SwitchTree: In-network Computing and Traffic Analyses with Random Forests

Jong-Hyouk Lee, Kamal Singh

Noname manuscript No.
(will be inserted by the editor)

SwitchTree: In-network Computing and Traffic Analyses with Random Forests

Jong-Hyouk Lee · Kamal Singh

the date of receipt and acceptance should be inserted later

Abstract The success of machine learning in different domains is finding applications in networking. However, this also needs real-time analyses of network data which is challenging. The challenge is caused by the big data size and the need for bandwidth to transfer network data to a central location hosting an analyses server. In order to address this challenge, an in-network computing paradigm is gaining popularity with advances in programmable data plane solutions. In this paper, we perform in-network analysis of the network data by exploiting the power of programmable data plane. We propose SwitchTree which embeds the Random Forest algorithm inside a programmable switch such that Random Forest is configurable and re-configurable at runtime. We show how some flow level stateful features can be estimated, such as the round trip time and bitrate of each flow. We evaluate the performance of SwitchTree using system level experiments and network traces. Results show that SwitchTree is able to detect network attacks at line speed with high accuracy.

Keywords In-network computing · Network traffic analyses · Programmable data plane · P4 · Random Forests · Detection of attacks

J.-H. Lee
Department of Computer and Information Security, Sejong University, Republic of Korea
Tel.: +82-3408-1846
E-mail: jonghyouk@sejong.ac.kr

✉ K. Singh (Corresponding Author)
Laboratory Hubert Curien,
University of Saint-Etienne, Jean Monnet,
Saint-Etienne, France
E-mail: kamal.singh@univ-st-etienne.fr

1 Introduction

Demands of analysing network traffic in order to detect anomalies and security attacks are ever growing. For example, for network management, data centers and network service providers require to analyse lot of network traffic at very high speed. Their goals are to improve the performance of their network and detect problems in real-time to be able to solve them.

This topic of data analyses and anomaly detection takes us to the topic of machine learning. Recently, machine learning approaches have been successfully applied in several domains and have shown significant breakthroughs. One advantage of machine learning is that it can deal with complicated problems. Therefore it is intuitive to exploit machine learning to solve the problems in networking where we are faced with complex problems [22].

Exploiting machine learning for networks or making networks more intelligent has been made possible by some recent advances in the domain of networking. For example, there has been the arrival of virtualisation, which has brought flexibility enabling smarter solutions and techniques. There has also been the idea of decoupling control and data planes, which was pushed by Software Defined Networking (SDN). One disadvantage of SDN was that it was limited to existing headers/header fields and there was no support for custom (encapsulating) protocols. This was because only the control plane was programmable. The data plane was not programmable and it could only support the pre-configured headers. Thus, the idea of programmable switches is gaining a lot of attention. The idea is to make switches as protocol-independent and target independent.

Programmable switches call for flexibility, but Application Specific Integrated Circuits (ASICs) present in the switches are traditionally fixed. New custom ASICs can achieve such flexibility required by programmable switches at terabit speeds and for that there are also other technologies like FPGA, etc. Thus, initiatives such as P4, a programming language for the switches, are gaining ground. With such initiatives, now we have the possibility to have both control plane and data plane as programmable. This is one of the factors contributing to the paradigm of in-network computation [19] and we are witnessing an evolution from configurable to programmable network equipment as shown in Fig 1.

With the above context, we study in-network analyses of packets at line-speed by exploiting the power of programmable data plane. This in-network analysis can be used, for example, to detect security attacks in the data plane itself. This can be done without sending all the data for analyses to the control plane. Thus, in-network analyses of data in the data plane economises the link between the switch and the control plane. In-network traffic analyses for attack detection has several advantages. It avoids sending all the heavy network data to a centralised server for analyses. Detection is done in the path itself and in a distributed fashion. When an abnormal flow is detected then it can be dropped, marked, re-classified, forwarded to a different port or forwarded to

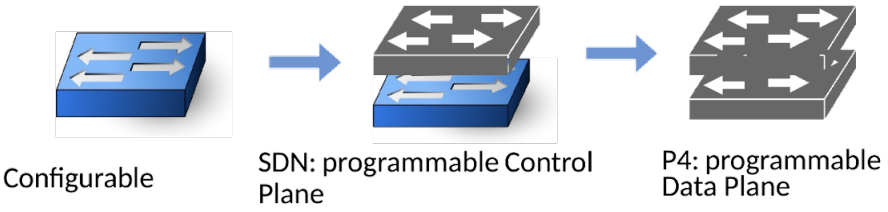

Fig. 1 From configurable to programmable

a different path with degraded QoS, for further analyses, to avoid wasting important resources.

In this paper, we propose to embed Random Forests inside a programmable switch. The key property of the embedded Random Forest is that it is reconfigurable at run-time by the control plane. As a use case of this Random Forest implemented in data plane, we study the detection of network security attacks at line speed. Using system evaluations, we show that it is possible to detect security attacks with high accuracy. The contributions of this paper are summarised as follows:

- We embed the Random Forest algorithm in a programmable switch. The control plane can configure or reconfigure the Random Forest parameters at run-time.
- For analysing incoming packets, we propose strategies to compute different stateless and stateful features inside the data plane which offers only limited operations.
- We use the embedded Random Forest for in-network analyses of incoming packets at line-speed.
- Experiments show that Random Forest embedded in a switch can detect security attacks with high performance.

This paper is organised as follows. Section 2 discusses the background and the related work. Section 3 shows how Random Forests can be used to detect attacks in general. Section 4 shows how we can embed Random Forests in a programmable switch and use it for online detection of attacks, at line speed in the data plane. Section 5 concludes the paper.

2 Background

This section presents the background on programmable data plane as well as related work on in-network computing and detection of security attacks in the network.

2.1 Programmable Data Plane

Traditionally, in networking domain, we have been limited to fixed functionalities. We have a little or no control on allocating switch resources. This is due

to proprietary and closed APIs. This means that we can not add new features which in turn slows down the innovation. However, recently new programmable switch chips are becoming available which let us define and customize how a switch processes packets.

This is a big step from limited options due to proprietary solutions towards open and programmable network devices. This opens a door to many possibilities. For example, now as soon as we have an innovative idea, we can directly program it into the switch and test it without waiting for the switch manufacture to implement it. ISPs and Data centers can adapt the devices to their own needs with tailor made solutions. Some interesting new ideas have come out of this such as the idea of In-band Telemetry.

Programmable switches can specify customized operations using a new language designed for this purpose such as P4 [1]. P4 is a language for programming switches which allows to program the packet parsing logic. It uses a fast match-action table, as shown in Fig 2. It is a re-configurable match-action table [4] which stores a key (Source IP address, Destination IP address, or any other packet parameter) pointing to what action can be performed (drop, forward, etc.). Thus, in case the corresponding values in an arriving packet matches with the key, then the corresponding action, pointed out by the matching entry in the table, is executed.

A P4 program needs to be compiled and then uploaded to the switch. Then the controller present in the control plane can change the content of the match-action table at run-time. Depending on switch capacity and available memory, many match-action pipelines are possible and they can be executed in parallel. This opens up the possibility to design many applications, design our own protocol, in-network computing, in-network processing to analyse packets in data planes and network telemetry.

2.2 Related work and State of the Art

Recently there has been several works on programmable data planes such as in-network data aggregation [19], in-network telemetry [12],[10], etc. Existing telemetry systems, which do not use programmable data plane, process all packets at servers such as [3] and [25]. They can express many types of queries, but can only support lower packet rates and need to send the packets to the server, causing bandwidth overhead. Some recent first attempts to monitor networks using programmable switches include [17], etc.

Recently a research group, called Computation in the Network (COIN)[2], has been created at Internet Research Task Force (IRTF). One of the use case put forward [8] argues that in-network computing can be applied to enhance security and privacy in the networks.

Coming back to our topic of security attack detection, several works have studied network anomaly detection [21] and network intrusion detection [9,

[1] https://p4.org

[2] https://irtf.org/coinrg

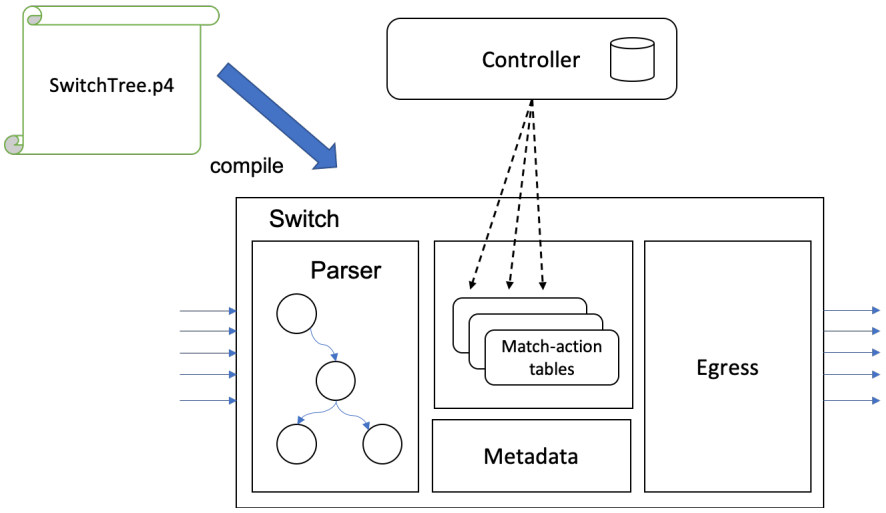

Fig. 2 A switch programmable using P4

7] using machine learning approaches. Distributed Denial of Service (DDoS) detection in the context of SDN is studied in [24]. Work in [2] proposes malware detection even when the traffic is encrypted. They found that Random Forest classifier to be the most robust for such types of problems and feature selection had significant impact on performance. An advanced feature selection algorithm applied to intrusion detection systems is proposed in [1]. Recently, [11] proposed a novel multi-stage optimized ML-based Network Intrusion Detection System (NIDS) framework that reduces computational complexity and enhances detection accuracy. They also used the University of New South Wales (UNSW) dataset for detecting attacks, which is the same dataset used in our experiments. They reported around 99.96% of F1 score for attack detection, however, they seemed to have used a label value (attack category) as an input feature. Attack category is actually a label as mentioned in the paper describing the UNSW dataset [15]. Apart from security attack detection, but for works related to traffic analyses such as traffic classification using machine learning, please see the following survey [18]. However, above works do not perform in-network or in switch detection.

Some works use in-network aggregation to help training distributed machine learning algorithms [20], but they do not embed machine learning inside data plane.

Two recent works are near to our approach. IISY [23] embeds machine learning algorithms such as Decision Trees, SVM, K-means in data plane. However, they do not embed Random Forests and they tested their algorithm for traffic classification with relatively simple features such as the value of source or destination port. As compared to their work, we show how to compute advanced stateful features such as round trip time of a flow. Another interesting work, which is very recent, pForest [6] implements Random Forests

in programmable switches. They propose an advanced feature compression technique which results in low consumption of memory. However, their source code is not available to this date and, thus, we implemented[3] our own version of Random Forests embedded in a programmable switch. As compared to pForest, we provide algorithms to compute more complex and stateful features and some of our features are more accurate as will be discussed in the text later.

3 Detection of Attacks using Random Forests

Before embedding Random Forests inside data plane, let us first focus on detecting attacks using Random Forests in general. In order to train the Random Forest, we use the UNSW-NB15 dataset[4] [15] [16]. It consists of 2 million records with 100GB of network traces. This dataset has nine types of attacks: Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode and Worms. There are a total of 49 network data features, which includes labels: attack category and class label. Some of these features are simply the fields found in packet headers such as source or destination IP address, ports, time to live (TTL), etc. Some are derived based on time such as duration, jitter, inter-arrival of packets, round trip time, etc. Others are again derived statistics such as bitrate, average packet size and loss rates. Finally the rest are further derived from the above features using special algorithms such as *ct_state_ttl* is derived using source/destination time to live. It should be noted that the features are specified separately for both directions: source to destination as well as in inverse direction. As we will explain later, we detail only 12 features in Table 1 as they were the ones chosen for our study. We also provide *ct_state_ttl* calculation algorithm (Algorithm 1), which is an existing algorithm, for the readers.

In this study, we do not focus on detecting individual types of attacks, but as a first step, we focus on detecting whether a network flow is normal or abnormal. We can embed all the 49 features in the switch, but one trade-off of embedding so many features is that it takes switch memory. In fact as we will discuss later, stateless features are easy to implement, but implementing stateful features inside a switch is challenging and requires significant switch memory. Additionally as Random Forest involves many Decision Trees, embedding several Decision Trees takes switch memory.

Thus, in this section, we consider the selection of the number of trees needed and number of features to consider. Note that the approach in general is not limited by these parameters and depending on the available switch memory, these parameters can be adapted as required.

[3] https://https://github.com/ksingh25/SwitchTree

[4] https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/

Result: Value of ct_state_ttl
Initialization: ct_state_ttl ← 0;
if *(sttl == (62 or 63 or 254 or 255) and dttl == (252 or 253) and state == FIN)* **then**
  ct_state_ttl=1 ;
else if *(sttl == (0 or 62 or 254) and dttl == 0 and state == INT)* **then**
  ct_state_ttl=2 ;
else if *(sttl == (62 or 254) and dttl == (60 or 252 or 253) and state == CON)* **then**
  ct_state_ttl=3 ;
else if *(sttl == 254 and dttl == 252 and state == ACC)* **then**
  ct_state_ttl=4 ;
else if *(sttl == 254 and dttl == 252 and state == CLO)* **then**
  ct_state_ttl=5 ;
else if *(sttl == 254 and dttl == 0 and state == REQ)* **then**
  ct_state_ttl=6 ;
else
  ct_state_ttl=0 ;
end

Algorithm 1: Algorithm to calculate *ct_state_ttl* [14]. The *sttl* is source to destination time to live, *dttl* is destination to source time to live, state is REQ (TCP, requested) INT (UDP, initiated), EST (TCP, established), CON (UDP, connected), FIN (TCP, finished) or CLO (UDP closed)

Table 1 Selected features

No.	Feature	Description
1	sttl	Source to destination time to live value
2	ct_state_ttl	An integer value calculated as a state according to specific range of values for source/destination time to live
3	dttl	Destination to source time to live value
4	sload	Source bits per second
5	dpkts	Destination to source packet count
6	dmeansz	Mean of the packet size transmitted by the destination
7	sbytes	Source to destination transaction bytes
8	dload	Destination bits per second
9	smeansz	Mean of the packet size transmitted by the source
10	tcprtt	TCP connection setup round-trip time
11	dsport	Destination port number
12	dur	Record total duration

3.1 Features Selection

First of all, we focus on selecting the most important features for detecting a network attack. The dataset has 49 features including labels. Thus, as a first step we trained a Random Forest classifier using all the features and labels. For training classifiers, we used the Scikit-Learn[5] library with Jupyter[6] Notebook. The dataset was randomly split into 75% of training data and 25% of test data. The test data was not shown to the training algorithm.

[5] https://scikit-learn.org/
[6] https://jupyter.org/

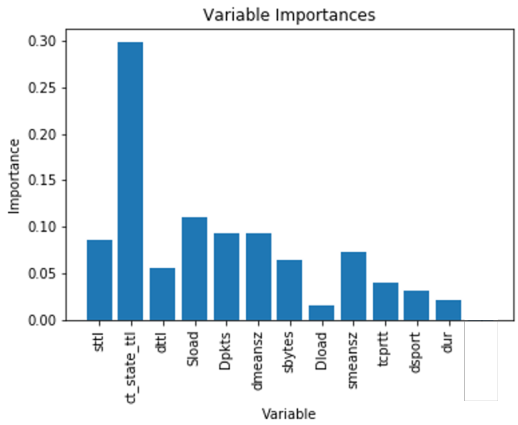

Fig. 3 Importance of different parameters after training a Random Forest with 10 trees and maximum depth of 10.

We first trained the Random Forest without any limit of maximum tree depth or number of trees. After that impurity-based feature importance were considered and the features were sorted according to their importance. As a given feature's importance changes when we eliminate certain features, we eliminated features only one by one and arrived at the final 12 features. Thus, finally we selected only 12 features for this study and this number was found while targeting a F1-score $\geq$ 0.95. Note that any more features can be easily added to SwitchTree. The features that we selected are shown in Table 1. Figure 3 shows relative feature importance among these 12 selected features. In Figure 3 the Random Forest was trained with 10 trees, with a maximum depth of 10 using a pre-compiled dataset extract (UNSW_NB15_training-set.csv and UNSW_NB15_testing-set.csv) which is provided within the UNSW dataset.

3.2 Selecting the Number of Trees

Another Random Forest parameter that we considered is the number of trees in the Random Forest. Note that again we are only considering this number for this study and this number can always be changed depending on the switch capacity.

From the step before, we already fixed the number of features to be 12. Now, we changed the number of trees to check the impact on the detection performance. Higher number of trees in Random Forests allows to generalize the classification task such that over-fitting is avoided. In our case a higher number of trees also takes switch memory thus there is a trade-off.

Table 2 shows the values of precision, recall and F1 scores vs. different number of trees in the Random Forest. It can be seen that the best precision is when there are 10 trees. Increasing to 20 decreases the precision slightly, but improves the recall. This is because more trees generalize better, but at

Table 2 Precision, Recall and F1 score on test data and with different numbers of trees in the Random Forest.

No. Trees	Precision	Recall	F1-score
3	0.9662	0.9725	0.9693
10	0.9791	0.9660	0.9725
20	0.9784	0.9689	0.9736

Table 3 Confusion matrix for test data after training a Random Forest with 3 trees and maximum depth of 10

	Predicted Class 0	Predicted Class 1 (attack)
Real Class 0	169111	369
Real Class 1 (attack)	292	5229

the cost of some precision. Decreasing the number of trees to 3 decreases the performance only slightly. Table 3 shows the confusion matrix when 3 trees are used with 12 features. There are still 292 (5.29%) flows which were not identified. To identify these remaining flows, we will have to add more features, or increase the maximum depth of the tree.

From these results, we decided to focus on using 3 trees only in the Random Forest. With this configuration, 12 features and 3 trees, we have a F1 score of 0.9693. Later we will see that increasing the number of trees to 5 increased the processing cost. It should be noted that there is also another Random Forest parameter called classification threshold which can impact the performance of precision and recall. We can always manipulate that parameter if needed.

4 SwitchTree: Embedding Random Forests in a Switch

This section describes the design of SwitchTree and presents the results.

4.1 General Architecture

Figure 4 shows the architecture of programmable switches that are programmable using P4. When a packet arrives then it is first parsed by the first parser block. After that the packet is passed through a pipeline of several match-action blocks which consist of several tables. The idea is that a key based on packet parameters can be generated and matched with the contents of the given match-action table. Different matches lead to different rows in the table which contain a specific action to be executed. For example a key composed of 5-tuple {Source IP address, Destination IP address, Source Port, Destination Port, Layer 4 protocol} can be used to match an action meant for this specific flow. As an example, a table carrying keys composed of just the destination IP address can be used to match to a row containing the destined exit port

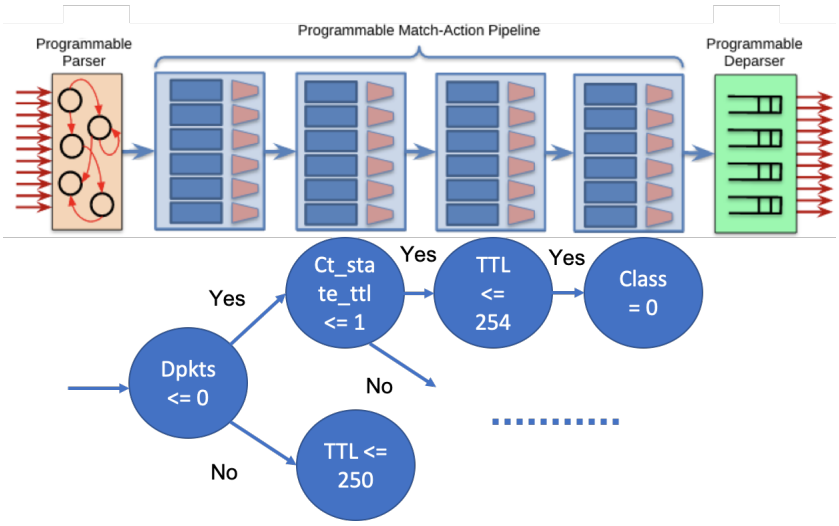

Fig. 4 Example embedding of one of the trees, from the Random Forest, in the switch. *dpkts* is the number of packets sent from destination to source, *ct_state_ttl* is a function of source TTL and destination TTL, and Class is the final classification of the packet.

to realise routing, and so on. Finally depending on a switch's capacities, there can be several pipelines which can be executed in parallel. These match-action tables use advanced hash algorithms and fast memory, which enables them to process packets very rapidly.

P4 enables the programmer to define the structure and semantics of the match-action tables. Once the P4 program is compiled then the controller is able to populate the tables, while respecting the structure and semantics, at runtime and control the match-action process.

In order to embed the Random Forest inside the switch, we need to define the structure and semantics pertaining to Random Forests for the match-action tables. Random forests are composed of several Decision Trees. Each Decision Tree consists of a decision node where a condition has to be satisfied. If the condition is satisfied the child node on left is evaluated, otherwise the child node on the right is evaluated. Thus the evaluation proceeds iteratively till a leaf node is encountered which outputs the classification result. Finally the results of different Decision Trees are combined, for example using majority voting, to obtain the final classification.

Figure 4 shows a Decision Tree embedded inside a match-action pipeline of the programmable switch. Each match-action stage is used to embed a particular level of the Decision Tree. In the example shown, the top most level checks if $dpkts \leq 0$ (which in this specific case translates to whether $dpkts = 0$ as its value cannot be less than 0). If the condition is satisfied (or not) then the processing moves to the next level carrying the results from the previous level where the next condition related to *ct_state_ttl* is checked (just for illustration and nothing to do with whether the checking moves left or right in the Decision

```
table level_n {
    key = {
       meta.node_id: exact;
       meta.prevFeature: exact;
       meta.isTrue: exact;
    }
    actions = {
       NoAction;
       CheckFeature;
       SetClass;
    }
    size = 1024;
}
```

Fig. 5 An example of a P4 table defined to embed a Decision Tree's node of nth level

Tree) and so on. The example of the structure of a table at the nth level, inside the P4 code, is shown in Figure 5.

Each table can define its own specific key. The key used in tables representing Decision Tree nodes is {node_id, previous feature id, previous result} which maps to actions such as NoAction, CheckFeature, SetClass, etc., and these actions accept parameters such as the threshold to compare, which in turn is configurable by the control plane. The parameter node_id is used to differentiate two different nodes in the Decision Tree, which may be on the same level, and thus the same table, and may have the same condition. Thus, node_id value is unique to each node and can be initiated arbitrarily as long as it remains unique for each node. The purpose of node_id is to track where we are in the Decision Tree. Whenever the CheckFeature action is called, it checks the condition described by the input parameters that are provided and uses the threshold value provided by the control plane. The SetClass action sets the final class of the analysed packet when the end node of the Decision Tree is reached.

Note that the exact content of the table will be populated by the control plane. Thus, during runtime, the control plane is not only able to configure the value of the threshold to be checked at each level, but it is also able to configure other things explained in the following. The control plane also configures which feature will be checked next and which action will be called, with which parameters, after a given condition is satisfied or not. For example in the case of the Decision Tree shown in Figure 4, the analyses starts from the level_1 at the root where the condition checked is: $dpkts \leq 0$. Next, the level_2 table will contain the following 2 entries (0 means that the condition, in the level before, was not satisfied and 1 means it was satisfied).

```
Key(node_id, dpkts feature id, 1) -> CheckFeature(ct_state_ttl, 1)
Key(node_id, dpkts feature id, 0) -> CheckFeature(ttl, 250)
```

Thus, in case in level_1 if the previous condition with *dpkts* was not satisfied, then a match will reach level_2 and the next condition to be checked will be: $TTL \leq 250$ (this is just an example and in our studied case we check either *sttl* or *dttl*). On the other hand if the previous condition was satisfied then the condition to check next will be: $ct\_state\_ttl \leq 1$. Some matches will ultimately produce the value of the Class by calling SetClass action. This happens when the end node is reached. However, if that is not the case then the processing will follow through to the next match-action stages.

Finally, several Decision Trees can be embedded inside several match-action stages as either serial stages, in the case of a single pipeline, or parallel stages with parallel pipelines depending on a switch's capability. A final stage compares the results of all the Decision Trees and depending on the algorithm implemented, it outputs the final class. We implemented the voting algorithm such that the final class is the one which is output by majority of the Decision Trees.

4.2 Stateless vs. Stateful Features

It should be noted that some features are stateless and others are stateful that require states to be kept in the switch for each flow. The stateless features are just *sttl* and *dsport* and rest all other 10 features are stateful. Even *dttl* is stateful as the currently arrived packet only contains *sttl* and in order to know about *dttl*, which corresponds to the return packets, it needs to be stored in the switch. Unfortunately for attack detection, mostly the stateful features are important. There are works [13] that study the problem of detecting attacks with stateful vs. stateless manner. They found that some categories such as bandwidth based DoS attacks can be detected in a scalable way. We leave such a study with SwitchTree for future work.

4.3 Estimation of Features

Embedding Random Forests inside programmable switches is relatively easy and the main trick is to make them configurable at runtime. After that, implementation of stateful features is challenging as they require tracking over time and take switch memory. Depending on the application, one could only consider stateless features, but stateful features are required for our case in order to efficiently detect attacks. We explain the estimation of most complicated features in the following text. For the purpose of the Decision Tree implementation using P4, we differentiate between the features that require division or not. The ones requiring division are treated in a special way.

Fractional thresholds: P4 does not support floating point. Thus, the fractional thresholds in the Random Forest models were rounded to the lowest integer. This did not have much impact on the performance as is shown by the results later. In future, we will study if Random Forests can be constrained to have only integer thresholds while training.

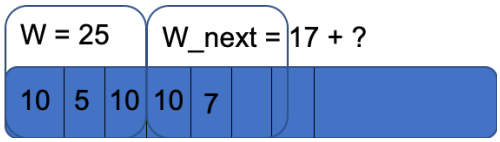

Fig. 6 Dual tumbling windows for bitrate estimation

Bitrate: Estimation of bitrate is required by features such as *sload* and *dload*. Calculating bitrate is challenging as P4 does not support division, does not support floating point and moving average will require to store a lot of past values to be able to calculate an average over a sliding window. pForest [6] uses EWMA filters (limited to only EWMA averaging parameter of 0.5 as division by 2 is achieved by a shift operation) to calculate averages, which does not provide precise values and can lead to inaccurate classification when using pre-trained Random Forests based on actual bitrates. We argue that for detection of attacks, we need to detect them as soon as possible, thus the flow will be short lived before it is detected. We thus estimate the absolute bitrate of each flow in stateful manner based on the total packets received till that point of time. It is estimated as $\frac{8*bytes}{dur}$, where dur is the flow duration as explained in the table describing the features, $bytes$ refers to the bytes transmitted during the duration dur. However, note that division is not allowed in P4. Thus instead of comparing

$$\frac{8 * bytes}{dur} \leq threshold$$

we compare:

$$8 * bytes \leq threshold * dur.$$

Nevertheless, if the long term tracking for flows is required then we use dual tumbling windows to estimate bitrate values. This requires only 2 stateful variables, irrespective of the window size, to store tumbling window values and another third variable to store the window duration. The idea is that the first parameter stores the total size of packets in the previous window and the next parameter stores the size of packets till now. A variable keeps track of time and as soon as more time than the granularity or the desired window size (for example 1s or 100ms, etc.) is elapsed than the W_next window is considered complete and its value is transferred to W. The old value of W gets deleted and W_next gets reset to start calculating the total size of packets received in the next window. This is shown in Figure 6. We now have the size of total packets received and need to calculate the bitrate without using division. The condition is transformed from

$$\frac{W}{time} \leq threshold$$

to:

$$W \leq threshold * time.$$

Average packet size: This feature (required by *smeansz* and *dmeansz* in our case) is also calculated in a similar way as bitrate feature. We show the case for *smeansz*, where *sbytes* is the number of bytes transmitted by the source to destination and *spkts* is the number of packets transmitted by the source to the destination. For short lived flows:

$$smeansz = \frac{sbytes}{spkts}$$

Again the division is avoided by transforming the condition to be checked into:

$$sbytes \leq threshold * spkts.$$

For long term estimations, we use dual tumbling windows. The only difference as compared to that of bitrate estimation is that instead of tracking the time elapsed for the window, a variable tracks the number of packets. Again as divisions are not supported, the condition is evaluated by the number of packets transmitted multiplied to the threshold.

TCP Round Trip Time: To estimate *tcprtt*, we only consider initial exchanges corresponding to the feature which is TCP connection setup round-trip time. A 3-way TCP handshake takes place during TCP connection setup. First the source sends a packet with SYN flag. Second if the destination accepts the connection it sends a packet with SYN flag + ACK for the first packet. Third the source sends an ACK and it can start sending data from this packet onwards. To estimate *tcprtt*, we first track the arrival time of the first packet with SYN flag from source in a stateful way. After that we look for the arrival of ACK (third packet of the 3-way handshake) from the same source and for the same flow. The value of *tcprtt* is the difference of time between ACK and SYN packets.

4.4 Hash Mapping

After estimating the features another challenge is to track them for each flow. We use registers and hash mapping to keep track of per flow parameters. The features or states of a flow are tracked using switch registers indexed using a hash over 5-tuple values of a flow as shown in Figure 7. As the number of registers are limited, a timeout value is used to erase the values of old flows which have not been seen since some time. In experiments, we used $15s$ as the timeout value. Hash collisions can still occur depending on how many registers are used and how many flows are observed at one time. In case of hash collision, the observed flow will not be classified and thus we mark it and forward it to the control plane for further analyses. The hash collision is detected by storing the packet header fields: source IP address, destination IP address, source port, destination port and protocol at the index pointed by

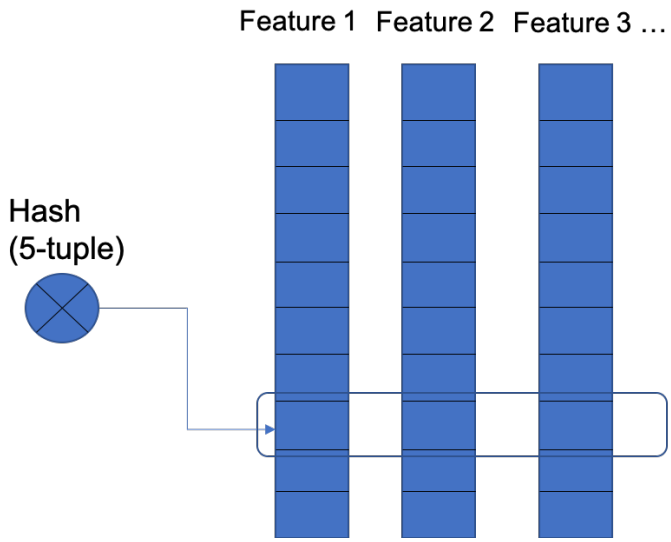

Fig. 7 Hash maps to track stateful features

the hash. Every time a packet arrives then its corresponding fields are checked with those present at the location pointed by the index. In case the values are different then it is a hash collision.

As a future work, we would like to use a bloom filter [5] for tracking some binary parameters and eliminate some redundant checks. For example, at present, one problem is that a given flow continues to be tracked even when previously it was detected as an attack. Otherwise of course the control plane can also immediately block the future packets from that flow once it was detected as an attack. A bloom filter allows a user to query the presence of a particular flow or not, using minimal memory space. A query returns either "possibly in set" or "definitely not in set." It uses multiple hash functions, for example, using the 5-tuple as the key. These multiple hash functions return different indexes and the bit is set at those index positions. Now if a query wants to know if a flow (based on 5-tuple values) was seen before or not then all these bit positions are checked. If all are set then the flow was probably seen (probably because may be other flows created the same index resulting in a hash collision), but even if one bit is unset then we are sure that the flow was not seen. This can be used to track previously classified flows. Many more flows are normal than abnormal flows. Thus better performance might be achieved by only storing information about abnormal flows as compared to storing information about normal flows.

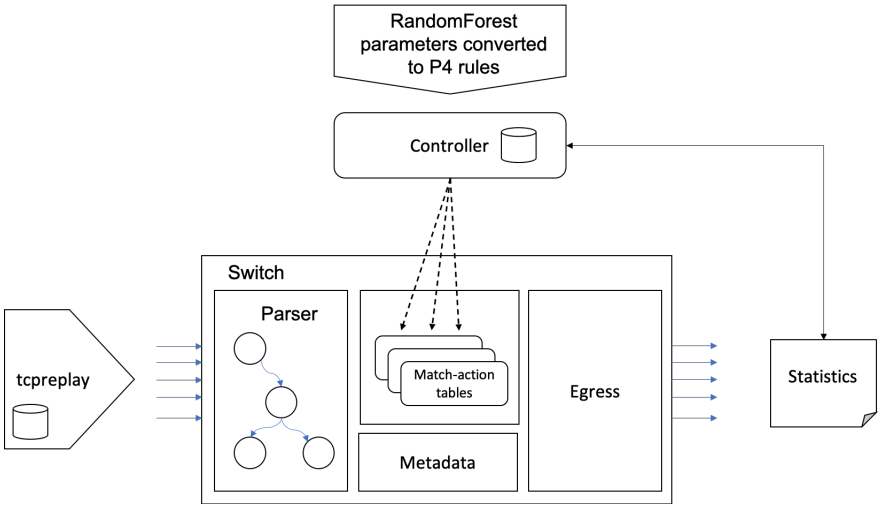

Fig. 8 Experimental setup

4.5 SwitchTree Performance

In order to test the performance of SwitchTree, we implemented SwitchTree[7] inside the BMV2 behavioral model[8] of the P4 switch. Note that BMV2 is not meant to be production grade and is meant for developing and testing P4 programs. The network trace from UNSW-NB15 was fed into the switch using tcpreplay as shown in Figure 8. We fed a total of 1800611 packets with peak 4000 packets/s approx. and average 1900 packets/s approx. It corresponds to the data captured in the UNSW traces between 2015-01-22 13:49:36 and 2015-01-22 14:05:04 UTC. The Random Forest parameters were converted to switch rules using a script and were installed from the control plane. The virtual BMV2 switch was run on a machine with 16GB RAM and 8 cores of CPUs. The debugging of BMV2 was disabled for test runs to avoid packet loss as packets are analysed at line speed. Enabling debugging results in high packet losses.

The packets arrive[9] at port 1 and then the packets classified as attack are forwarded to port 2 and the ones classified as normal are forwarded to port 3. At the end the received packets on these 2 ports are compared with the ground truth to evaluate the performance. The statistics are also obtained from the controller which keeps track through the use of counters available in P4 language.

In order to compare the packet processing performance of SwitchTree with respect to number of trees, we performed stress testing. Here SwitchTree is

[7] https://github.com/ksingh25/SwitchTree

[8] https://github.com/p4lang/behavioral-model

[9] As we use network traces, we send the inbound as well as the outbound packets on this same port for analyses.

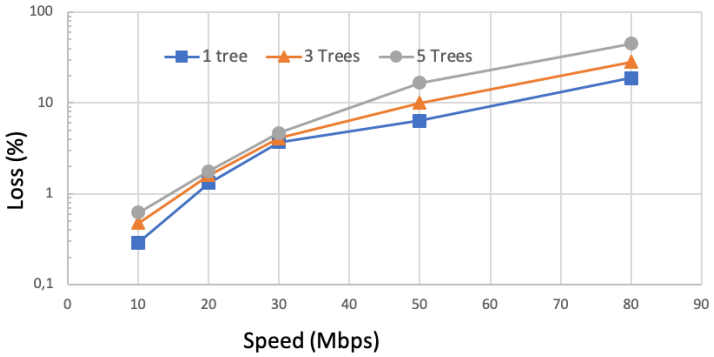

Fig. 9 Stress testing. The first point corresponds to the speed at which original traffic was captured and is only approx. 10 Mbps. Note that SwitchTree here runs on a machine and better performance should be expected while running on a switch.

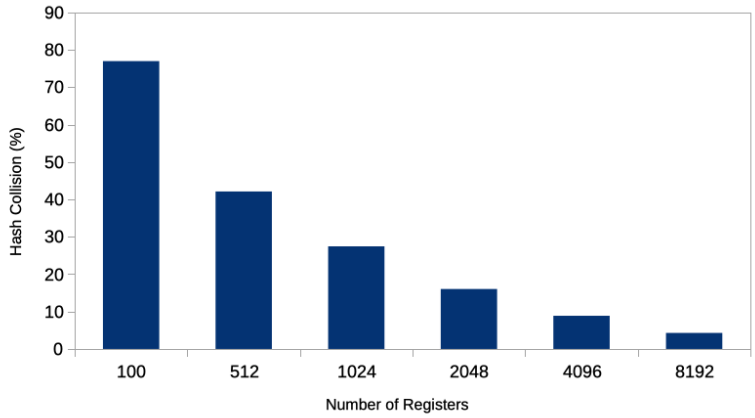

Fig. 10 Hash collision percentage for arriving packets vs. number of registers per tracking variable

running on a machine and BMV2 which is not production grade. Better performance is expected when it runs on a production grade switch. First the traffic was sent at its normal rate of capture. Then the traffic injection speed was gradually increased using a tcpreplay parameter. This was to compare the performance when the number of Decision Trees are increased. As shown in Figure 9 the losses increase when the traffic speed is increased. Moreover, losses increase as the number of trees increase. This is because with more trees the packets have to pass through more conditions. This increases the processing cost and thus increases the chance that the queues will be full and arriving packets will be dropped. The attack detection accuracy also decreases because already the time related features do not get estimated correctly when the traffic speed is not original. Thus, the detection accuracy for accelerated traffic is not shown.

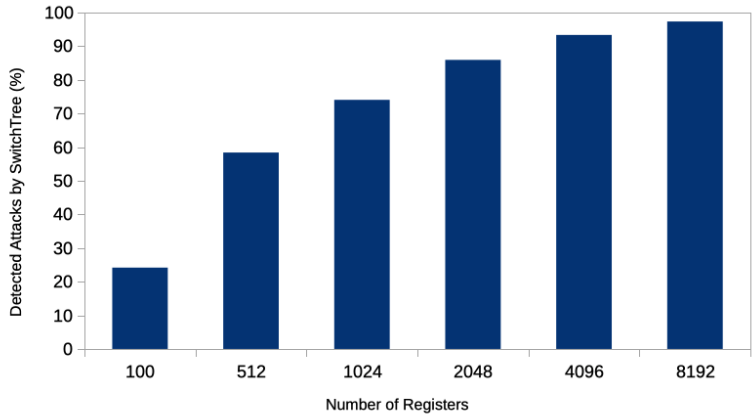

Fig. 11 Detected attack percentage in terms of flows vs. number of registers per tracking variable

About attack detection accuracy, we first show results when using just 1 tree, thus in this case the Random Forest becomes a Decision Tree. After that we show the results with Random Forest of 3 and 5 Decision Trees.

One of the crucial switch parameter is the number of registers available or other memory elements which can store the feature states. Each feature needs to track certain number of tracking variables per flow. For example *dttl* just needs 1 tracking variable per flow to store the value of *dttl* for a given flow, however, *sload* required at least 2 tracking variables to store the number of transmitted bytes and total duration for a given flow. Thus, we study the performance of SwitchTree by varying the number of registers per *tracking variable*. Low number of registers mean that there will be hash collisions and we will have to let the flow pass with just a mark, to be analysed by some other network element later. Thus, a low number of registers can still help by detecting some attacks and marking the other flows for further analyses by other elements in the network. Figure 10 shows that the number of hash collisions decrease with the increasing number of registers. With 8192 registers, we observed 4.282% collisions. We also observe in Figure 11 that the attack detection percentage improves with the number of registers used. For example with 8192 registers per feature, SwitchTree detects 97.36% of attacks. The above results correspond to just 1 Decision Tree. Also note that even after disabling debugging during our experiments, for optimal performance, we noted around 0.26% of packet losses. They decreased the detection performance, even though very slightly. Losses are still present even when no table is implemented inside BMV2. These packet losses are either due to system errors or due to sudden peaks of heavy load in the traffic.

Next, we tested Random Forest with 3 and 5 trees. With multiple trees, their classification results are aggregated to achieve a final classification. The SwitchTree implementation with more trees takes more memory as more num-

Table 4 SwitchTree with more trees

Number of Trees	Detected Attacks
SwitchTree (1 Tree)	97.36% (on 1 GB Data at line speed
SwitchTree (3 Trees)	94.45% (on 1 GB Data at line speed)
SwitchTree (5 Trees)	87.35% (on 1 GB Data at line speed)

ber of match-action tables are used. However, the behaviour of hash collisions remain same as features are only computed once per packet even when there are multiple trees. Table 4 shows the results. It can be seen that adding more trees degraded the performance in terms of a decrease of 2.91% with 3 trees and a decrease of 10.01% with 5 trees. This is because of increasing losses with increasing number of trees. Moreover, some times the results from different trees can conflict and this increases with number of trees. In any case, this is a trade-off as more trees are known to generalise better and thus perform better on new data, at the cost of decrease in performance.

5 Conclusions

In this paper, we proposed SwitchTree which is an embedded Random Forest algorithm inside a programmable switch. We showed that complicated features needed to detect network attacks can be estimated using hash mapping, inside data plane. This however makes the performance results dependent on the number of registers, or other memory elements, which can store the states. Thus, it becomes an equipment dimensioning question and in-network detection can be performed if the switch has the required capacity. Switch with low memory can still help by detecting and offloading some attacks and marking other flows, in case of hash collision, to be detected by other network elements or servers. Using system level experiments, SwitchTree is able to detect more than 94% of network attacks.

As future work, we would like to add more features to improve performance and would like to explore ways to decrease the number of registers needed and optimise the packet processing capability. One idea is to use bloom filters for features requiring binary states.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Ambusaidi, M.A., He, X., Nanda, P., Tan, Z.: Building an intrusion detection system using a filter-based feature selection algorithm. IEEE transactions on computers **65**(10), 2986–2998 (2016)

2. Anderson, B., McGrew, D.: Machine learning for encrypted malware traffic classification: accounting for noisy labels and non-stationarity. In: Proceedings of the 23rd ACM SIGKDD International Conference on knowledge discovery and data mining, pp. 1723–1732 (2017)
3. Borders, K., Springer, J., Burnside, M.: Chimera: A declarative language for streaming network traffic analysis. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), pp. 365–379 (2012)
4. Bosshart, P., Gibb, G., Kim, H.S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., Horowitz, M.: Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. ACM SIGCOMM Computer Communication Review **43**(4), 99–110 (2013)
5. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet mathematics **1**(4), 485–509 (2004)
6. Busse-Grawitz, C., Meier, R., Dietmüller, A., Bühler, T., Vanbever, L.: pforest: In-network inference with random forests. arXiv preprint arXiv:1909.05680 (2019)
7. Devan, P., Khare, N.: An efficient XGBoost–DNN-based classification model for network intrusion detection system. Neural Computing and Applications pp. 1–16 (2020)
8. Fink, I.B., Wehrle, K.: Enhancing Security and Privacy with In-Network Computing. Internet-Draft draft-fink-coin-sec-priv-00, Internet Engineering Task Force (2020). URL https://datatracker.ietf.org/doc/html/draft-fink-coin-sec-priv-00. Work in Progress
9. Ghanem, W.A., Jantan, A.: A new approach for intrusion detection system based on training multilayer perceptron by using enhanced bat algorithm. Neural Computing and Applications pp. 1–34 (2019)
10. Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., Willinger, W.: Sonata: Query-driven streaming network telemetry. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, pp. 357–371 (2018)
11. Injadat, M., Moubayed, A., Nassif, A.B., Shami, A.: Multi-stage optimized machine learning framework for network intrusion detection. IEEE Transactions on Network and Service Management (2020)
12. Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., Wobker, L.J.: In-band network telemetry via programmable dataplanes. In: ACM SIGCOMM (2015)
13. Kompella, R.R., Singh, S., Varghese, G.: On scalable attack detection in the network. In: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, pp. 187–200 (2004)
14. Moustafa, N.: Designing an online and reliable statistical anomaly detection framework for dealing with large high-speed network traffic. Ph.D. thesis, University of New South Wales, Canberra, Australia (2017)
15. Moustafa, N., Slay, J.: UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In: 2015 military communications and information systems conference (MilCIS), pp. 1–6. IEEE (2015)
16. Moustafa, N., Slay, J.: The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set. Information Security Journal: A Global Perspective **25**(1-3), 18–31 (2016)
17. Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., Kim, C.: Language-directed hardware design for network performance monitoring. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 85–98 (2017)
18. Pacheco, F., Exposito, E., Gineste, M., Baudoin, C., Aguilar, J.: Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. IEEE Communications Surveys & Tutorials **21**(2), 1988–2014 (2018)
19. Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M., Kalnis, P.: In-network computation is a dumb idea whose time has come. In: Proceedings of the 16th ACM Workshop on Hot Topics in Networks, pp. 150–156 (2017)
20. Sapio, A., Canini, M., Ho, C.Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D.R., Richtárik, P.: Scaling distributed machine learning with in-network aggregation. arXiv preprint arXiv:1903.06701 (2019)
21. Tama, B.A., Rhee, K.H.: An in-depth experimental study of anomaly detection using gradient boosted machine. Neural Computing and Applications **31**(4), 955–965 (2019)

22. Wang, M., Cui, Y., Wang, X., Xiao, S., Jiang, J.: Machine learning for networking: Workflow, advances and opportunities. IEEE Network **32**(2), 92–99 (2017)
23. Xiong, Z., Zilberman, N.: Do switches dream of machine learning? Toward In-Network Classification. In: Proceedings of the 18th ACM Workshop on Hot Topics in Networks, pp. 25–33 (2019)
24. Xu, Y., Liu, Y.: Ddos attack detection under sdn context. In: IEEE INFOCOM 2016-the 35th annual IEEE international conference on computer communications, pp. 1–9. IEEE (2016)
25. Yuan, Y., Lin, D., Mishra, A., Marwaha, S., Alur, R., Loo, B.T.: Quantitative network monitoring with NetQRE. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 99–112 (2017)