Towards real-time deep learning-based network intrusion detection on FPGA

Le Jeune, L.; Goedemé, T.; Mentens, N.; Sako, K.; Tippenhauer, T.O.

Towards Real-Time Deep Learning-Based Network Intrusion Detection on FPGA

Laurens Le Jeune[1,2](✉), Toon Goedemé[2], and Nele Mentens[1,3]

[1] ES&S - imec-COSIC, ESAT, KU Leuven, Leuven, Belgium
{laurens.lejeune,nele.mentens}@kuleuven.be
[2] EAVISE - PSI, ESAT, KU Leuven, Leuven, Belgium
toon.goedeme@kuleuven.be
[3] LIACS, Leiden University, Leiden, The Netherlands

Abstract. Traditionally, network intrusion detection systems identify attacks based on signatures, rules, events or anomaly detection. More and more research investigates the application of deep learning techniques for this purpose. Deep learning significantly increases detection performance, and can abolish the need for expert knowledge-intensive feature extraction. The use of deep learning for network intrusion detection also has a major disadvantage, however, as it is not deployed yet in real-time implementations. In this paper, we propose two approaches that facilitate the transition towards functional real-time implementations: (1) the use of flow buckets to collect raw traffic-based features, and (2) the acceleration of neural network architectures for intrusion detection using the Xilinx FINN toolchain for FPGAs. We obtain promising results that show our flow bucket approach does not deteriorate detection performance when compared to traditional approaches, and we lay a foundation to further build on with respect to accelerating deep learning algorithms for network intrusion detection on FPGA.

Keywords: Network intrusion detection · Deep Learning · FPGA

1 Introduction

Ever since the introduction of AlexNet [13] in 2012, deep learning has gained increasing attention as its results significantly exceed those of traditional approaches. Besides applications in image processing and natural language processing, a wide range of other domains have since started employing deep learning as well. Also Network Intrusion Detection Systems (NIDSs), which up until that point were based on signatures or rules [22,23,25] as well as anomaly detection and other traditional machine learning techniques [10,28,32], have begun considering deep learning as a reliable approach for intrusion detection [2,33]. NIDSs aim at detecting intrusions in a network environment by inspecting incoming network traffic and either recognizing known attacks or observing deviations from what is considered to be normal traffic. And while (deep) learning-based

J. Zhou et al. (Eds.): ACNS 2021 Workshops, LNCS 12809, pp. 133–150, 2021.
https://doi.org/10.1007/978-3-030-81645-2_9

NIDSs have certainly improved over time, there still remain challenges. One major challenge is to be able to execute these new deep learning-based detection algorithms in real-time in network environments with ever increasing bandwidths. Most NIDSs in literature are currently CPU- or GPU-based and do therefore not have the capabilities of handling large network streams at a high throughput. Moreover, the features in publicly available datasets are typically pre-calculated offline, and it is not trivial to extract them online in real-time. In this paper, we examine these challenges, and aim to develop a deep learning-based NIDS with real-time throughput and feature extraction as well as high detection performance. Our contributions are the following:

- We propose an approach using *flow buckets* for real-time extraction of raw traffic-based features (Sect. 3);
- We examine the flow distribution of the CICIDS2017 and UNSW-NB15 datasets when using that approach (Sect. 5);
- We propose a method to relate model and network throughput for those flow bucket features (Sect. 4);
- We evaluate the performance of the resulting features in a deep learning model and compare to a baseline (Sect. 5);
- We train various quantized deep learning models to examine the effects on accuracy, and then deploy such a quantized model on a PYNQ-Z2 FPGA board using FINN [30] to examine the throughput (Sect. 5);

Before explaining these contributions, Sect. 2 gives an overview of related work.

2 Related Work

While the hardware acceleration of machine learning and deep learning algorithms is certainly increasingly relevant, the reported designs for network intrusion detection systems remain limited in number. This section considers work that has been done regarding the acceleration of machine learning algorithms on FPGAs, and provides insights regarding their performance and model sizes.

Das et al. [8] propose both a Feature Extraction Module (FEM) and a Principal Component Analysis (PCA) based anomaly detector. The FEM is able to collect connection-based as well as time-based features using feature sketches, with a reported throughput of 21.25 Gbps. Likewise, while the PCA achieves 99% attack detection with a 1.95% false alarm rate for the KDDCup1999 dataset, it also supports a 23.76 Gbps data throughput.

Ngo et al. [20] devise both a decision tree as well as a neural network trained on 6 features from the NSL-KDD dataset. The neural network consists of an input layer with the 6 input values, 2 hidden layers with 2 neurons and finally an output layer with 1 neuron. Both classifiers are implemented on a Xilinx Virtex-5 xc5vtx240t FPGA for a 100 MHz clock. They report a maximum throughput of 9.86 Gbps for both classifiers, but only for packets that contain 1500 bytes. The decision tree throughput drops to 7.42 Gbps for 64-byte packets, with the neural network only retaining 1.78 Gbps. While dropping some accuracy percentiles when compared to the same models on GPU, these hardware implementations feature a speedup of

11.6×. Expanding on this work in [19], the same authors provide a neural network with one hidden layer and 0–10 neurons in that layer as a building block for software defined network security. For this application, they report a maximum throughput of 5.12 Gbps if the hidden layer in the neural network is removed.

Ioannou et al. [11] train a three layer fully connected neural network with one hidden layer with 21 hidden neurons on the NSL-KDD dataset and accelerate it using a Xilinx Zynq Z-7020 FPGA. The authors indicate that the architecture supports over 10 Gbps, while they report an accuracy of 80.52%.

Murovič et al. [18] use a binarized neural network with 1 layer of 100 neurons to detect intrusions of the UNSW-NB15 dataset. They report an accuracy of 90.74%, with a latency of 19.62 ns.

Umuroglu et al. [29] propose *LogicNets* to map a quantized neural network to hardware building blocks that can be very efficiently accelerated on hardware. By representing neurons in a network as truth tables with a specific number of input and output bits, they are able to create scalable designs that allow for very high clock frequencies while maintaining a good performance. They report a supported clock frequency of 471 MHz with an accuracy of 91.30% for an architecture trained on the UNSW-NB15 dataset with 5 layers and $593 + 256 + 128 + 128$ neurons.

Maciel et al. [5] implement a reconfigurable architecture for K-means and K-modes intrusion detection on FPGA. They investigate the number of clock cycles and energy consumption for 5 iterations of the algorithm, when using a specific number of centroids and points. When compared to an Intel Xeon E5-2060, they require 91% fewer clock cycles and consume 99% less energy.

Even though these works are significant steps in the right direction, there is still a lot of room for improvement. While the proposed models vary in size, they are still quite small overall. As the 5-layer model of [29] appears to be the largest, real deep learning architectures are not available just yet. Furthermore, while the machine learning models are trained on (a subset of) pre-calculated features, the extraction of those features remains limited. Feature extraction modules serve to extract features to some extent, but do not support all features that are available in publicly available datasets. Finally, reporting the detection performance often remains limited to reporting an accuracy value, while previous work indicates this is not a good representation due to prevalent class imbalance [14]. This paper investigates both the application of deep learning for network intrusion detection, as well as the real-time extraction of generic features that are dataset independent.

3 Feature Extraction

Before any machine learning can be done, it is important to decide on what features to use. These features should represent all information that is required to allow the machine learning algorithm to make reliable predictions. Their representation can also determine what operations are required in the machine learning algorithm: Multidimensional matrices may require convolutions, while sequences might rely on recurrent structures. In this section, we will first discuss how and what features are traditionally used for network intrusion detection, after which we will discuss an alternative: raw traffic based features.

3.1 Traditional Features

Traditionally, network intrusion detection features are dataset dependent: They have been extracted during the inception or the analysis of a particular dataset. Some features can recur among multiple datasets, while others only concern one dataset. For example, both UNSW-NB15 [17] and CICIDS2017 [24] contain the source and destination IP (Internet Protocol) addresses, the TCP (Transmission Control Protocol)/UDP (User Datagram Protocol) ports and the traffic protocol. However, only UNSW-NB15 considers source and destination jitter, while CICIDS2017 includes the 8 TCP flags, among other things.

Typically, features represented in such a manner are numeric and/or symbolic and require some preprocessing before usage. The 41 features of KDDCup1999 [1] and NSL-KDD [26] contain 38 numeric values and 3 symbolic features, which are often represented with a vector containing 122 values[1] [4,7,9,16]. Of course, these features can also be used in other ways, for example by encoding them to pixel values [12] or reducing them in auto-encoders [3].

Although popular, these traditional features do not lend themselves easily for hardware acceleration for two reasons:

1. Extracting traditional features in real-time on hardware is not trivial, as many features require high-level interpretation and/or rely on network flow statistics. High-level features require additional design complexity and resource usage to be functional: Counting the number of HTTPS methods in UNSW-NB15 dataset for the *ct_flw_http_mthd* feature requires parsing incoming traffic to identify HTTPS traffic. Computing the standard deviation of the packet length in a network flow in CICIDS2017 requires keeping track of a flow until it is considered to be finished. This ties up resources and can result in significant detection delays for large flows. Feature extraction modules on FPGA, such as in [8] or [27], do not keep track of all features provided in different datasets, but rather concern smaller feature sets.
2. A machine learning model that is trained on one dataset cannot simply be used for a different dataset: Different (numbers of) features in a potentially different context require a new model for almost every dataset. This is not practical when designing models for real applications as opposed to synthetic datasets, as a model should be retrainable for use in various contexts.

Moreover, an additional argument against these traditional pre-calculated features, from a machine learning point of view, is that there is no guarantee that the provided features suffice to reliably model the circumstances, and that they do not contain unnecessary redundancy. In this paper we therefore investigate an alternative to traditional network intrusion detection: raw traffic-based features.

[1] The three features are one-hot encoded to 3, 11 and 70 values respectively.

3.2 Raw Traffic-Based Features

Raw traffic-based features, in contrast to traditional features, are based only on the raw network traffic that is being inspected by the NIDS. They can be extracted from this traffic by selecting a number of transmitted bytes and using those bytes directly as input for the machine learning model in one way or another. This abolishes the extensive resource usage to monitor traffic flows, and can be used in any situation: Independent of the situation, it is possible to train a model on extracted traffic bytes. In [31], two approaches to use raw traffic are investigated: HAST-I (HierArchical Spatial-Temporal features) considers the first i bytes of a traffic flow to create a large rectangle image. HAST-II extracts the first j bytes of the first k packets in a flow to create a sequence of smaller images. These images can then be processed in convolutional layers in a machine learning model. Similarly, in [33], the leading bytes of subsequent packets in a flow are concatenated to obtain a square image. These approaches report promising results, attaining state-of-the-art results on their evaluation datasets. Moreover, in [14], a comparison is made between the state-of-the-art results for traditional features and the use of raw traffic-based features in machine learning. The findings indicate that raw traffic-based features have the potential to match and even surpass traditional approaches. For real-time application however, the techniques considered in [14,31,33] are not sufficient, as they first sort the entire dataset before extracting features from the resulting flows. In a real-time scenario, it is not possible to wait until an entire dataset, or even a subset thereof, is available before features are extracted, as this would occupy too many memory resources and excessively delay detection. In the following section we will therefore present an approach to extract raw traffic-based features in real-time.

3.3 Our Proposed Approach: Flow Buckets

The research in [31] and [33] works on data that have been completely preprocessed before extracting the traffic bytes for machine learning. Concretely, this works on traffic that has been captured and sorted into complete flows. Traffic can be sorted into flows using flow identifiers (FID). A FID is a tuple comprising 5 values: The flow source and destination IP addresses, the source and destination TCP/UDP ports and the protocol number (6 for TCP or 17 for UDP). Raw traffic can be extract from packet capture files (PCAP), and the flows can be derived by using these FID values. Such an approach is however not representative of a real-time scenario with different flows in rapid succession, and where it might be necessary to keep track of various flows simultaneously. Once again, keeping track of too many flows for too long occupies too many memory resources and delays detection, so an alternative real-time solution is necessary. For this purpose, we will discuss *flow buckets* to keep track of l bytes from up to m packets for n separate flows concurrently.

A flow bucket represents a storage element that can store $l \times m$ bytes from one specific flow. Every flow packet adds l bytes to the bucket, causing it to fill up gradually. When the bucket is full, it produces one set of input features for

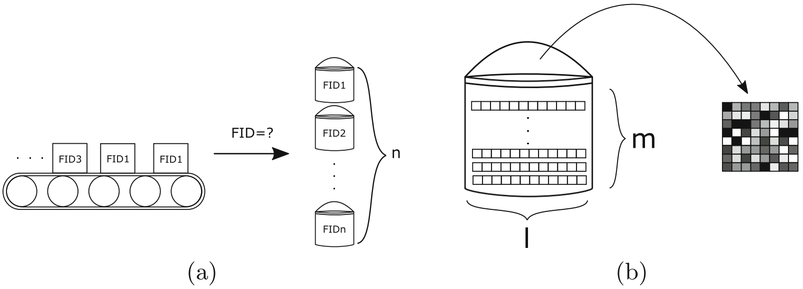


Fig. 1. (a) Packets are put into one of n buckets according to their FID. (b) Each bucket stores up to l packet bytes for up to m packets, and emptying a bucket produces input features for a machine learning model. Missing bytes are zero-padded.

the machine learning model, and it is emptied such that new bytes can be added, as shown in Fig. 1b.

Combining n separate buckets allows for the construction of a mechanism to process a packet stream in real-time, as demonstrated in Algorithm 1 and Fig. 1a. Each bucket can be identified with the FID of the flow it is currently keeping track of, and every flow consists of either a TCP or a UDP session. Whenever adding a packet to the buckets, it is first checked whether one of the current buckets is keeping track of the packet's FID. If this is not the case, either an empty bucket is filled, or the oldest bucket, that is the bucket that has gone the longest without new packets, is emptied to make place for the new FID. On the contrary, if there already is a bucket for the FID, the packet is added to that bucket instead. However, if this would cause the number of packets in the bucket to exceed m, the bucket should also be emptied first. Emptying a bucket simply implies taking l bytes of every packet in a bucket and using the bytes as input features for the machine leaning model. In the case that the number of packets in an emptied bucket is less than m, the missing packet bytes are padded with zeros to ensure the input features retain a fixed shape.

This principle is simpler to translate to hardware in comparison to traditional feature extraction, and by choosing a value for l, m and n, the total resource usage can be constrained. As only very limited processing is actually required, the flow bucket approach is very unlikely to form a bottleneck in any NIDS. We explore this approach further in the experiments in Sect. 5.

When comparing (l, m, n)-flow buckets against other approaches, such as HAST-II [31] or PCCN [33], there are common qualities as well as distinct differences: Flow buckets use a packet's leading bytes as features, and rely on the FID to distinguish between flows. Moreover, the resulting features can be used to construct images to use in a convolutional neural network. The most significant difference however is that unlike HAST-II or PCCN, flow buckets are suitable for extracting incoming network traffic at line speed. As PCCN or HAST-II are devised based on collected packet traces, they do not consider real-time extraction. In principle, the flow bucket approach can be used to extract features with

Algorithm 1: Processing a packet using flow buckets

```
input  : A packet stream $PStream$, where each packet is a sequence of bytes
output: Machine learning feature maps of size $l \times m$

'Buckets' contains all buckets and their current FIDs;
Buckets ← InitializeBuckets($l, m, n$);
for Packet in $PStream$ do
    Extract the FID from the input packet;
    FID ← GetFlowID($P$);
    if IsValidFlowID(FID) then
        if FID is in Buckets then
            AddPacket(Buckets, FID, Packet);
        else
            if IsEmptyBucket(Buckets) then
                AddNewBucket(Buckets, FID);
                AddPacket(Buckets, FID, Packet);
            else
                EmptyOldestBucket(Buckets);
                AddNewBucket(Buckets, FID);
                AddPacket(Buckets, FID, Packet);

Function AddPacket(Buckets, FID, Packet):
    if IsBucketFull(Buckets, FID) then
        EmptyBucket(Buckets, FID);
        AddNewBucket(Buckets, FID);
    FillBucket(Buckets, FID, Packet);
```

the same dimensions as PCCN or HAST-II features, and as such can serve as a tool to make these approaches suitable for real-time use.

3.4 Our Alternative Proposed Approach: Bidirectional Flow Buckets

It is possible to extend the flow bucket concept to bidirectional flows. Given flow F_1 with $FID_1 = (A, B, a, b, x)$ and its reverse flow F_2 with $FID_2 = (B, A, b, a, x)$, the bidirectional flow $F_{1,2}$ includes all packets from both F_1 and F_2. It is bidirectional because it considers all traffic in one connection, both the forward traffic (from host A at port a to host B at port b) and the reverse, backward traffic (from host B at port b to host A at port a). A and B are in this case IP addresses, while a and b are port numbers and x is the protocol number. It is possible to consider bidirectional flows instead of regular flows for the flow bucket approach. This implies, after calculating the FID of an input packet, the algorithm should not only check whether any of the buckets capture that flow, but also whether any of the buckets capture the reverse FID instead. If that is the case, the packet is considered a backward packet in that bidirectional flow, and is added to the bucket. If m and n would be infinitely large, and if for every forward flow its backward alternative would exist, using bidirectional flows would effectively halve the number of employed buckets as opposed to using regular flows. It might therefore be interesting to use bidirectional flows when trying to maximize the number of packets captured inside each bucket. Of course, in a real implementation, it is not feasible to achieve such large values for m or n. Our experiments in Sect. 5.1 show that realistic values already contribute to reducing the number of used buckets.

4 From Software to Hardware

There are various ways to map a software-based machine learning algorithm to an FPGA, such as using HLS4ML[2], Vitis AI[3] or FINN [6,30]. In this paper, we choose FINN for the acceleration of deep learning models, as it is open source and publicly available, and as it supports sufficient layers to synthesize an entire deep learning model. It works using a dataflow architecture, where each layer has its own compute engine and is implemented concurrently. FINN is suitable for high throughput applications, and allows for significant customization through its folding parameters that determine parallelization. The slowest layer will be the bottleneck in a pipelined architecture. In this section, we discuss the steps FINN takes to accelerate hardware, as well as how to interpret the reported results.

4.1 Accelerating a Model Using FINN

FINN requires a number of steps to turn a regular deep learning network into a dataflow architecture. Initially, the model needs to be quantized, for example using Quantization Aware Training (QAT) in Brevitas [21]. The resulting quantized model can then be processed using a set of transformations. First, streamlining transformations serve to remove floating point operations from the model, by shifting them around and aggregating them in multi-thresholding layers. This prepares the other layers to be then turned in HLS (High-Level Synthesis) layers, once all non-supported operations have been (re)moved. FINN can then synthesize these HLS layers into hardware using Xilinx Vivado, and this hardware can finally be deployed on an FPGA. In Sect. 5 we describe how we use FINN to accelerate a custom deep learning model.

4.2 Fps to Bandwidth

In FINN, the default reporting of model throughput uses the framerate r_f in *frames per second* or *fps*, which is common practice for image processing models. While this is a useful metric, it does not directly translate to a supported

[2] https://fastmachinelearning.org/hls4ml/.
[3] https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

maximum network bandwidth. Nevertheless, as each emptied bucket in the flow bucket approach also corresponds with one input image for the subsequent algorithm, in our context, the r_f can also be defined as the number of *flows per second*, where each flow is one emptied bucket. When B is the number of traffic bytes in a dataset, and f is the total number of captured flows using the flow bucket method, $bpf = B \cdot 8/f$ is the average number of *bits per flow* that are transmitted in the dataset. This also includes non-flow bits such as Address Resolution Protocol (ARP) data. Consequently, it is also possible to find the relationship between the supported throughput r_f and the supported bitrate r_b in bits per second: $bpf = r_b/r_f$. This allows a FINN user to estimate what the supported bandwidth r_b would be, for a specific dataset with B, a specific flow bucket configuration with f and an accelerated model with r_f.

5 Experiments

In this section, we will describe the experiments we conducted for both the evaluation of flow buckets and the translation of machine learning models to hardware. All code is publicly available[4].

5.1 Flow Buckets

We have conducted a number of experiments to evaluate the impact of using flow buckets to extract features instead of using a completely preprocessed dataset. We consider buckets with $l = 96$ and $m = 5$, or concretely, buckets that can contain the leading 96 bytes of up to 5 packets with the same FID. Using these settings, one learning sample is a concatenation of 5 vectors of 96 elements, where each vector is separated from the other vectors with a FF-valued byte as described for Header-Payload features in [33]. Both the results in [33] as well as in [14] suggest this configuration is suitable to detect attacks. The resulting vector has a length of 484 bytes and can be reshaped into a 22 × 22 image. First we will consider the impact of m and the number of buckets, n, on the flow distribution, then we will inspect machine learning performance for these alternative features.

Flow Distribution. While maintaining l fixed at 96, different values for m and n will lead to different feature compositions for the machine learning model. First, we inspect the impact of n when m remains constant at a value of 5. We choose $m = 5$ to be able to generate features with the same shape as those in [33]. Increasing n leads to keeping track of more flows concurrently, which should in theory provide buckets with more opportunities to be filled up before being emptied to make space for a different flow. For this experiment, we use various values of n for the CICIDS2017 dataset. Figure 2a gives the flow distribution for this experiment. The flow distribution considers the percentage of all extracted flows (=emptied buckets) that consist of s packets, where s is a value between

[4] https://gitlab.com/EAVISE/real-time-dl-nids-on-fpga.

1 and m. Clearly, nearly 80% of all flows consist of only 1 packet, and nearly the entire other 20% comprise the 2-packet flows. Apparently, while there is space for 5 packets, the vast majority of buckets are already emptied when they contain 1 or 2 packets. One reason for this could be that, for the vast majority of the dataset, only flows with 1 or 2 packets exist. That however is very unlikely, as it would limit any communication to be finished with a maximum of 2 packets transmitted. Moreover, Fig. 2b considers a scenario with an infinitely large value for n in CICIDS2017. This can be achieved by sorting all packets in the dataset according to their FID first, and then by continuing filling buckets until all packets have been used for every FID separately. As there clearly are many flows with at least 5 packets, the results in Fig. 2a suggest that there are so many different concurrent flows in CICIDS2017 that it is nearly impossible to keep track of a flow long enough to consistently fill buckets. If we would keep on increasing n, more and more flows would contain 5 packets. However, as each bucket requires $m \cdot l = 96 \cdot 5 = 480$ bytes of storage, using $n = 10^4$ would already require 4.8 MB of on-chip memory, which is unrealistic for real-time acceleration on FPGA, especially if this memory might also be reserved for the machine learning model.

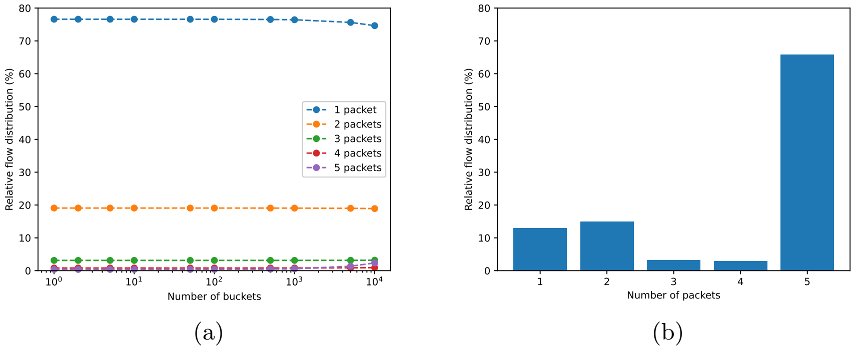
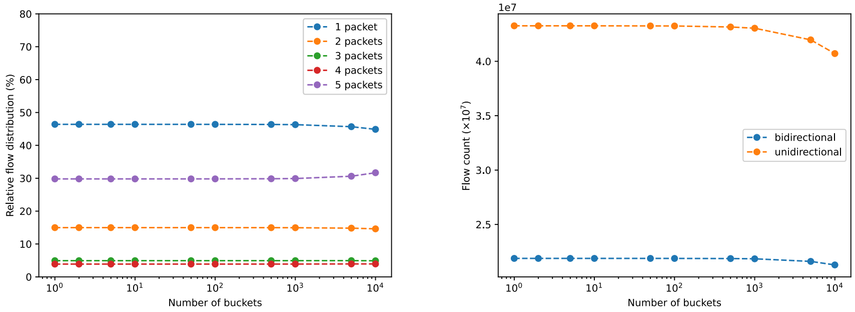


Fig. 2. Flow distribution for unidirectional flow buckets, with $m = 5$ and various values of n (a) and baseline flow distribution (b) for CICIDS2017.

Interestingly, when performing the same experiment with bidirectional flows instead of unidirectional flows, the results are significantly different, as shown in Fig. 3a. While 1-packet and 2-packet flows still encompass about 65% of all flows, about 30% of all flows now comprise 5 packets. This indicates a significant portion of the traffic is bidirectional, and that it is a lot easier to keep track of bidirectional flows compared to unidirectional flows. As a result, the total number of extracted flows is considerably smaller for the bidirectional approach, as shown in Fig. 3b. For this approach more packets can be captured in the same bucket, which in turn results in fewer buckets being needed. In a real-time scenario,



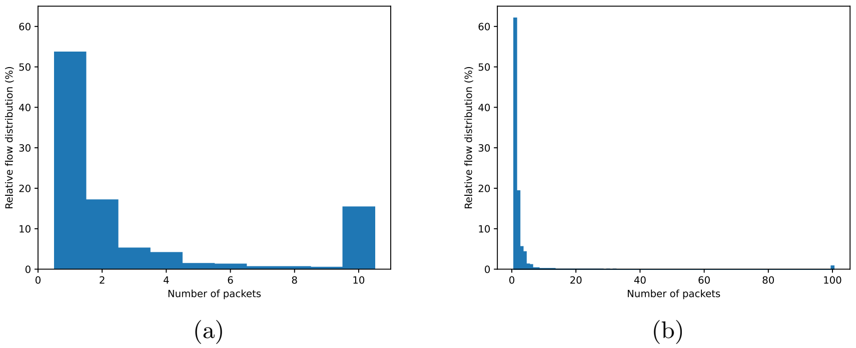
(a) (b)

Fig. 3. Flow distribution for bidirectional flow buckets (a) and flow counts for unidirectional and bidirectional flow buckets (b) with $m = 5$ and various values of n for CICIDS2017.

when compared to the unidirectional alternative, this would imply that for the same network traffic the overall architecture supports a higher throughput. It is also clear that realistic values for m and n already result in considerable improvements, which in turn implies that infinitely large values are not required.

For the bidirectional case, there can be merit in increasing m, as it is likely that deeper buckets will indeed be sufficiently utilized. Each 5-packet flow has the potential to be a larger total flow that is terminated due to the bucket being full. This is not the case for the unidirectional approach, where nearly all flows terminate at only 1 or 2 packets. Figure 4a gives the flow distribution for the bidirectional approach with $m = 10$ and $n = 100$, with Fig. 4b similarly giving the distribution for $m = 100$. As the bucket depth m increases, the portion of m-packet flows decreases, while other portions increase. This trend is most noticeable for the portion of 1-packet streams, that significantly increases from 54% to 62% as m goes from 10 to 100.

When repeating the first experiment for UNSW-NB15, the results in Fig. 5 show a behaviour similar to CICIDS2017. For unidirectional flows, most flows are captured with 2 or 4 packets, with only a few flows consisting of 5 packets. Once again, using bidirectional flows considerably decreases the total number of flows, which goes from about 7.5×10^7 unidirectional to about 5.3×10^7 bidirectional flows. This effect is also visible in Fig. 5b, as a significantly larger portion of the flows contains 5 packets. Both unidirectional and bidirectional cases also more clearly follow the expected behaviour compared to CICIDS2017 when increasing n, as being able to monitor more flows concurrently increases the relative portion of 5-packet flows, starting from about $n = 1000$. One reason for this can be found through the average number of packets per unique flow: CICIDS2017 contains 55,953,889 packets that belong to a flow, while UNSW-NB15 contains 187,072,760 packets. However, when counting the number of unique FIDs per day, and calculating the sum for every day for both datasets, CICIDS2017 has 2,649,163 unique FIDs and UNSW-NB15 has 3,996,364. On average, this means



(a)

(b)

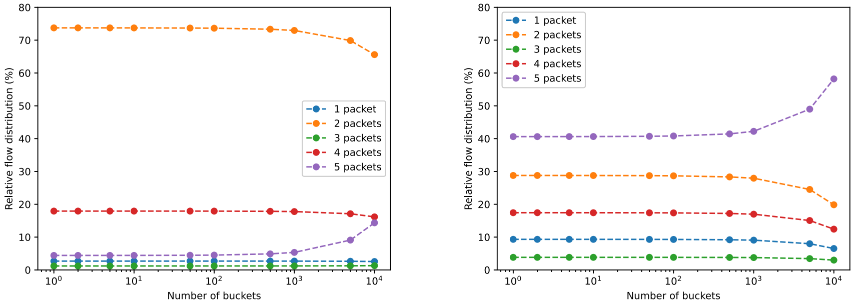
Fig. 4. Flow distribution for $m = 10$ (a) and $m = 100$ (b) bidirectional flow buckets with $n = 100$ for CICIDS2017.

that UNSW-NB15 has about 47 packets for each unique FID, while CICIDS2017 only has about 21. Therefore, on average more packets will belong to the same flow in UNSW-NB15, which in turn means it is easier to start completely filling up buckets while fewer buckets are available.

Machine Learning Performance. In order to measure the impact of using flow buckets, we first establish a baseline performance, obtained from training a machine learning model on raw traffic-based features from a dataset that has been completely sorted into flows. The employed model is a Convolutional Neural Network we name BaseCNN2D as presented in Fig. 6. Each convolutional layer is followed by one batch normalization layer (with $\epsilon = 10^{-5}$) and one ReLU activation layer. The 6×6 feature map is downsampled using max pooling, with the result being reformed to a 1024-sized vector. This vector, after one final batch normalization layer as well as a 50% dropout layer, is used as an input to a fully connected layer.

The model was trained for an initial learning rate of 0.001, which was divided by 10 if no improvement was shown during the last 10 epochs, for a total number of 100 epochs, with an SGD optimizer. We report our results using the accuracy Acc, the weighted average detection rate[5] DR_w, the weighted average F-measure F_w, as well as two metrics that are proposed in [14]: The detection score DS and the identification score IS. The DS is the F-measure of the binarized confusion matrix, where each detected attack that actually was an attack (even if it was another attack class) is considered a true positive. Equation 1 then gives the IS, which is the harmonic mean of the weighted average F-measure F_w and macro average F-measure F_M in a multiclass scenario. As the IS considers both minority and majority classes through respectively F_M and F_w, it assesses how well a model can identify specific attacks in a dataset.

[5] The DR is commonly used in intrusion detection literature to denote the recall.



(a)

(b)

Fig. 5. Flow distribution for unidirectional (a) and bidirectional (b) flow buckets, with $m = 5$ and various values of n for UNSW-NB15.

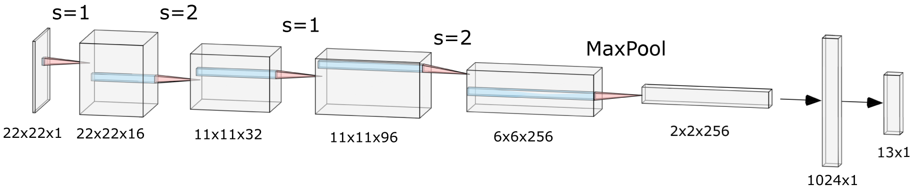


Fig. 6. The BaseCNN2D architecture that was used in the experiments, with s indicating the stride of a convolution layer while the feature map and fully connected layer sizes are provided below the graph. Visualized using [15].

$$IS = \frac{2 \cdot F_w \cdot F_M}{F_w + F_M} \tag{1}$$

For these training parameters and these metrics, the machine learning results are presented in Table 1.

To investigate the effect of using flow bucket feature extraction, we consider a unidirectional bucket configuration with $l = 96$, $m = 5$ and where n is either 1, 10, 100 or 1000. This configuration produces features that are in line with the features in [14,33] and allows for assessing whether flow bucket features actually work for machine learning. As the results in Table 1 show, that is actually the case: For 10, 100 and 1000 buckets the results are only slightly lower than the baseline, without any indications that one is significantly worse than the other. This is to be expected, as Fig. 2a shows very little difference between the flow distributions of the selected values for n. Somewhat unexpectedly, the model for $n = 1$ achieves a near perfect classification, making only 1,263 errors for over 4 million test samples. This further underlines the potential that raw traffic-based features have for real-time deep learning based network intrusion detection.

Table 1. Machine learning performance of the BaseCNN2D model for datasets extracted from CICIDS2017 using flow buckets for different values of n. The first row considers the scenario in which all packets have been sorted before feature extraction.

l	m	n	Acc	DR$_w$	F$_w$	DS	IS
/	/	/	99.54	99.54	99.54	97.70	98.77
96	5	1	99.97	99.97	99.97	99.97	99.28
96	5	10	99.37	99.37	99.40	96.03	98.42
96	5	100	99.38	99.38	99.40	96.05	98.46
96	5	1000	99.37	99.37	99.39	96.02	98.46

5.2 Quantization

Before using FINN for hardware acceleration, it is required to quantize the target machine learning model. Using the Brevitas [21] library for quantization-aware training, we trained three neural networks using either 8-bit, 4-bit or 2-bit quantization of both weights and activations. For this purpose, a simplified version of the BaseCNN2D model was used as presented in Fig. 7, with a smaller fully connected layer, global average pooling instead of max pooling and without regularization after pooling. By training this network without quantization, and by then using the proposed quantization alternatives, we obtain the results presented in Table 2. All training is done using the CICIDS2017 dataset with raw traffic-based features extracted from presorted flows.

Table 2. Result of training the simplified BaseCNN2D model using 8-bit, 4-bit and 2-bit quantization, as well as baseline results without quantization.

Quantization	Acc	DR$_\mathrm{w}$	F$_\mathrm{w}$	DS	IS
None	99.52	99.52	99.52	97.59	98.58
8-bit	99.53	99.53	99.53	97.65	98.65
4-bit	99.49	99.49	99.49	97.46	98.36
2-bit	99.41	99.41	99.41	97.26	97.23

These results suggest that the 8-bit quantization has no negative impact on the model's performance. On the contrary, the 8-bit case appears to achieve better results than the baseline without any quantization. When the quantization is further extended to 4-bit, there are only minor drops in performance. And remarkably, even going down to 2-bit quantization, the results remain high. Because we want to limit the hardware footprint of our FPGA implementation as much as possible, we choose the 2-bit w2a2 model for hardware acceleration.

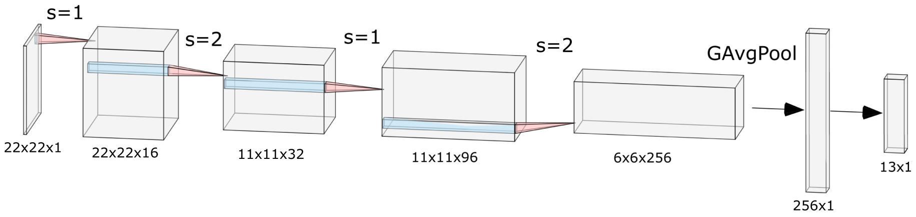


Fig. 7. The simplified BaseCNN2D architecture, used for quantization and hardware acceleration with s indicating the stride of a convolution layer while the feature map and fully connected layer sizes are provided below the graph. Visualized using [15].

Table 3. Resource utilization of the w2a2 quantized simplified BaseCNN2D model as accelerated on a PYNQ-Z2 board.

Resources	Utilization	Available	Utilization (%)
LUT	24,635	53,200	46.31%
LUTRAM	1,980	17,400	11,38%
FF	27,450	106,400	25,80%
BRAM	37.50	140	26.79%

5.3 FPGA Performance

We use the FINN workflow [6,30] to accelerate the trained and quantized model on FPGA. We first use a specific set of transformations in a specific order to turn the entire model into synthesizable hardware. Then, this hardware is synthesized using FINN in collaboration with Vivado 2020.1, resulting in a bitfile and Python code for the PYNQ-Z2 board with a Xilinx ZYNQ XC7Z020-1CLG400C FPGA. Using out-of-context synthesis for the accelerated model leads to the resource utilization depicted in Table 3. The included Python code reports a throughput of about 9635 images/second, or $r_f = 9635$ *fps* using a 100 MHz clock.

As $B = 50,557,729,836$ bytes for the CICIDS2017 dataset, we can calculate *bpf* for any flow bucket approach to estimate the supported bandwidth. For example for $(l, m, n) = (96, 5, 5)$, $f = 43,263,642$ flows, which in turn means *bpf* is about 9349 bits per flow. This means that for this architecture with $(96, 5, 5)$-buckets, the supported bitrate is $r_b = r_f \cdot bpf = 90.08$ Mbps.

The supported throughput for the flow buckets in hardware will depend on the network interface. Consider for example an implementation on a PYNQ-Z2 FPGA with a *Xilinx 1G/2.5G Ethernet Subsystem* and a physical ethernet interface supporting 1Gpbs. While internally the system may run faster, the total throughput will be constrained by the physical interface. For this example, the internal system could run with a 100 MHz clock and receive the ethernet packets in 32-bit blocks over an AXI Stream bus. In the absolute worst case, with the smallest possible 64-byte Ethernet packets, all featuring the same FID, it should take $max(l, 64)/4 \cdot m$ clock cycles to fill a bucket. For $(96, 5, n)$, that would be 120 clock cycles or $1.2 \cdot 10^6$ buckets per second.

5.4 Discussion

We developed, to the best of our knowledge, the first reported sufficiently deep convolutional neural network for network intrusion detection accelerated on an FPGA. It obtains state-of-the-art detection performance on CICIDS2017 using 2-bit quantization and raw traffic-based features, and fits on a lower-end PYNQ-Z2 board. The results in Sect. 5.1 show that these raw traffic-based features can be extracted in real-time using flow buckets without considerably adversely effecting detection performance. There is still room for improvement, as the throughput of the accelerated model should still be increased for real-time intrusion detection. We observe the following options to achieve this goal:

- There are still some computational bottlenecks in the network that can be solved in order to increase the speed. Currently the network is limited by the slowest layer.
- The current platform is relatively small resource-wise. Using a larger platform such as a Xilinx Alveo card would allow for more parallelization, which would in turn allows to speed up the computations in the layers.
- Further optimization of the machine learning model, through 1-bit quantization, pruning or other methods, reduces the functionality that needs to be translated to hardware.

We are confident that these optimizations will further improve the throughput towards real-time intrusion detection.

6 Conclusion and Future Work

In this paper, we set out to develop a deep learning-based NIDS with real-time throughput and feature extraction as well as high detection performance. We proposed the flow bucket approach to enable real-time raw traffic-based feature extraction. Analyzing the results of this approach for two recent and publicly available datasets, UNSW-NB15 and CICIDS2017, shows that using these features does not deteriorate learning performance, but rather allows for very high detection accuracy. Moreover, we also proposed a deep learning architecture that retains performance even when quantized towards 2-bit weights and activations. Finally, we have laid a foundation to further build upon with respect to accelerating our deep learning model in hardware. While the maximum bandwidth is currently still limited, there are clear possibilities to further improve throughput. For future work, we will aim at optimizing the hardware architecture to reach higher network traffic bandwidth. Moreover, we will further explore different ways in which raw-traffic based features can be used as input for machine learning models. E.g., for unidirectional flow buckets with $(l, m, n) = (96, 5, 5)$, the majority of the flows contains 1 or 2 packets. Using $m = 2$ significantly reduces input feature size, and thus model size, but might not impact the detection performance too drastically. Additional research concerning the applicability of bidirectional flows is also in order. Exploring all these options may finally lead to viable deep learning architectures for real-time NIDSs.

Acknowledgements. This work is supported by COllective Research NETworking (CORNET) and funded by VLAIO under grant number HBC.2018.0491. This work is also supported by CyberSecurity Research Flanders with reference number VR20192203.

References

1. KDD Cup 1999 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
2. Abdulhammed, R., Musafer, H., Alessa, A., Faezipour, M., Abuzneid, A.: Features dimensionality reduction approaches for machine learning based network intrusion detection. Electronics **8**, 322 (2019)
3. Al-Qatf, M., Lasheng, Y., Al-Habib, M., Al-Sabahi, K.: Deep learning approach combining sparse autoencoder with SVM for network intrusion detection. IEEE Access **6**, 52843–52856 (2018)
4. Alrawashdeh, K., Purdy, C.: Toward an online anomaly intrusion detection system based on deep learning. In: 2016 15th IEEE ICMLA, pp. 195–200, December 2016
5. Andrade Maciel, L., Alcântara Souza, M., Cota de Freitas, H.: Reconfigurable FPGA-based K-means/K-modes architecture for network intrusion detection. IEEE TCAS-II **67**(8), 1459–1463 (2020)
6. Blott, M., et al.: FINN-R: an end-to-end deep-learning framework for fast exploration of quantized neural networks. ACM TRETS **11**(3), 1–23 (2018)
7. Chuan-long, Y., Yue-fei, Z., Jin-long, F., Xin-zheng, H.: A deep learning approach for intrusion detection using recurrent neural networks. IEEE Access **5**, 21954–21961 (2017)
8. Das, A., Nguyen, D., Zambreno, J., Memik, G., Choudhary, A.: An FPGA-based network intrusion detection architecture. IEEE Trans. Inf. Forensics Secur. **3**(1), 118–132 (2008)
9. Ding, Y., Zhai, Y.: Intrusion detection system for NSL-KDD dataset using convolutional neural networks. In: Proceedings of 2018 CSAI (CSAI 2018), pp. 81–85. ACM, New York (2018)
10. García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., Vázquez, E.: Anomaly-based network intrusion detection: techniques, systems and challenges. Comput. Secur. **28**(1), 18–28 (2009)
11. Ioannou, L., Fahmy, S.A.: Network intrusion detection using neural networks on FPGA SoCs. In: 2019 29th FPL, pp. 232–238, September 2019
12. Kim, T., Suh, S.C., Kim, H., Kim, J., Kim, J.: An encoding technique for CNN-based network anomaly detection. In: IEEE BigData, pp. 2960–2965, December 2018
13. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Adv. NIPS **25**, 1097–1105 (2012)
14. Le Jeune, L., Goedemé, T., Mentens, N.: Machine learning for misuse-based network intrusion detection: overview, unified evaluation and feature choice comparison framework. IEEE Access **9**, 63995–64015 (2021)
15. LeNail, A.: NN-SVG: publication-ready neural network architecture schematics. J. Open Source Softw. **4**(33), 747 (2019). https://doi.org/10.21105/joss.00747
16. Lopez-Martin, M., Carro, B., Sanchez-Esguevillas, A., Lloret, J.: Shallow neural network with kernel approximation for prediction problems in highly demanding data networks. Expert Syst. Appl. **124**, 196–208 (2019)

17. Moustafa, N., Slay, J.: UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In: 2015 MilCIS, pp. 1–6, November 2015
18. Murovič, T., Trost, A.: Massively parallel combinational binary neural networks for edge processing. Elektrotehniski Vestnik/Electrotech. Rev. **86**, 47–53 (2019)
19. Ngo, D.M., Pham-Quoc, C., Thinh, T.N.: Heterogeneous hardware-based network intrusion detection system with multiple approaches for SDN. Mob. Netw. Appl. **25**(3), 1178–1192 (2020)
20. Ngo, D.-M., Tran-Thanh, B., Dang, T., Tran, T., Thinh, T.N., Pham-Quoc, C.: High-throughput machine learning approaches for network attacks detection on FPGA. In: Vinh, P.C., Rakib, A. (eds.) ICCASA/ICTCC -2019. LNICST, vol. 298, pp. 47–60. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-34365-1_5
21. Pappalardo, A.: Xilinx/brevitas. https://doi.org/10.5281/zenodo.3333552
22. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. **31**(23), 2435–2463 (1999)
23. Roesch, M.: Snort - lightweight intrusion detection for networks. In: 13th USENIX Conference on System Administration (LISA 1999), pp. 229–238. USENIX Association, USA (1999)
24. Sharafaldin, I., Habibi Lashkari, A., Ghorbani, A.: Toward generating a new intrusion detection dataset and intrusion traffic characterization. In: 4th ICISSP, Portugal, pp. 108–116 (2018)
25. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: 10th ACM CCS (CCS 2003), pp. 262–271. ACM, New York (2003)
26. Tavallaee, M., Bagheri, E., Lu, W., Ghorbani, A.A.: A detailed analysis of the KDD CUP 99 data set. In: 2009 IEEE CISDA, pp. 1–6, July 2009
27. Tran, C., Vo, T.N., Thinh, T.N.: HA-IDS: a heterogeneous anomaly-based intrusion detection system. In: NAFOSTED NICS, vol. 2017, pp. 156–161 (2017)
28. Tsai, C.F., Hsu, Y.F., Lin, C.Y., Lin, W.Y.: Intrusion detection by machine learning: a review. Expert Syst. Appl. **36**(10), 11994–12000 (2009)
29. Umuroglu, Y., Akhauri, Y., Fraser, N.J., Blott, M.: LogicNets: co-designed neural networks and circuits for extreme-throughput applications. In: FPL, vol. 2020, pp. 291–297 (2020)
30. Umuroglu, Y., et al.: FINN: a framework for fast, scalable binarized neural network inference. In: Proceedings of the 2017 ACM/SIGDA FPGA, pp. 65–74. ACM (2017)
31. Wang, W., et al.: HAST-IDS: learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection. IEEE Access **6**, 1792–1806 (2018)
32. Zhang, J., Zulkernine, M., Haque, A.: Random-forests-based network intrusion detection systems. IEEE Trans. Syst. Man Cybern. Part C (Appl. Rev.) **38**(5), 649–659 (2008)
33. Zhang, Y., Chen, X., Guo, D., Song, M., Teng, Y., Wang, X.: PCCN: parallel cross convolutional neural network for abnormal network traffic flows detection in multi-class imbalanced network traffic flows. IEEE Access **7**, 119904–119916 (2019)