capable DPDK packet generator[5], and we use HTTP clients and ngnix as receiver, both hosted on the second machine. We always measure that the SuT is the performance bottleneck, ensuring that the setup achieves line-rate when removing the SuT from the loop.

Comparison term. We compared our prototypes with a traffic analysis system (bnn-exec) that performs feature extraction on the NIC and the analysis task in software, using binary neural networks like those employed by N3IC. bnn-exec is available at [43]. We wrote bnn-exec in C, and optimized it for the Haswell CPU, with some parts in assembler to take full advantage of the CPU's architecture features, such as AVX2 instructions. bnn-exec is faster than any other software BNN executor we tested, and performs the analysis task with performance comparable to that of optimized libraries for DTs and RFs [17]. We setup bnn-exec to read flows statistics/data from the Netronome NIC and ran bnn-exec only with the Netronome NIC since its driver is more mature than the NetFPGA's: it can better handle fast communication between the NIC and the host system. When performing analysis with bnn-exec we took into account (1) the time to read one or more flow statistics; (2) the time to run the BNN itself; and (3) the time to write back the result on the NIC. This allows us to perform a fair comparison against N3IC.

Feature extraction. Our end-to-end system need feature extraction to be implemented in the NIC's data plane.In fact, the quality of the inference tasks performed by the downstream ML model strictly depends on the quality of the extracted features. In some use cases, feature extraction may be simpler than in others. For instance, in the IoT use case the outcome of the inference task assigns flows to QoS classes. While a mis-classification is undesirable, its impact on the infrastructure is usually limited, and one may give priority to efficiency of implementation vs accuracy. Instead, in security use cases there may be a stricter need to ensure that feature extraction is robust, e.g., to protect against adversarial attacks [37].

For the tests in this section, we use two different state-of-the-art feature extraction strategies. In stress tests, we use a simpler approach that allows us to evaluate the N3IC implementations, ensuring that N3IC is the actual system bottleneck. In this case, the NIC stores the per-flow features in a hashtable, using the flow's 5-tuple as lookup key. When a packet is received, the corresponding flow's features are retrieved from the hashtable and updated. If the lookup produces a miss, the packet is considered as belonging to a new flow. Entries are removed from the hashtable lazily, if no packets for the corresponding flow are received in a given time window. In

[4]The Haswell CPU was produced with a 22nm factory process, i.e., a technology comparable to the NFP4000 (22nm) and NetFPGA Virtex7 (28nm).

[5]https://git.dpdk.org/apps/pktgen-dpdk/

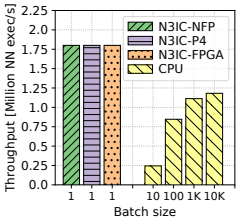

Figure 14: For IoT and Security use cases, N3IC is 1.5x-7x faster than bnn-exec, while forwarding packets at 40Gb/s.

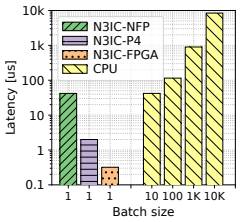

Figure 15: For IoT and Security use cases, N3IC implementations can provide at least 10-100x lower latency than bnn-exec.

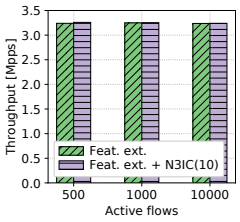

Figure 16: N3IC-NFP throughput for the IoT use case when analyzing 500, 1k, and 10k parallel TCP flows.

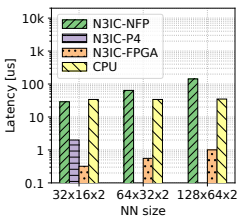

Figure 17: N3IC-FPGA can support the network tomography use case even in fast 400Gb/s networks with probes sent every 25µs.

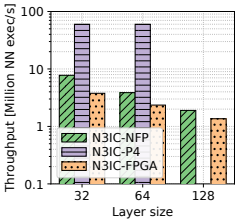

Figure 18: Maximum throughput in number of BNNs execution per second for N3IC BNN executors.

this approach, only the TCP's connection establishment is tracked, and no further connection tracking is performed.

In a second approach, we use a more complex solution performing full TCP-connection tracking. A connection tracking automaton validates that a received packet belongs to the 5-tuple flow (e.g., checking sequence numbers), before performing the features update as in the simpler approach. We used the TCP-connection tracking implementation of Flowblaze [36] that also allows us to change the behavior for sequence number checking, e.g., using either *window shifting* or *window advancing* solutions.

In both cases, we only track flow-level features. Collecting the 3 host-level features used in the Security use case requires more complex operations, which we did not implement since the impact of such features on the BNN classification accuracy is negligible (Cf. Appendix for a detailed report). Flow-level features can be: directly extracted from the packet headers (e.g. protocol number), computed by accumulating values extracted from packet headers (e.g, total transferred bytes) or derived from the calculation of flow level metrics (e.g. packet interarrival, mean flow size). In the latter case difference based metrics (e.g. flow duration) are computed for each packet in the flow, while mean based metrics are only partially computed (i.e. total and number of values are stored separately) per packet and then finalized (i.e. total/number) each time the feature is fed to the NN. Additional per flow values are stored in order to compute the flow level metrics (Table 4 in Appendix). The computation of the per-flow statistics is a memory-bound operation so the extra overhead due to the metric computation is negligible respect to the cost of accessing the flow tables. We implement the feature extraction strategies both in the Netronome NFP and in the NetFPGA, for which we report its resources consumption in Table 1. We refer to the two Feature Extraction (FE) strategies as *simple* FE and *advanced* FE, respectively.

6.1 End-to-end performance tests

In all the end-to-end tests, we measure the analysis throughput and latency, *while the system-under-test forwards network traffic at 40Gb/s within the NIC (NFP4000 or NetFPGA).*

Traffic analysis use cases We perform two different tests to measure the end-to-end N3IC performance with the use cases from Section 3. First, we run a stress test generating a large number of small packets with the DPDK packet generator, then we perform a performance test with real TCP flows generated by HTTP clients and nginx. For the stress test, the provided traffic contains 1.8M flows per second.[6] This is a challenging load for a single server, being more common in ToR switches handling traffic for high throughput user-facing services [23]. If N3IC can meet this performance goal, it is likely to be capable of handling a large range of ordinary use cases. For the TCP tests, we vary the number of flows between 500 and 10k, and always generate 40Gb/s of traffic.
Baseline: We measured the NIC performance when only collecting flow statistics with the simpler approach introduced earlier. The Netronome provides its 40Gb/s line rate only with packets of size 256B (18.1Mpps) or bigger. This is achieved using 90 out of the 480 available threads, and it is in line with the device's expected performance for such class of applications. In fact, the NFP can efficiently hide hash-table lookup latencies by distributing the processing on multiple threads, while consistently assigning flows to different threads. This avoids expensive locking of the hash-table, since different parallel executors do not access the same entry. The NetFPGA, instead, is capable of forwarding 40Gb/s with minimum size (64B) packets while collecting flow statistics, in any case.
Stress Tests: We use the smaller BNN models reported in § 3 to test N3IC performance, since they achieve comparable accuracy with the larger DTs and RFs models. We summarized the throughput results in Figure 14. N3IC implementations can all achieve the offered throughput of 1.81M flow analysis/s. Instead, even if using larger batch sizes, bnn-exec is unable to cope with such load, when running on a single CPU core. bnn-exec maximum throughput is 1.18M analyzed flows/s, when using very large batches of 10K flows. More interestingly, Figure 15 shows that N3IC implementations provide also a low processing latency, with a 95-th percentile of 42µs

[6] That is, an average of 10 packets per flow at 40Gb/s@256B.

for N3IC-NFP, and only 2µs and 0.5µs for N3IC-P4 and N3IC-FPGA, respectively. In comparison, for bnn-exec to achieve a throughput above the 1M flows/s, the processing latency is 1ms and 8ms with batch sizes 1K and 10K, respectively.

TCP Test: we run an additional experiment, using the IoT application, to check the functionality of N3IC with flows generated by HTTP clients and *nginx*, when using the second feature extraction strategy with full TCP tracking. The HTTP clients generate 40Gb/s distributed among 500, 1k, and 10k parallel flows. Since TCP flows have larger average packet size (close to the maximum of 1.5KB), this corresponds to about 3.2Mpps at 40Gb/s line rate. We further instrument N3IC to perform inference on a flow after every (10, 100, 1000) received flow's packets. This corresponds to up to over 320k ML inferences per second. Figure 16 shows that N3IC can forward all the received packets, while collecting statistics and performing ML inference (we show only N3IC-NFP, and for inferences every 10 flow's packets, since results for the other experiments and for N3IC-FPGA are similar). An interesting observation is that the NFP's throughput does not change when adding N3IC inference load. This happens since feature extraction requires memory lookups, whereas BNN inference requires mostly processing power from the NFP's MEs, thus the two workloads can be efficiently co-located.

Network Tomography When testing the network tomography use case, the NIC stores the one-way-delay value for the received network probes, before passing them to the analysis engine, i.e., either N3IC or bnn-exec. Here, processing latency is the critical performance indicator. Figure 17 shows that bnn-exec provides a processing latency of about 40µs, which is within the budget of 100µs.[7] However, upcoming network links of 400Gb/s could not be supported, since they would lower the periodicity of the probes to 25µs. N3IC processing latency for SIMON's BNNs with 128, 64, 2 neurons is 170µs for N3IC-NFP and below 2µs for N3IC-FPGA. As we further clarify next, N3IC-P4 cannot scale to run larger BNNs, and can only run the smaller 32, 16, 2 neurons networks with about 2µs of delay, at the cost of reduced accuracy. For upcoming 400Gb/s network speeds, the BNN hardware primitive enables running more accurate BNN models, while being within the processing latency requirement.

6.2 Scalability tests

We now evaluate the processing throughput and latency when varying the size of the BNN. We performed this evaluation fully loading N3IC, and by executing a single BNN layer with 256 binary inputs. We varied the number of neurons to be 32, 64, and 128.[8] Figure 18 shows that the throughput decreases linearly with the layer's size for N3IC-NFP and N3IC-FPGA. Latency, instead, increases linearly (not shown). This is expected given the design presented in § 4. In comparison, N3IC-P4 throughput results are much higher for a layer with 32 and 64 neurons. Unfortunately, results for 128 neurons are missing, since N3IC-P4 could not scale to handle such layers. We provide more insight on this in the next subsection.

[7] In this case high-throughput is not required, so we use a batch size of 1.

[8] The layer is fully-connected, therefore its size is the number of input times the number of neurons: a layer with 128 neurons has 4KB of weights, i.e., about 4x the size of the NN used for the traffic analysis use cases.

Design	LUT		BRAM	
	#	% tot	#	% tot
Reference NIC (RN)	49.4k	11.4%	194	13.2%
RN + simple Feature Extraction (FE)	50.0k	11.56%	258	17.6%
RN + simple FE + N3IC-FGPA	52.6k	12.16%	275	18.8%
RN + simple FE + N3IC-P4	145.1k	33.56%	582	39.6%
RN + advanced FE	92.0k	21.56%	458	32.6%
RN + advanced FE + N3IC-FPGA	95.0k	22.86%	475	33.8%

Table 1: NetFPGA resources usage. N3IC-FPGA requires little additional resources. N3IC-P4 uses a large amount of NIC resources due to the PISA computation model constraints.

6.3 System resources usage

We quantify the resources needed by N3IC. Compared to state-of-the-art systems like bnn-exec, N3IC does NOT use any CPU cores and keeps the PCIe bus free. It does however use additional resources on the NIC. We evaluate this referring to the BNNs used in the traffic analysis use cases.

In the NFP case, N3IC has to store the NN's weights in the NFP4000's memory system. The NNs used with the traffic analysis use cases require 1.5% of the CLS memory, and 480 threads to face the offered load, instead of the 90 required to achieve line-rate throughput when the NIC is only collecting flow statistics. Here, it should be noted that it is possible to use less threads, if a performance drop in NN inference throughput is acceptable. For instance, using only 120 threads, i.e., 30 additional threads compared to the baseline, reduces the throughput of flows analyzed per second by 10x. This still provides the ability to analyze over 100k flows per second, which is sufficient for many workloads.

In the NetFPGA cases, we measured the hardware resources required to synthesize N3IC on the Virtex7 FPGA, and compare them to the standard NetFPGA reference NIC design's resources, including the resources required to implement the feature extraction logic. Table 1 summarizes the results. N3IC-FPGA requires only an additional 0.6% and 1.2% of the FPGA's LUTs and BRAMs, respectively. The resource consumption is so small since we included in the data plane a single BNN executor module, which was dimensioned to achieve the analysis throughput measured in the tests reported in this section. Instead, the N3IC-P4 implementation requires a relatively large amount of resources, with an additional 22% for both LUTs and BRAMs. For comparison, the implementation of DTs with depth 5 in the data plane reported in [55] requires 27% and 40% of LUTs and BRAMs, respectively. This is the case because the P4 implementation embeds the BNN executor within a PISA-like pipeline targeted by the P4->NetFPGA toolchain. That is, the computations of the BNN are unrolled to be distributed on multiple PISA's

match-action stages. While this has the effect of completely pipelining the BNN execution, it also requires using a large amount of FPGA resources. That is, like it is the case for other P4 programs using the P4->NetFPGA toolchain, with N3IC-P4 the successful compilation and synthesis of the P4 program guarantees the NIC's line rate during execution. Therefore, N3IC-P4 can run a BNN inference for each received packet and still match packet forwarding line rate (cf. Figure 18). For this reason, it should also be noted that in N3IC-P4 most of the resources can be reused to implement also regular packet forwarding, since the pipeline stages required by N3IC can host forwarding rules coming from other processing tasks.

7 Discussion

What are the limitations? Since N3IC BNNs run in the data plane, only features that can be computed/extracted within the data plane can be used as input. This limits the applicability of N3IC to devices that offer such functionality. For instance, porting N3IC to a switch's data plane may be limited by the availability of input features. For similar reasons, more complex models that require application-level data, e.g., payload of packets and with KBs of input size, are not well handled by N3IC. For these kind of analysis tasks, more relevant solutions may be previous work such as Brainwave [24] and Taurus [47], or some recently presented NICs that combine specialized executors for ML models, e.g., NVIDIA EGX A100 [32] and Xilinx Alveo SN1000 [53]. In fact, although these executors are not well suited for the low latency analysis tasks addressed by N3IC (cf. § 2), they are especially designed to perform complex algorithms on larger data, with processing latency in the ms.

Is it all about scalability and performance? While N3IC improves the performance of existing traffic analysis systems, we believe the ability to perform flow-level traffic analysis entirely in the NIC can provide a tool to rethink system architectures. For instance, the ability to track the queue status of network switches in near real time (§ 5.2) would make it practical the implementation of load-aware data center load balancing schemes that take decision from the end host [18], or it could enable new congestion control algorithms.

8 Related Work

Traffic analysis with machine learning is performed by systems in many operational settings [39], e.g., for traffic classification [2, 5, 14, 26, 40] and security [3, 10, 19, 25]. Some solutions scale traffic analysis performance using NICs [12, 29, 48] that have the ability to perform feature extraction (e.g., flow statistic collection [1, 3, 28]). Unlike these solutions, N3IC enables also the execution of machine learning-based analysis within the NIC's data plane. Previous work presented a similar idea when targeting switches [8, 55], and [8] covers also the issue of selecting the subset of features that can be efficiently collected within the data plane. In N3IC we leverage the flexibility of a NIC's data plane, designed to process significantly less traffic than a switch, to relax this issue.

The idea of using binary neural networks within the network data plane was presented in some early works [41, 42]. [42] presents a conceptual design for RMT [7] switches. [41] targets end-host ML applications, in which the NIC works as a co-processor for Convolutional Neural Networks for image classification that runs on the host. We build on similar insights and extend those early ideas in many ways. First, we show the suitability of BNNs for traffic analysis use cases, comparing them with state-of-the-art ML techniques. Then, we present an end-to-end system design that builds BNN executors for different NIC architectures, starting from a labeled dataset. Finally, we present a complete evaluation of BNN executors on two NICs, propose a dedicated hardware-native implementation, and include an end-to-end evaluation of three networking use cases, with related trade-offs.

Finally, while not directly related to N3IC, recent work on the security of network applications that use machine learning is likely to influence developments in this area [22, 31].

9 Conclusion

We addressed the problem of improving throughput, latency, and efficiency of packet- and flow-level network traffic analysis, usually performed by software middleboxes and network functions. We first show that binary neural networks can replace widely-adopted decision trees and random forests, on the tested network traffic analysis tasks. Then, we make the case for implementing them in the data plane of commodity programmable NICs.We design and implement an end-to-end system composed of a binary neural network model generation module, and a compiler that generates data plane programs to execute the binary neural network model in the data plane of commodity programmable NICs (i.e., Netronome SmartNICs and P4-enabled NICs). Moreover, we also design and prototype a new hardware primitive that allows a NIC to perform BNN model execution directly. We evaluated our approach using two different NICs, Netronome NFP4000 and NetFPGA, and for a set of use cases representing a large variety of current traffic analysis applications, including traffic classification, anomaly detection and network tomography. Our results show that our system can accurately perform analyses for millions of flows per second, with low latency, while processing packets at NICs' line rates.

Acknowledgments. We thank our shepherd, Chuanxiong Guo, and the anonymous reviewers for their feedback, which have substantially improved this paper. Thanks also to Manya Ghobadi for feedback on earlier version of the work. This work is partially supported by the UK's EPSRC under the projects NEAT (EP/T007206/1), and EP/T023600/1 within the CHIST-ERA program, and the ECSEL Joint Undertaking and the European Union's H2020 Framework Programme (H2020/2014-2020), under grant agreements n. 876967 ("BRAINE") and n. 883335 ("PALANTIR").

References

[1] Accolade Technology. ANIC Host CPU Offload Features Overview, [Online; accessed 04-March-2021]. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.

[2] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019.

[3] Diogo Barradas, Nuno Santos, Lui Rodrigues, Salvatore Signorello, Fernando M.V. Ramos, and Andre Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *Network and Distributed Systems Security (NDSS)*. USENIX, 2021.

[4] M. Beeler, R.W. Gosper, and R. Schroeppel. Hakmem AI Memo No. 239. In *MIT Artificial Intelligence Laboratory, Cambridge, US*, 1972.

[5] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36(2):23–26, April 2006.

[6] 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 *Computer Communication Review, Volume: 44, Issue: 3*. ACM, 2014.

[7] 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 *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[8] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019.

[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshuao Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. In *Computing Research Repository, Volume: abs/1602.02830*, 2016.

[10] Luca Deri. Using ndpi for monitoring and security, 2021. https://fosdem.org/2021/schedule/event/nemondpi/.

[11] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mark Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish K. Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[13] Gartner. Market guide for network detection and response, 2020. https://www.gartner.com/doc/reprints?id=1-25DOQJMT&ct=210304&st=sb.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2016.

[16] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4->netfpga workflow for line-rate packet processing. In *Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

[17] intel. Daal, 2019. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onedal.html.

[18] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2017.

[19] Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. VNIDS: Towards Elastic Security

with Safe and Efficient Virtualization of Network Intrusion Detection Systems. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 17–34, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2017.

[21] Francis Matus. Distributed services architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–17. IEEE Computer Society, 2020.

[22] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. (Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs. In *Hot Topics in Networks (HotNets)*. ACM, 2019.

[23] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[24] Microsoft. Microsoft unveils project brainwave for real-time ai, 2017. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[25] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[26] Andrew W. Moore and Denis Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 2005.

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

[28] Napatech. SmartNICs features overview, [Online; accessed 04-March-2021]. https://www.napatech.com/support/resources/data-sheets/napatech-smartnic-feature-overview/.

[29] Netronome. Netronome AgilioTM CX 2x40GbE intelligent server adapter, 2018. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[30] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[31] Carlos Novo and Ricardo Morla. Flow-Based Detection and Proxy-Based Evasion of Encrypted Malware C2 Traffic. In *ACM Workshop on Artificial Intelligence and Security*, AISec'20, New York, NY, USA, 2020. Association for Computing Machinery.

[32] NVIDIA. Nvidia egx a100, 2018. https://www.nvidia.com/en-us/data-center/products/egx-converged-accelerator/.

[33] OVH. Managing a ddos attack, 2021. https://www.ovh.com/world/anti-ddos/managing-ddos-attacks.xml.

[34] OVH. What is anti-ddos protection?, 2021. https://www.ovh.com/world/anti-ddos/anti-ddos-principle.xml.

[35] Vern Paxon. The Zeek Network Security Monitor, [Online; accessed 04-Feb-2020]. https://www.zeek.org/.

[36] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. FlowBlaze: stateful packet processing in hardware. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[37] Zhiyun Qian and Z. Morley Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361, 2012.

[38] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. 54(4), 2021.

[39] Paulo Angelo Alves Resende and André Costa Drummond. A survey of random forest based methods for intrusion detection systems. *ACM Comput. Surv.*, 51(3), May 2018.

[40] Said Jawad Saidi, Anna Maria Mandalari, Roman Kolcun, Hamed Haddadi, Daniel J Dubois, David Choffnes, Georgios Smaragdakis, and Anja Feldmann. A haystack full of needles: Scalable detection of iot devices in the wild. In *Proceedings of the ACM Internet Measurement Conference*, pages 87–100, 2020.

[41] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the AI accelerator? In *In-Network Computing (NetCompute)*. ACM, 2018.

[42] Giuseppe Siracusano and Roberto Bifulco. In-network neural networks. In *Computing Research Repository, Volume: abs/1801.05731*, 2018.

[43] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. N3ic github repository, 2022. https://github.com/nec-research/n3ic-nsdi22.

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

[45] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: high-level programming for line-rate switches. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[46] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 329–343, 2020.

[47] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Neeraja Yadwadkar, Yaqi Zhang, and Kunle Olukotun. Taurus: An Intelligent Data Plane. In *P4 Workshop*, 2019.

[48] Mellanox Technologies. BlueField SmartNIC, 2019. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[49] Mellanox Technologies. TILEncore-Gx72, 2019. https://www.mellanox.com/page/products_dyn?product_family=231&mtag=tilencore_gx72_adapter_mtag.

[50] The University of Washington NS-3 Consortium. NS3 official website, [Online; accessed 10-Jan-2020]. https://www.nsnam.org/.

[51] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Field-Programmable Gate Arrays (FPGA)*. ACM, 2017.

[52] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[53] XILINX. Xilinx sn1000, 2018. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html.

[54] Xilinx. SDNet compiler, 2019. https://www.xilinx.com/sdnet.

[55] 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*, pages 25–33, 2019.

A Appendix

We provide additional details and test results about the tested machine learning (ML) models, and about the implementation of the BNN executors.

A.1 Input Features

Table 2 reports the set of features used by the IoT Traffic Classification use case to perform the classification, while Table 3 reports the ones used by the Security Anomaly Detection use case. It should be noted that all the features used by the former use case are also used by the latter. However, some of the features shared by the two use cases differ in the number of bits used for their binary encoding. For example, feature *dur* in the IoT use case requires twice the number of bits with respect to the Security use case.

For the Security use case, we tested the classification performance of the ML models with and without the host-based features. In fact, these features complicate significantly the feature extraction process on the NIC. As we will see in the next subsections, this impacts significantly the classification performance of Decision Trees, while it has minimal impact on BNNs. We speculate that this is the case since not only BNNs perform classification using all the available features, but they also naturally build intermediate features (i.e., feature engineering) in their hidden layers; whereas DTs and RFs use only a subset of the provided features. This observation suggests that there maybe more advantages in using BNNs, beyond those reported in the paper. E.g., BNNs may enable to perform inference using a set of features that are cheaper to collect. However, we leave more investigation into this for future studies, and therefore we only report that this is indeed the case for the Security use case.

Features number vs Memory requirements. Another aspect we did not discuss in the paper is the memory requirement associated with the features. This is usually a bigger issues in switching devices that deal with larger amounts of traffic, such as network switches and routers, while it is not a hard constraint in NICs that are provided with larger (per-flow) memories. In the use cases analyzed in the paper, we use a 256b feature vector, i.e., each flow entry has a memory occupation of 45B (13B for the flowkey, and 32B for the features). That is, a features table for e.g., 10K active flows needs less than 0.5MB of (SRAM) memory.

Feature extraction additional counters. Table 4 lists additional counters that are needed for the feature extraction. Indeed, in order to calculate duration and average input features, five per flow counters have to be stored. Timestamps of the flow start and the last packet sent by the source/destination are used to calculate: the duration of the flow (dur Table 2, 3), the average load (sload, dload), the interarrival times (sinpkt and dinpkt) and TCP connection setup time (ackdat, synack).

Feature	Description	Bin. enc. length
dur	record total duration	16
proto	transaction protocol	8
sbytes	src -> dst transaction bytes	24
bytes	dst -> src transaction bytes	24
sttl	src -> dst TTL value	8
dttl	dst -> src TTL value	8
sload	source bits per second	24
dload	destination bits per second	24
spkts	src -> dst packet count	16
dpkts	dst -> src packet count	16
smean	Mean of the flow packet size tx by the src	16
dmean	Mean of the flow packet size tx by the dst	16
sinpkt	source interpacket arrival time	16
dinpkt	destination interpacket arrival time	16
tcprtt	TCP connection setup round-trip time the sum, of 'synack' and 'ackdat'.	8
synack	TCP connection setup time, the time between, the SYN and the SYN_ACK packets	8
ackdat	TCP connection setup time, the time between the SYN_ACK and the ACK packets.	8

Table 2: IoT Traffic Classification input features

While the source/destination total packet counters are used only to calculate the mean flow size.

A.2 Machine Learning Models

A.2.1 Additional evaluation metrics

This section provides supplementary evaluation results to complement the F1-score and False Positive Rate (FPR) metrics presented in Section 3 of the paper. TP, TN, FP and FN indicate the True Positives, True Negatives, False Positives, and False Negatives, respectively. We report here the following metrics:

- Accuracy: computed as $(TP+TN)/(TP+TN+FP+FN)$, it quantifies the percentage of correct predictions.
- Precision (P): computed as $TP/(TP+FP)$, it quantifies the quota of positive class predictions that actually belong to the positive class.
- Recall (R) or True Positive Rate (TPR): computed as $TP/(TP+FN)$, it quantifies the quota of positive samples that are correctly predicted as positive.
- F1-score: computed as $2TP/(2TP+FP+FN)$, it is the harmonic mean of Precision and Recall.
- False Positive Rate (FPR): computed as $FP/(FP+TN)$, it quantifies the quota of negative samples that are wrongly predicted as positive.
- False Negative Rate (FNR): computed as $FN/(FN+TP)$, it quantifies the quota of positive samples that are wrongly predicted as negative.
- ROC-AUC: the Receiver Operating Characteristic (ROC) curve captures the TPR-FPR tradeoff at different classification thresholds. ROC-AUC is the area under the ROC curve and provides an aggregate measure to quantify the performance of a classification model across all the classification thresholds.

Feature	Description	Bin. enc. length
dur	record total duration	8
proto	transaction protocol	8
sbytes	src -> dst transaction bytes	16
bytes	dst -> src transaction bytes	16
sttl	src -> dst TTL value	8
dttl	dst -> src TTL value	8
sload	source bits per second	24
dload	destination bits per second	24
spkts	src -> dst packet count	16
dpkts	dst -> src packet count	16
smean	Mean of the flow packet size tx by the src	16
dmean	Mean of the flow packet size tx by the dst	16
sinpkt	source interpacket arrival time	16
dinpkt	destination interpacket arrival time	16
tcprtt	TCP connection setup round-trip time, the sum of 'synack' and 'ackdat'.	8
synack	TCP connection setup time, the time between the SYN and the SYN_ACK packets	8
ackdat	TCP connection setup time, the time between the SYN_ACK and the ACK packets	8
Host-based features		
ct_src_ltm	No. of connections of the same dst address in 100 connections according to the last time	8
ct_dst_ltm	No. of connections of the same src address in 100 connections according to the last time	8
ct_ds_src_ltm	No of connections of the same src/dst address in 100 connections according to the last time	8

Table 3: Security Anomaly Detection input features

Counter	Description
flow start	flow start timestamp
dst pkt count	Total number of packets sent by dst
src pkt count	Total number of packets sent by src
dts last pkt ts	Timestamp of the last pkt sent by dst
src last pkt ts	Timestamp of the last pkt sent by src

Table 4: Feature extraction additional counters

Here, we notice that the False Negative Rate (FNR) is not reported in the results, since it is computed as $FNR = 1 - Recall$, and we already report Recall for all the cases.

For each metric we report the average and standard deviation resulting from a 5-fold cross-validation. In the IoT case we are dealing with a 10-classes classification problem, thus, we used a one-vs-rest strategy to evaluate the False Positive and True Positive Rates. Following the description of Section 3, we focused on 3 representative configurations for each type of model. Specifically, for the Decision Tree (DT) and Random Forest (RF) models we considered tree depths values of 3, 6, and 9, and always 5 trees for the RF. The BNN models use a Multi-layer Perceptron architecture, with 256 input binary features and three fully-connected layers. The three models differ by the number of neurons in the hidden layers: [32, 16, n]; [64; 32; n]; [128, 64, n] where $n = 10$ for the IoT use case and $n = 2$ for the Security use case.

We also include two additional columns (TCAM and SRAM) reporting the memory consumption for the TCAM-based and SRAM-based implementations. In the case of BNNs, there is only an SRAM-based implementation, as reported in the paper in Section 3.2.

The results for the IoT use cases are reported in Table 5, while Table 6 reports the results for the Security use case.

A.2.2 Security Anomaly Detection without host features

For the Security use case, as mentioned earlier, we also run the classifier tests to check that the implications of removing the three additional non-flow level features is minimal for the BNN accuracy: the three BNN models described in the paper (32,16,2; 64,32,2; 128,64,2) achieve accuracy [0.9114, 0.9162, 0.9198] when including the 3 extra features, and [0.9106, 0.9164, 0.9201] when not including them (a difference of at most 0.1% point). The results for Decision Trees and Random Forests are instead more impacted, as shown in Table 7.

A.2.3 Confusion Matrices

Figures 19 and 20 report the confusion matrices for the IoT Traffic Classification and Security Anomaly Detection (with all features) use cases, respectively. The matrices have been normalized by dividing the counts by the sum of each row. For each use case we selected a single fold for each of the 9 representative models. In the 3x3 grid, each row contains a different type of model, i.e. Decision Trees (DT), Random Forests (RF) and Binary Neural Networks (BNN). For a given row, different columns contain an increasingly more complex model of a same type, e.g. a more deep tree-based model or a MLP with a larger number of neurons in the hidden layers.

The confusion matrices in the IoT use case confirm that small DTs and RFs fail to properly classify samples belonging to some classes. This is also a byproduct of using binary-decision trees, which fail to identify all of the 10 classes when so shallow. Performance improves as the complexity of the model increases. BNNs are instead able to classify almost all the classes even in the smallest configuration.

A.3 In-NIC Feature Extraction

As mentioned in the Section 6 of the paper, we implement two different features extraction strategies in both the Netronome NIC and NetFPGA. We give more details about these implementations in this subsection.

In both cases, we leverage the modern NIC's ability to host a large number of flow entries (several 10ks) in memory. For instance, both the Netronome and the NetFPGA are also equipped with relatively large DRAMs that can be leveraged to host very large flow tables.

A.3.1 Feature Extraction without connection tracking

The simpler feature extraction strategy keeps a hashtable with the active flows, and performs the following operations, on packet reception: (i) packet parsing to extract the needed

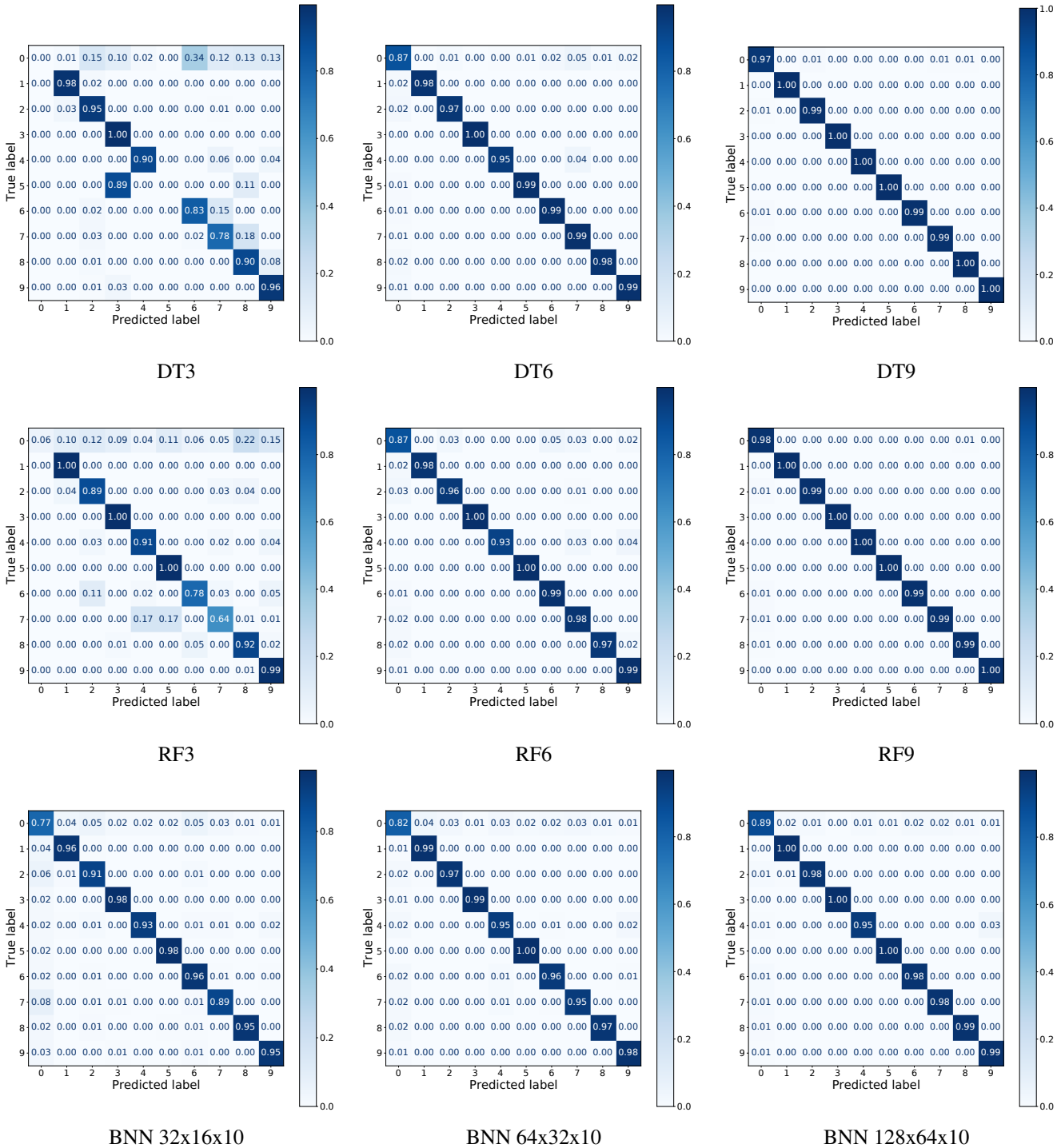

Figure 19: Confusion Matrices for the IoT use case

information, including the 5-tuple used as lookup key; (ii) lookup in the hashtable to retrieve the corresponding flow counters; (iii) update of the values to account for the new packet reception.

To keep the implementation as simple as possible, we do not perform any TCP connection tracking for TCP flows. To measure the flow features that we need for traffic analysis, in fact, in this simpler implementation it is enough to track the initial TCP handshake (e.g., to extract SYN-ACK RTTs). To measure flow duration, instead, we store the timestamp of the first packet of a flow (recognized by the absence of a flow entry in the flow hashtable) and check the timestamp of the

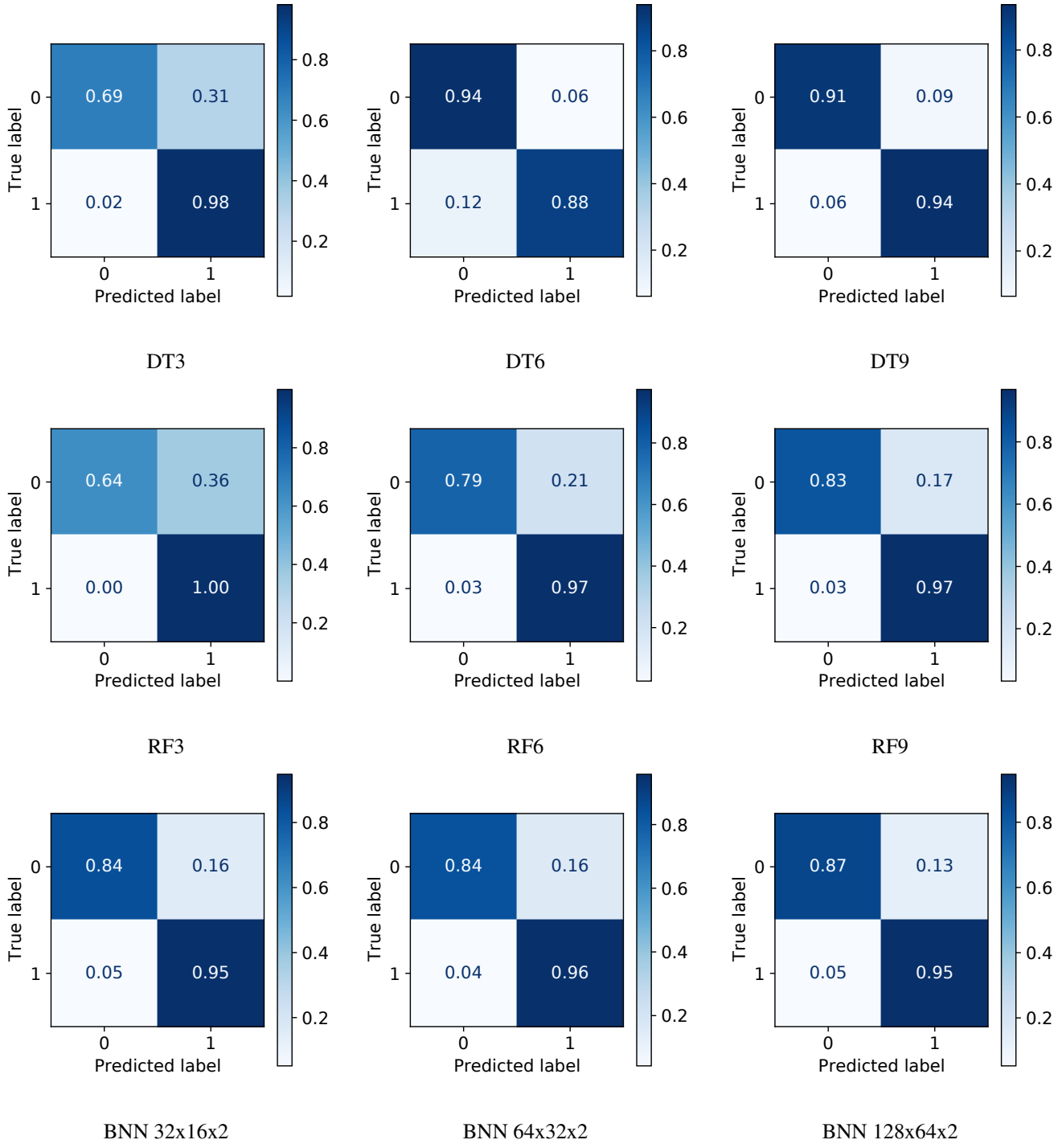

Figure 20: Confusion Matrices for the Security Anomaly Detection use case

last received flow's packet.

Flow entries are removed from the hashtable if no packets match them for a given amount of time. This is implemented as a lightweight task that can be performed lazily. For instance, when a new packet is received, if there is already an entry for the corresponding flow, but `current time - last packet timestamp > timeout value`, then the existing entry is discarded and the flow is considered as a new flow, and the received packet as the first packet of this flow. The timeout value should be configured depending on the deployment environment, taking into account the properties of the monitored traffic. For instance, in telecom operators

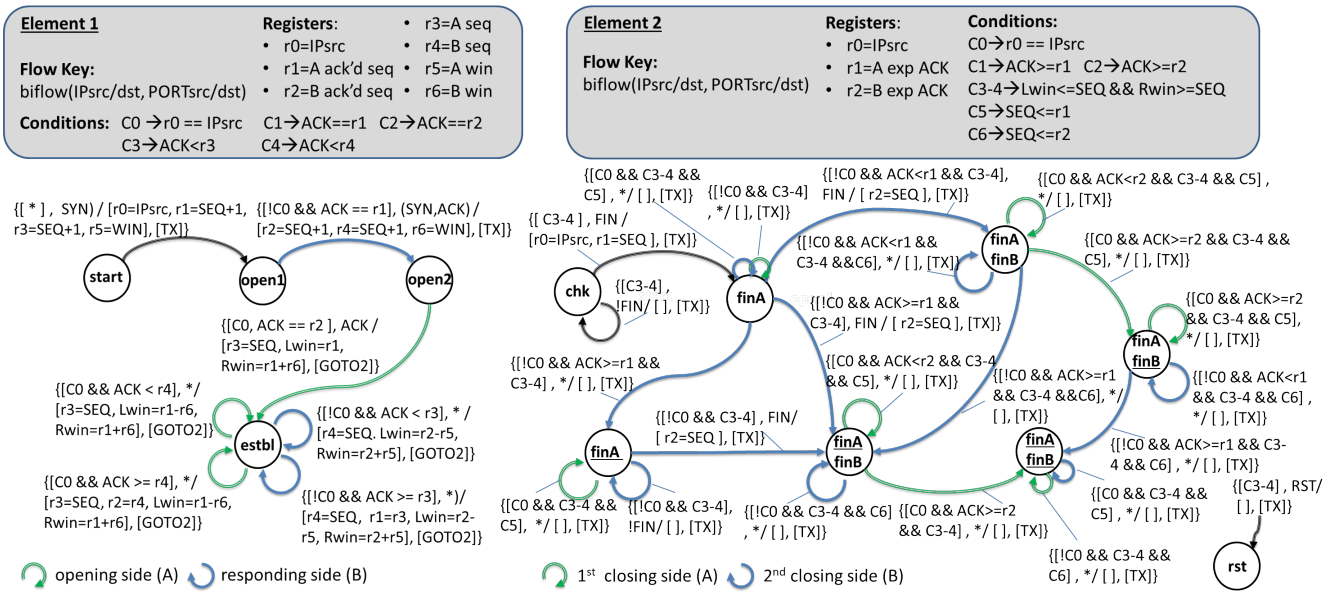

Figure 21: TCP Connection Tracking state machine, reported from [36]

networks that deploy Carrier-grade NATs, the timeout value can be set strictly smaller than the CG-NAT (address,port) re-use timeout, to avoid potential flow entries collision issues.

For the NFP, we implemented this functionality as part of our micro-C programs. For FPGA NICs, this is a feature usually provided by the device vendor, i.e., collecting a small set of flow statistics is usually a built-in function of the provided FPGA firmware. For our NetFPGA implementation, we implemented this basic feature ourselves, using Verilog.

A.3.2 Feature Extraction with connection tracking

The simpler implementation presented earlier is not safe in presence of misbehaving packets. For instance, an attacker may forge packets to impact the measured flow's features. This is possible, since the flow counters are only retrieved using the packet's 5-tuple, which in a general case may be e.g., forged. To avoid this class of issues, for TCP flows it is possible to perform TCP Connection Tracking. Connection tracking verifies that the flow's behavior is consistent with the TCP's state machine, and it includes fine granular per-packet checks, e.g., reading sequence and ack numbers.

We implement TCP connection tracking using the implementation presented in FlowBlaze [36], and the state machine is reported in Figure 21. Here, it should be noted that the state machine is in fact a sequential composition of two state machines. This is a by-product of using the FlowBlaze abstraction, which implements state machines in a sequence of stages that resemble a match-action pipeline similar to that of devices supporting the P4 language.

The two state machines are always executed in sequence, for each packet of an established connection. However, it is possible to identify different responsibilities of each of the two state machines. The first one tracks connection establishment, and computes the allowed sequence numbers (e.g., computing Rwin and Lwin). These values are *forwarded* to the second state machine that performs the actual checks, and which also implements the transitions to check the connection termination.

We implemented this connection tracking solution both in the NetFPGA and in the Netronome NIC. For the NetFPGA, we add two FlowBlaze stages in front of the `N3IC` design. These two stages are used to then to implement the state machine of Figure 21. For the Netronome, we implement the state machine of Figure 21 using micro-C, and extending the `N3IC`'s Netronome firmware.

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	73.1 ± 0.1	61.0 ± 0.0	73.1 ± 0.1	26.9 ± 0.1	3.0 ± 0.0	65.8 ± 0.1	85.1 ± 0.0	119 B	40.2 kB
DT(6)	97.0 ± 0.1	97.0 ± 0.0	97.0 ± 0.1	3.0 ± 0.1	0.3 ± 0.0	97.0 ± 0.1	98.3 ± 0.0	1.3 kB	161.9 kB
DT(9)	99.4 ± 0.0	99.4 ± 0.0	99.4 ± 0.0	0.6 ± 0.0	0.1 ± 0.0	99.4 ± 0.0	99.7 ± 0.0	7.2 kB	170.2 kB
RF(3,5)	81.5 ± 0.3	83.4 ± 0.2	81.5 ± 0.3	18.5 ± 0.3	2.1 ± 0.0	77.5 ± 0.6	89.7 ± 0.2	595 B	200.8 kB
RF(6,5)	96.9 ± 0.2	97.0 ± 0.1	96.9 ± 0.2	3.1 ± 0.2	0.3 ± 0.0	96.9 ± 0.2	98.3 ± 0.1	6.4 kB	809.3 kB
RF(9,5)	99.4 ± 0.1	99.4 ± 0.1	99.4 ± 0.1	0.6 ± 0.1	0.1 ± 0.0	99.4 ± 0.1	99.7 ± 0.0	35.9 kB	851.0 kB
BNN [32,16,10]	92.4 ± 0.2	92.4 ± 0.3	92.4 ± 0.2	7.6 ± 0.2	0.8 ± 0.0	92.4 ± 0.2	95.8 ± 0.1	-	1.2 kB
BNN [64,32,10]	96.0 ± 0.1	96.0 ± 0.1	96.0 ± 0.1	4.0 ± 0.1	0.4 ± 0.0	96.0 ± 0.1	97.8 ± 0.1	-	2.5 kB
BNN [128,64,10]	97.4 ± 0.2	97.5 ± 0.2	97.4 ± 0.2	2.6 ± 0.2	0.3 ± 0.0	97.4 ± 0.2	98.6 ± 0.1	-	5.5 kB

Table 5: IoT dataset

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	88.0 ± 0.2	85.3 ± 0.2	98.2 ± 0.0	1.8 ± 0.0	30.1 ± 0.4	86.0 ± 0.2	84.1 ± 0.2	102 B	173.3 kB
DT(6)	90.3 ± 0.1	96.3 ± 0.1	88.2 ± 0.2	11.8 ± 0.2	5.9 ± 0.1	89.8 ± 0.1	91.1 ± 0.1	677 B	18.9 MB
DT(9)	92.5 ± 0.2	95.0 ± 0.1	93.2 ± 0.2	6.8 ± 0.2	8.7 ± 0.3	91.9 ± 0.2	92.2 ± 0.2	3.4 kB	19.9 MB
RF(3,5)	87.3 ± 0.2	83.4 ± 0.3	99.9 ± 0.0	0.1 ± 0.0	35.2 ± 0.6	84.8 ± 0.3	82.4 ± 0.3	512 B	866.4 kB
RF(6,5)	90.5 ± 0.5	88.6 ± 1.3	97.7 ± 0.9	2.3 ± 0.9	22.2 ± 2.9	89.2 ± 0.7	87.7 ± 1.0	3.4 kB	94.7 MB
RF(9,5)	92.3 ± 0.3	92.7 ± 1.3	95.5 ± 1.1	4.5 ± 1.1	13.3 ± 2.7	91.6 ± 0.4	91.1 ± 0.8	16.9 kB	99.3 MB
BNN [32,16,2]	91.1 ± 0.1	91.4 ± 0.6	95.1 ± 0.6	4.9 ± 0.6	15.9 ± 1.2	90.2 ± 0.2	89.6 ± 0.4	-	1.2 kB
BNN [64,32,2]	91.6 ± 0.1	92.4 ± 0.6	94.7 ± 0.6	5.3 ± 0.6	13.8 ± 1.2	90.8 ± 0.1	90.4 ± 0.3	-	2.5 kB
BNN [128,64,2]	92.0 ± 0.1	92.8 ± 0.4	94.8 ± 0.4	5.2 ± 0.4	13.0 ± 0.8	91.2 ± 0.1	90.9 ± 0.2	-	5.4 kB

Table 6: Security dataset

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	88.0 ± 0.2	85.3 ± 0.2	98.2 ± 0.0	1.8 ± 0.0	30.1 ± 0.4	86.0 ± 0.2	84.1 ± 0.2	102 B	173.3 kB
DT(6)	89.9 ± 0.5	87.6 ± 1.4	98.2 ± 1.2	1.8 ± 1.2	24.7 ± 3.5	88.5 ± 0.7	86.8 ± 1.1	677 B	18.9 MB
DT(9)	91.2 ± 0.1	90.6 ± 0.5	96.2 ± 0.8	3.8 ± 0.8	17.7 ± 1.1	90.2 ± 0.1	89.2 ± 0.2	3.4 kB	19.9 MB
RF(3,5)	87.3 ± 0.2	83.4 ± 0.2	100.0 ± 0.0	0.0 ± 0.0	35.2 ± 0.6	84.8 ± 0.3	82.4 ± 0.3	512 B	866.4 kB
RF(6,5)	89.6 ± 0.4	86.7 ± 0.8	98.9 ± 0.8	1.1 ± 0.8	26.9 ± 2.2	88.0 ± 0.5	86.0 ± 0.7	3.4 kB	94.7 MB
RF(9,5)	91.4 ± 0.3	90.3 ± 0.7	97.0 ± 0.6	3.0 ± 0.6	18.5 ± 1.5	90.4 ± 0.4	89.3 ± 0.5	16.9 kB	99.3 MB
BNN [32,16,2]	91.1 ± 0.2	91.3 ± 0.6	95.1 ± 0.7	4.9 ± 0.7	16.1 ± 1.3	90.1 ± 0.2	89.5 ± 0.3	-	1.2 kB
BNN [64,32,2]	91.6 ± 0.1	92.7 ± 0.2	94.4 ± 0.3	5.6 ± 0.3	13.3 ± 0.5	90.9 ± 0.1	90.6 ± 0.1	-	2.5 kB
BNN [128,64,2]	92.0 ± 0.2	93.0 ± 0.5	94.6 ± 0.4	5.4 ± 0.4	12.6 ± 0.9	91.3 ± 0.2	91.0 ± 0.3	-	5.4 kB

Table 7: Security dataset when not including the three host features