2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)

Tracking network flows with P4

1st Joseph Hill
System and Network Engineering group (SNE)
University of Amsterdam
Amsterdam, The Netherlands
j.hill@uva.nl

2nd Mitchel Aloserij
Informatica Bachelor
University of Amsterdam
Amsterdam, The Netherlands
mitchelalosery@quicknet.nl

3rd Paola Grosso
System and Network Engineering group (SNE)
University of Amsterdam
Amsterdam, The Netherlands
p.grosso@uva.nl

***Abstract*—Tracking flows within a single device, as well as tracking the full path a flow takes in a network are core components in securing networks. Malicious traffic can be easily identified and its source blocked. Traditional methods have performance and precision shortcomings, while new programmable devices open up new possibilities. In this paper we present methods based on the P4 programming language that allow to track flows in a device, as well methods toward full path tracking. A core component of this work are Bloom filters, which we have implemented fully in P4. To validate our approach and implementation we have carried a study in a specific use case, namely the detection of SYN attacks.**

***Index Terms*—Programmable networks, P4, Bloom filters, SYN attacks.**

I. Introduction

Tracking network flows allows network administrators to maintain a good overview of their network operations and health. Tracking network flows also has clear advantages from a security perspective; knowing the path that a flow takes allows to identify the source of the malicious traffic; recognising that a specific flow is exhibiting a non-standard behaviour can thwart emerging DDoS in their initial phases.

Conventional methods of tracking the flow of data through a network can be resource intensive. This is especially true when the control plane has to examine each packet. Resource utilisation can rise to the point that it is detrimental to the performance of the device. In order to keep resource consumption at acceptable levels some implementations will resort to the sampling of data [1]. This combined with a device centric approach, where each device has to do all the work of logging traffic, can lead to incomplete information. While in hardware implementations of traffic logging exist, these can be inflexible with a limited number of ways of identifying and correlating traffic. Hardware implementations may also have fixed amounts of memory allocated to traffic logging that are not sufficient in the network environment.

The interesting question we can ask ourselves is if programmable devices can offer a solution to this problem. We believe this is the case, and the work we present here looked at how P4 can be used to help secure networks by ultimately enabling the tracking of the complete path of data through the network.

P4 [2] is a language designed to program the data plane of packet forwarding devices such as switches and routers. P4 is protocol independent meaning that it has no predetermined notion of the format of a packet. This allows for the definition of new protocols as necessary and eliminates constrains on how individual packets can be correlated. P4 also allows for the flexible allocation of device memory. For instance a P4 programmer can decide to allocate memory not needed for routing tables to flow tracking instead. While P4 still requires a control plane to handle most state changes, it has the potential to allow for the efficient tracking of network traffic with a solution tailored for the environment.

In the following we will provide background on the P4 programming language, as well as on Bloom filters, which are an essential element in our approach (see Sec.II and Sec.III). Sec.IV provides details on our P4 implementation of the bloom filter in P4; while Sec.VI describes counting Bloom filters and their P4 implementation.

Sec.V shows the accuracy that we can achieve in identifying flows as we vary the parameters of Bloom filters. We chose to apply our implementation on a specific usecase: namely SYN attacks, Sec.VII describes this type of attacks and how we can detect them; Sec.VIII presents our results. We conclude this paper with Sec.IX and Sec.X discussing limitations and possible extensions of our work.

II. P4 language

The P4 forwarding model consists of three parts: parser, ingress match+action and egress match+action, as shown in Fig. 1.

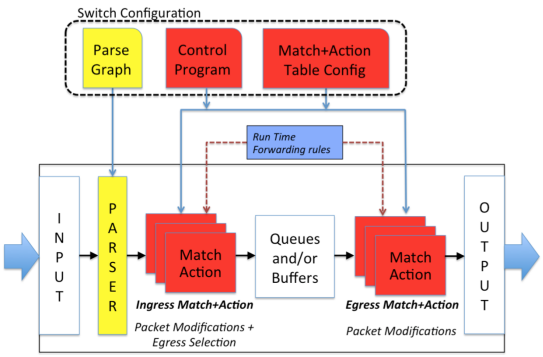

Fig. 1. The P4 forwarding model

A P4 program starts by parsing a packet based on user defined headers. The protocol independent nature of P4 means

978-1-7281-0194-1/18/$31.00 ©2018 IEEE
DOI 10.1109/INDIS.2018.00006

that these headers can be from a well-know protocol such as TCP/IP or something completely new. P4 does not specifically support the parsing of trailers. In some cases it is possible to workaround this by treating the trailer as the last element in a sequence of headers. However, this may not work in implementation that limit the maximum total size of headers parsed. P4 programs will typically need to rely on some external mechanism to handle trailer processing [3, pages 12-17].

Once the headers are parsed, actions are taken based on the results of table lookups. P4 allows matches to be performed based on fields from the packet headers or on other meta-data such as ingress port. There are a variety of match options including exact, longest prefix, and ternary. Each entry in a table specifies an action to be carried out upon match. In the event of a table miss a default action may be performed. User defined actions are based on a limited set of action primitives defined in the P4 specification. These primitive actions allow for the modifying of fields, adding or removing headers, and the cloning of packets. P4 is very limited in the state it can keep from packet to packet. What stateful memories it has are limited to counters, meters and registers [4, page 26]. When a more significant state change is required data must be sent to the control plane.

While P4 allows for parsing, table lookups, and actions to all be done in the data plane, modifications to the tables must be done by the control plane. The P4 specification does not define the interface between the data plane and control plane. How the two communicate is device specific. There are many other aspects that are device dependent as well such as egress port selection, packet replication and queuing [4, pages 61-62].

There are two version of the P4 language specification. This research focuses on $P4_{14}$; this is the older version of the language but it is also currently better supported. $P4_{16}$ is the latest version of the language, which makes significant changes and is not backwards compatible with $P4_{14}$. Many features have been removed from the core language, such as stateful memories, and are intended to be implemented in external libraries [3, page 9]. Until a specification of a standard library for $P4_{16}$ is defined it is difficult to determine how the capabilities of $P4_{16}$ will compare to $P4_{14}$.

This research was conducted using the P4 software switch Behavioral Model (BMv2) and the P4 compiler provided by the P4 Language consortium [5].

III. Tracking Flows

The goal of flow tracking is to gather enough information to be able to determine the entire path a flow took through a network. If multiple paths were taken by what would otherwise be considered the same flow, this should be identifiable from the collected data. It is also important to minimize resource usage while doing this. While NetFlow limits what fields can be used to define a flow, P4 can use any field that can be parsed from packet headers or available via meta-data. The ability to track flows can also be performed at multiple layers. For instance a flow could be tracked in a LAN using a combination of source and destination MAC addresses along with the higher layer protocol fields. With NetFlow each device operates independently, logging the flow defining fields along with what limited information it knows about the path (i.e. ingress port). This means that some state is kept in each device along the path along with the flow data having to be stored in each node transversed. In order to determine the path of a flow each device needs to be queried to determined if it has an entry for the flow. There are a number of P4-based solutions we have devised and that we will describe shortly in our Discussion (see Sec. IX). They all could provide a more cooperative mean of traffic logging with the goal of performing accurate tracking while minimizing the impact on system resources.

In all of these methods the actual logging of the flow is done only at the last node in the path. This requires each node to be able to determine when it is forwarding data out of the tracking domain, but this eliminates the need to replicate the same flow data over every device in the path, saving memory.

A. Bloom filters

Bloom filters can be used in this last node to track flows. A Bloom filter implemented in the data plane allows additional state to be tracked without involving the control plane. It can be used by a node to determine if a flow has been seen previously and therefore should not be sent to the central collection point. This would reduce the amount of memory required on each node to hold logging data.

A Bloom filters is a way to represent a set. They allow items to be added to the set and they can be queried to see if an item is a member of the set. Items can not be retrieved or deleted from the set. Once an item has been added to a Bloom filter, a membership query will always return true. However, there is a chance that a membership query for an item that has not been added to a set will also return true. In other words a Bloom filter has no false negatives by may have false positives. This possibility of a false positives is a trade off for the memory efficiency of a Bloom filter [6].

A Bloom filter is implemented with an array of bits and a predetermined number of hash functions [7]. When an item is added to the set it is hashed by each function. Based on the output of each hash function a bit is set to one in the array. When checking to see if an item is a member of the set, the same process is followed except instead of setting the bits they are checked to see if they are already set. If all the bits are set then the item is considered to be in the set. As more items are added to the set, there is an increasing chance that an item that is not actually a member of the set will coincidently have the same bits set, causing a false positive.

Figure 2 shows the insertion process of 2 objects into a Bloom filter; this figure also illustrates the problem indexes from hash functions colliding with indexes from other objects.Such false positives occur when every bit that corresponds to a specific object has already been set to 1, while that object has never been inserted into the bloom filter.This can happen as the size of the Bloom filter is finite and when the

filter gets saturated with objects, the hashed indexes of these objects start to collide up to a point where all the indexes of a new object have already been marked by previous objects. This problem can not be solved; thus Bloom filters should only be used in implementations where false positives are not fatal, can be prevented or can be handled in any other way.

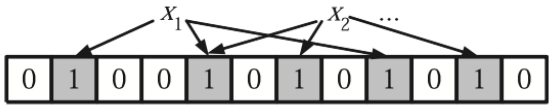

Fig. 2. A basic Bloom filter of size 12 where two tags (x_1 and x_2) are being inserted into the bloom filter using 3 different salts [8]

IV. P4 Bloom filters

To implement the network flow detection with bloom filters on a P4 enabled device, the implementation has to be split into 6 different sections and these are: 'packet header parsing', 'creating the bloom filter', 'hashing the packet', 'storing the values', 'obtain the flow amount' and 'creating the response'.

1) Packet header parsing: The first step is to define the possible packet headers that an incoming packet can have. To do this, there need to be a header structure inside the program that describes how a packet header is organized and how big every entry in the packet header is. In case of IPv4 the packet header contains 12 different values with a total size of 160 bits and a variable options sections which can have a maximum length of 60 bits. After every packet header that is allowed to pass through the device is defined, the next step becomes the creation of the packet header parsers. A packet header parsers parses the raw data that a forwarding device obtains upon receiving the packet into one or more of the pre-defined packet headers. A developer can now program the parser to always attempt to parse a packet using a certain packet header or the developer can also make this dynamic by letting the packet header depend on the packet parameters it has already parsed.

```
header_type ipv4_hdr_t {
        fields {
                version    : 4;
                ihl        : 4;
                tos        : 8;
                total_len  : 16;
                id         : 16;
                flags      : 3;
                offset     : 13;
                ttl        : 8;
                proto      : 8;
                checksum   : 16;
                src        : 32;
                dst        : 32;
                options    : *;
        }
        length      : 4 * ihl;
        max_length : 60;
}
header ipv4_hdr_t ipv4_hdr;
```

Listing 1. The header definition for an IPv4 packet

In the implementation of this project the following headers are defined: ethernet, IPv4, IPv6, TCP, UDP, flow_metadata and bf_metadata. An example of a chosen header structure are the headers TCP and UDP. The header that is chosen is based upon the protocol which is defined in IPv4 or IPv6 header. The flow_metadata header is a header that will contain the port of a specific packet. The reason we created this header is that the port number can either be defined into the TCP header or in the UDP header; by creating a new header and copying the port number into it, the corresponding value will be at one centralized place. The bf_metadata header is a header that will be used internally to temporarily store values that are computed by the implementation.

2) Creating the bloom filter: The next step is to create a bloom filter. The implementation of the bloom filter is quite easy because it is in fact just an array of a fixed size. The register object can be used to create objects inside the forwarding device where the program can read from or write towards. To define a register the user needs to define the size of an entry and the amount of entries inside the register. This register can then, after is has been defined, be manipulated using the register_read() and register_write() actions. Furthermore registers can also be manipulated from the control plane which makes them even more useful because an user can obtain and manipulate the network flow statistics and eventually also the DDoS detection statistics from the control plane.

```
register default_bf {
    width : 16;
    instance_count : 1000;
}
```

Listing 2. An register with a size of 1000 entries where each entry has the size of an integer (16 bits)

3) Hashing the packet: In order to transform the array of integers which was previously created into a counting bloom filter, the program must use a hashing algorithm to compute an index of bloom filter. In P4 this can be done by using the field_list and field_list_calculation objects. The field_list object is an object that defines a list of values that can be found in the parsed packet headers. The field_list_calculation object is an object that takes an input and performs a certain operation on the content of the field list. In the case of this project the field_list_calculation object will take a field_list that contain all the values that define a network flow and perform a hashing operation on these values in order to obtain an index. This index can then be used to access a specific entry in the previously created bloom filter. To improve the accuracy of bloom filters developers often create multiple hashes with different salts of the same field_list. This will result into different index values where the values of each entry can then be compared to each other to obtain a more accurate result. In the implementation of this project the salt has been added to the bf_metadata and also to the field_list and this will add it to the list of values that will be hashed upon.

```
field_list bf_flow_fields {
    ipv4_hdr.src;
    ipv4_hdr.dst;
    flow_metadata.src_port;
    flow_metadata.dst_port;
    bf_metadata.salt;
}

field_list_calculation bf_hash_func {
    input {
        bf_flow_fields;
    }
    algorithm : xxh64;
    output_width : 64;
}
```

Listing 3. The field_list and field_list_calculation

4) storing the values: The next step after defining the hashing methods is to actually hash the packets and increment certain values in the bloom filters. In order to perform the hash functions the implementation has to call the action 'modify_field_with_hash_based_offset'. This action will generate an index value based upon a specific field_list_calculation. The resulting index will then be stored in the bf_metadata which is then used to increment an entry in bloom filter with that specific index. A downside of the P4 programming language is that this programming language only allows actions to be called within actions or when a table+action match has been found. But in this implementation this action needs to be called upon every packet and the way this problem is solved in this project is to create a table which tries to match on an arbitrary value, which every packet has, and the default match will result into the desired hash action.

5) obtain the flow amount: After the packet has been processed and the right values in the bloom filter have been incremented, it is time to obtain the flow values from the bloom filter. The purpose of obtaining these values is in this stage of the implementation just for testing purposes but in a later stage these values will be used to decide if a packet is malicious or not. In order to obtain the amount of packets that have traveled through this flow the same function as to increment the values in the bloom filter is executed to obtain the right indexes for the bloom filter. The major difference is that the program now reads the values and stores all the values into the bf_metadata object. The implementation will then search for the smallest entry between all the results and saves that value. Also here the implementation is faced with the problem that actions can only be called using a match+action table or within other actions. Furthermore this implementation faces another down side of the P4 programming language and that is that control objects are the only objects that can perform variable comparison. So the implementation first has to perform a match+action from the control object to obtain the flow values from the data structure which will then be stored in the bf_metadata header. It then needs to return to the control object to compare the flow values and pick the smallest one, which will then used in another match+action table to trigger the action that will save the lowest value. The following figure IV-5 will show several code snippets that illustrate this program flow for incrementing one entry in the bloom filter.

```
// The ingress control object, this will start
    the processing of the packets
control ingress {
    ...
    ...
    apply(detect_flow);
    ...
    ...
}

// The match+action table that will match
    every packet with the def_bf_insert action
table detect_flow {
    reads { ether_hdr.ethertype : exact; }
    actions {
        def_bf_insert;
    }
}

// This action will increment a value in the
    bloom filter bf using a specific salt.
action def_bf_insert() {
    // Set values in the bloom filter using
        different salts
    bf_set_bit(bf, BF_SALT1);
    bf_set_bit(bf, BF_SALT2);
    bf_set_bit(bf, BF_SALT3);
    bf_set_bit(bf, BF_SALT4);
}

// This action will use the salt and the
    field_list to generate an index and then
    increment the value inside the bloom
    filter at that specific index
action bf_set_bit(bf, salt) {
    modify_field(bf_metadata.salt, salt);
    modify_field_with_hash_based_offset(
        bf_metadata.index, 0,
    bf_hash_func, BF_WIDTH);
    register_read(bf_metadata.tmp_val, bf,
        bf_metadata.index);
    add_to_field(bf_metadata.tmp_val, 1);
    register_write(bf, bf_metadata.index,
        bf_metadata.tmp_val);
}
```

Listing 4. A simplified program flow for incrementing entries in the bloom filter based upon a specific packet

6) Creating the response: The final step is to create a response based on the amount of packets a certain flow has seen. In this stage of the implementation the result simply gets added to an unused field of the packet and then returned but in later stages the implementation can make decisions based upon this value to, for example, mark this packet as malicious or to just drop this packet. In the case of this implementation the result simply gets written to the 'tos' field of the IPv4 packet header. The reason for this field is that this field is not required for the communication to succeed and thus can be used to store data while experimenting with the implementation.

V. Accuracy

The performance of a Bloom filter can be thought of in terms of its accuracy in identify new flows. Specifically accuracy will be measured as the possibility that a new flow will be correctly identified as not a member of a set after a given number of unique flows have been added. As more items are added to the Bloom filter the false positive rate increases, decreasing its accuracy. By adjusting the number of bits in the array and the number of hash functions used, the performance can be tuned. Note that the size of the items being added to the set have no effect on the accuracy of a Bloom filter.

To calculate the accuracy of a specific Bloom filter the following process is followed. A statistics array is created with elements indexed from zero to the maximum number of flows to be sent. Each element is initialized to zero. For each round the Bloom filter is initialized by setting all bits to zero. A single packet for each of the unique flows is generated in advance. Each packet is marked by the P4 switch to show whether or not it is detected as previously seen. Each time the P4 switch detects it as previously seen, a false positive, the element in the statistics array for the number of flows seen at that point is incremented. After a number of rounds has been run the total number of false positives for each flows seen amount is divided by the number of rounds run. This gives the false positive rate for that number of flows seen. The inverse is taken to determine the accuracy.

To show how the size of the array affects performance figure 3 shows the accuracy of Bloom filters with an array size of 1024 and 2048 bits. As one might expect, the Bloom filter with twice the number of bits shows a higher accuracy. Figure 4 shows the performance of three different Bloom filters all with arrays of the same size but using a different number of hash functions. Here it is not as clear what the best option is. Initially, the Bloom filter with two hash functions has slightly worse performance then the other two but has the highest accuracy when more than 500 flows have been seen. The optimal array size (m) can be calculated using formula 1 given a desired false positive rate (p) with n items in the set [8]. The optimal number of hash functions (k) can be calculated using formula 2 given the optimal number of bits (m) and the same n items in the set (n) [9].

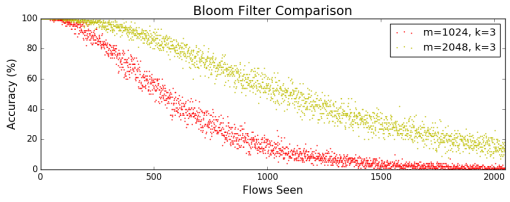

Fig. 3. Bloom filters with different array sizes.

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{1}$$

$$k = \frac{m}{n} \ln 2 \tag{2}$$

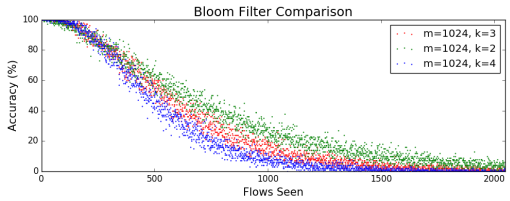

Fig. 4. Bloom filters with a varying number of hash functions.

These equations are then used to get the parameters for a Bloom filter tuned to have an accuracy of 95% after 64,000 flows have been added to the set. This results in a Bloom filter with an array size of 408,632 bits (≈50 KiB) and four hash functions. Figure 5 shows the performance of this Bloom filter in red. In blue the cumulative accuracy is also shown, which is the percentage of flows correctly identified as not in the set up to that point. While this Bloom filter is tuned to have a 95% chance of correctly identifying the 64001st unique flow, it is important to note that it has correctly identified 98.84% of the first 64000 unique flows seen. With sampled NetFlow, a flow may escape logging by being short lived or keeping its traffic to a low percentage of the current traffic moving through the network. With Bloom filters a flow's chance of not being logged only increases as more flows have been seen, regardless of the number of packets in a flow. This property of Bloom filters makes it more difficult for malicious traffic to remain undetected by minimizing traffic.

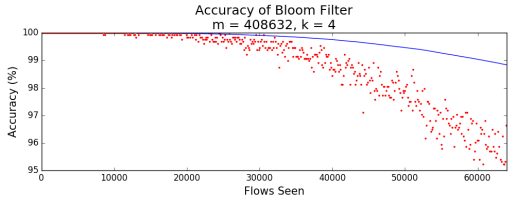

Fig. 5. Bloom filter tuned for performance with cumulative average..

VI. Counting Bloom filters

Apart from detecting flows, it is interesting in some cases to count how many times packets from a flow have passed through a device. This require us to use counting Bloom filters. The counting Bloom filter uses counters instead of bits as elements of the array. This allows for the value at a certain index to exceed past one and thus allows the user to not only check if a certain object has been seen before but also how often a certain object has been seen before [9]. To do this, one has to obtain the values from every corresponding entry in the Bloom filter and return the entry with the lowest value. The lowest value is the amount of times the object has at least passed through the bloom filter. The other values have been influenced by hashes from other objects, and they effectively combine different counters.

A problem with all types of Bloom filters is that they fill up over time. We will use a method that will decrease the entries

in the counting bloom filter over time. Our implementation will keep track of the amount of times an object is inserted into the Bloom filter; after inserting a certain amount of entries, every entry in the bloom filter will be decremented by one except for when an entry is equal to zero because an entry cannot go below zero and also cannot wrap around to a maximum value. Note that 'the amount of times an object is inserted into the bloom filter' refers to the amount of times an object is inserted as a whole and not the individual insertions performed by the unique hash functions. By doing this a lot of flows with a small amount of packets will be removed from the counting Bloom filter; this will improve the accuracy of the Bloom filter, as collisions are minimized.

A. Counting Bloom filters in P4

1) Counting the objects: The first step is to count the objects that are inserted into the Bloom filter. This is done by creating a register object which contains only one entry and that entry has a size that is at least as large or preferably larger as the threshold that is going to be used to decide whether to decrease the entries in the bloom filter. Figure 4.6 shows the register used in the project. Now when ever an entry gets inserted into the bloom filter, for example while using the 'def_bf_insert()' action inside figure 4.5, this register can be incremented by one and this will allow the implementation to perform packet counting.

```
register pkt_counter {
    width : 8;
    instance_count : 1;
}
```

Listing 5. Packet counting register with 1 entry with a size of 8 bits.

2) Applying threshold to the counter: The second step is to check whether the counter has surpassed a specific threshold. This validation can be done in the P4 programming language by using the match+action table which will attempt to match the value of the register to the value of the threshold; if there is match we will decrease every entry in the bloom filter as well as resetting the packet counter.

3) Decreasing the counting bloom filter: The final step is to actually decrease every entry in the counting bloom filter. The P4 programming language unfortunately does not support looping the adjusting of every entry in a register at the same time. Thus in order to decrease every entry in the bloom filter the implementation has to decrease every entry in the bloom filter by one. Figure 4.7 shows a small snippet of this list of actions.

```
...
decr_register(default_bf, 50, 1);
decr_register(default_bf, 51, 1);
decr_register(default_bf, 52, 1);
decr_register(default_bf, 53, 1);
decr_register(default_bf, 54, 1);
decr_register(default_bf, 55, 1);
...
```

Listing 6. Decrementing the packet counter registers

Lastly after every entry in the bloom filter has been decreased, the packet counter also has to be reset back to zero.

B. Accuracy of counting Bloom filters

Like for traditional Bloom filters the accuracy of a counting Bloom filters will vary depending on a number of parameters, such as the size of the filters, the number of hashes used and the number of flows passing through the device. Fig. 6 shows the accuracy for three counting Bloom filters of different size, as function of the number of flows. The filters use 4 unique hashes to insert packets in the filter. Like in the case of regular Bloom filters accuracy is higher for larger sizes.

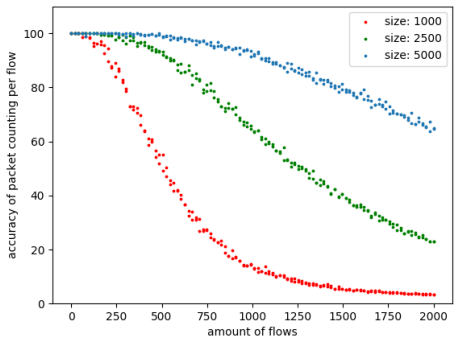

Fig. 6. The accuracy of the counting Bloom filter as a function of the bloom filter size while varying the amount of flows.

Even more interesting for us is to observe the accuracy as function of the decrementing rate. Fig. 7 shows the accuracy of a counting Bloom filters of size 2500 as function of the number of flows passing through the device. Every curve represents a different number of entries before the counters are decremented by one, namely 100, 250 and 500 entries.

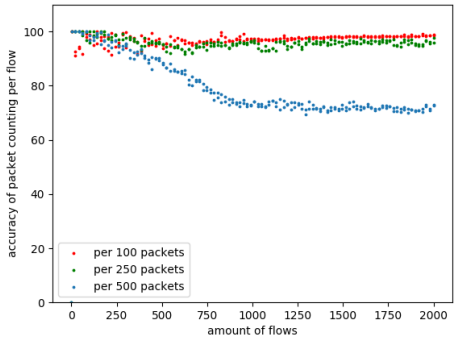

Fig. 7. The accuracy of the bloom filter as function of the decrementing speed while varying the flow amount.

Not surprisingly, thanks to the removal of entries from the bloom filter the accuracy remains very high and it will even stabilize over time.

VII. SYN ATTACKS DETECTION

To validate the use of P4 Bloom filters we chose a concrete scenario. Namely, we chose to focus on the SYN flooding attack, an attack that in the first quarter of 2018 has occurred the most often [10]. In fact, in 2018 57.8% of the DDoS attacks were SYN flooding attacks followed by DDoS attacks focusing on the TCP protocol (14.7%).

A SYN flooding attack is a DDoS attack which abuses the three-way TCP handshake to rapidly fill up the memory of the server. This will eventually lead to a situation where the server cannot handle any new connection and thus starts to block any new client until some memory has freed up. This will prevent legitimate users to start a new communication with the server.

Our P4 implementation will try to detect malicious SYN packets and mark them so that they cannot initiate a handshake and thus preventing that more memory is being wasted. Our implementation relies on a counting Bloom for all the packets of a flow. Our program will compute for every SYN packet that passes through the program the ratio of regular packets per SYN packet. This is done by dividing the total amount of regular packets of that flow by the total amount of SYN packets. If this ratio crosses a certain threshold, we will mark the packet as malicious. Figure 8 shows this marking process in a simplified way.

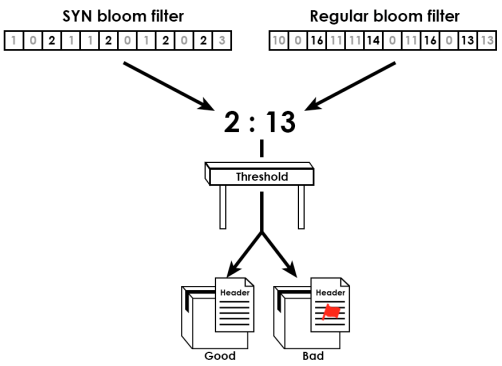

Fig. 8. A simplified visualization of detection and marking of potential malicious packets. This illustration visualizes the process of obtaining the bloom filters values of a packet, then comparing them to each other, checking this ratio against a threshold and lastly based on that result the packet is marked or not

Our approach is similar to the one proposed in the paper written by Changhua Sun, Jindou Fan and Bin Liu [11]. They also suggest to use counting bloom filter: they count the amount SYN, FIN and RST packets and they then make the decision if a SYN packet is malicious based on the rate between SYN and FIN/RST packets. When the traffic is legitimate, the amount of SYN and FIN/RST packets should be in balance.

We instead chose to compare SYN packets to all the other packets; our motivation for this is that an attacker could easily obfuscate its attack by sending a couple of fake FIN or RST packets.

We developed counting Bloom filters it on a software switch using the P4 programming language. One of the bloom filters will count the SYN packets and the other count the other packets in the flow.

The logic that will decide whether a packet is a SYN packet is as follows: we first parse the packet header and then check if the header contains a valid TCP header; we then evaluate the flags parameter inside the TCP header. If the flags parameter is equal to 0x2, i.e. the SYN flag, then the packet will be inserted into the SYN bloom filter otherwise the packet will be inserted into the regular bloom filter.

After the packet has been added to one of the bloom filters, the program will then check for the SYN packets if that specific packet is malicious or not. In order to check the packet, the program first needs to obtain the ratio of regular packets per SYN packets of the flow that the packet corresponds to. This is done by first obtaining the amount of SYN packets that belongs to that flow. This is done by hashing the current SYN packet and finding the corresponding entries in bloom filter. When the entries are available we search for the lowest value between them and use that value as the total amount of SYN packets we have seen for that flow. Note that comparing four variables to each other to find the smallest value is not a trivial process in the P4 language; P4 is only able to compare values to each other in control objects and it can only change a value in the header by performing an action, which can only be performed after a table match. So to solve this problem, the program first obtains all the values from the bloom filter and stores them in the bf_metadata header. It then moves back to the control object where it will compare the values and then call a specific table+match table which will copy the value lowest value to another entry in the bf_metadata so it can be used later in the program. The code in Listing 7 shows how this process is done using the P4 programming language.

```
control handle_syn_flow_decision{
    apply(obtain_syn_flows);
    if(bf_metadata.flw1 <= bf_metadata.flw2)
        apply(use_syn_flow1);
    else
        apply(use_syn_flow2);
}

table use_syn_flow1 {
    reads { ether_hdr.ethertype : exact; }
    actions {
        set_syn_flow1;
    }
}

action set_syn_flow1(){
    modify_field(bf_metadata.syn_flw, bf_metadata.flw1);
}
```

Listing 7. The P4 code that decides what the actual flow amount is for a specific flow. The table 'use_syn_flow1' and action 'set_syn_flow1' are in fact duplicated for the second option but have been omitted in this figure for the sake of simplicity

VIII. Results

A. Variable amount of flows

In our first experiments we checked how the implementation will behave when the same SYN flood attack is performed over multiple concurrent flows. We set the duration of the attackat 60 seconds and the packets will be sent at a rate of 100 packets per second. The parameter that will vary during this experiment is the amount of flows that will perform this SYN flooding attack.
While we executed this experiment, it became clear that we could not perform this experiment for high amount of flows due to hardware limitations, and we limited our study to a maximum of 40 flows. The reason was that when we created even more attacking flows, the sending packet rate would become so low that too many packets are lost in transit. The results are shown in Figure 9.

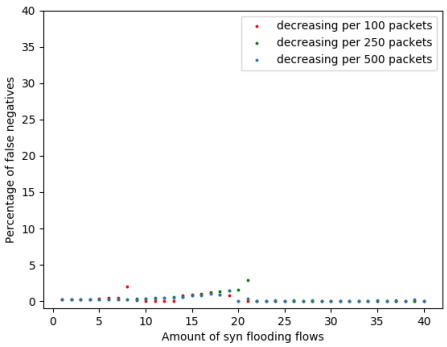

Fig. 9. The percentage of false negatives when performing SYN flooding as function of the decrementing speed while varying the amount of flows

Figure 9 shows that when the implementation gets attacked by up to 40 concurrent flows, we can still detect the attacking SYN packets with a high precision. We expect that when we increase the amount of attacking flows even more, the implementation will start to have some trouble in detecting the malicious packets. The reason for this is that the increasing amount of packets will cause the decrease of the bloom filter values to happen more often. This in turn means that it will take longer for a flow to grow past the thresholds.
False positive in this scenarios are absent; the number of attacking flows is still too small in order to have an impact on the false positive percentages.
As clear from Figure 9 the parameters of our experiment show that the implementation performs well. We repeated the experiment with less ideal circumstances: we used a SYN flooding attack which will attack the server for 20 seconds with a packet sending rate of 20 packets per second. The results of this experiment are shown in Figure 10.

Figure 10 shows that indeed when the circumstances of the SYN flooding attack are less ideal our implementation has more trouble detecting the attacking flows. The graph also

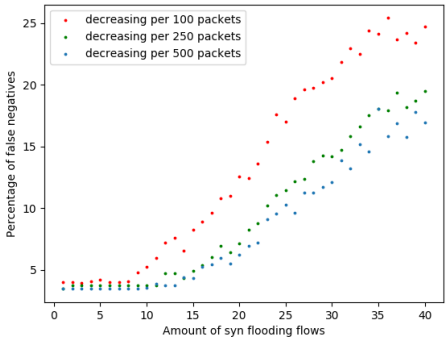

Fig. 10. The percentage of false negatives when performing SYN flooding as function of the decrementing speed while varying the amount of flows

shows that when the decrementing rate becomes lower the percentage of false negatives will also decrease.

B. Protecting a network device

In our second experiment we checked how well our implementation can protect a network device. We assume a server which can only handle 10.000 half open TCP connections at the same time. To check how well the implementation can protect this server, we will be sending the malicious packets at a rate of 10.000 packets per minute over a duration of 5 minutes. We will also send every minute 10.000 malicious SYN packets, which are divided over a variable amount of flows. So we start by using one flow which will send 10.000 packets per minute and in the end we will be using 100 flows, where each flow will send 100 packets each minute. After we performed this experiment we obtained the graph shown in Figure 11. Figure 11 shows that the implementation is very

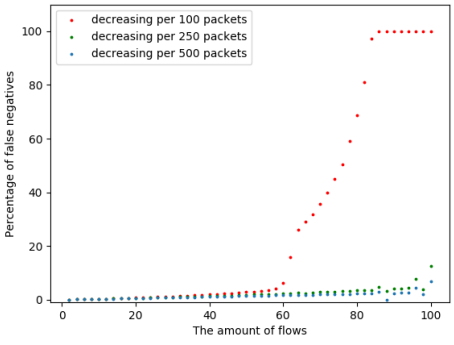

Fig. 11. The percentage of false negatives as function of the decrementing speed while dividing the packets over a variable amount of flows

good at detection SYN flooding attacks that are performed by a small amount of flows. When the SYN flooding attack gets split up over an increasing amount of flows it start to decrease in its precision. This graph also clearly shows the difference between different decrementing rates. For example

the implementation that uses a decrementing rate of 100 packets drops dramatically in precision when its attacked by 60 or more different flows. Also for this experiment the false positive rate was again equal to zero and thus did not show anything noteworthy.

IX. Discussion

Our implementation shows promising results for detecting network flows using P4 Bloom filters. As we mentioned before this is an essential element to track the full path of a flow in a P4 network.

In all the tracking methods we have envisioned, data is retained on the last node along the path. Ideally all of this data is collected at a central point. Bloom filters implemented in the data plane can be used by a node to determine if a flow has been seen previously and therefore should not be sent to the central collection point.

We defined and want to explore three different methods to accomplish path tracking with P4:

- Hop Recording – recording each hop in the packet.
- Logging Forwarding State – logging what forwarding rules are in effect.
- Dynamic Path Labeling – dynamically defining a label for each path used.

In *Hop recording* the path can be captured in the packet itself. As a packet moves through the network each node adds its own node identifier (NID) into the packet. Before forwarding the packet out of the network or to its final destination, the last node on the path would capture and strip the path information from the packet and record it with the flow fields.

This is easily implemented in P4 using header stacks with the control plane providing the NID in a register. The width of the NID field can be optimized based on the number of devices in the network. How NIDs are determined would need to be determined by some control plane function, ensuring that each is unique. When the packet is determine to be egressing through a port identified as external, the header stack representing the path and the flow determining fields are sent to the control plane for logging.

The *Logging forwarding state* is usable in networks running link-state protocols. Each node generates a digest of the link-state database. When the network is converged this digest will be the same on all nodes. As routers do not typically keep old versions of the link-state databases, one copy of each stable link-state database will need to be archived. When a packet first enters a network, it is tagged with the NID and link-state database digest of that router. Intermediate routers check the digest in the packet against their own. If it matches then the packet is forwarded unmodified. If it does not match then the tag is changed to a special value meaning that the packet is not able to be tracked. At the last node, if the digest is still valid then there was no change in the routing as it transversed the network and it can be logged.

The P4 program would check for the existence of a tracking header. If it does not exists it would be created using the NID and digest made available in registers. If it does exist, the value in the packet is XOR'd with the value of the router's digest. A table look up is then performed on the resulting value. A value of zero would leave the packet unchanged. Anything other than zero would result in an action being run that replaces the digest value in the packet with a special value meaning not able to track. On the last router along the path the NID of where the packet entered the network, the link-state database digest, and the flow determining fields are sent to the control plane to be logged.

Dynamic Path Labeling (DPL) borrows from the concepts of MPLS (Multiprotocol Label Switching). Instead of using a label to determine the path, as in MPLS, DPL assigns a label based on the path already taken. The last node on the path records the final label, its local label, and the flow determining fields. This has the effect that at the egress node the single final label can be used to determine the exact path taken through the entire network. To reconstruct the path a flow took through the network, each label is used to lookup the previous label. These look ups are performed recursively moving backwards along the path the packet took until arriving at the node the flow originally entered the network on.

The P4 implementation of DPL would start by checking for the existence of a DPL header. If it does not exists a value of null would be used for the incoming label. A table lookup is done on the combination of ingress port and incoming label to determine if a matching local label already exists. If a local label does exist then the DPL header is updated, being created if necessary. If a matching local label does not exist, the packet, along with meta-data, is sent to the control plane to have a label generated. Once the control plane generates the new local label, the packet, including original meta-data, is resubmitted to the data plane for processing.

X. Conclusions

Our research shows that the ability to implement custom headers and packet processing rules with P4 in a network could ultimately enhance its security. Our SYN attack use case provided evidence of this, and our approach forms the basis for extensions towards flow tracking, as discussed in the preceding section.

However, there are some factors that currently limit the usability of P4. Not many devices currently support P4. This could be because P4 is in active development and vendors are waiting for the language to stabilize before implementing it. P4 also leaves a lot of details unspecified. This will hopefully be addressed by $P4_{16}$ and the specification of a standard library [3, page 9]. Also, there is a need for a standardized interface between the data plane and the control plane. As the programmable data plane becomes more prevalent this is something that is likely to be developed. While P4 in its current state may not be ready for production, it is already clear that there are significant benefits from being able to program the data plane. Finally, we run our implementation on we use a software switch, which is a simulation of a forwarding device. The software switch does not have the same

limitation that a hardware switch would have: in a hardware switch it might not be possible to have a bloom filter with 2500 entries; or, it might not be possible to execute the large amount of matching we use in our implementation. Furthermore by only using the software switch, we were not able to test the latency of our implementation. It might thus be possible that our implementation causes too much latency which makes it not realistic to implement it in the real world. It is thus interesting for future research to check what the impact is of our implementation on an actual forwarding device and it there are ways to optimize the implementation.

Still, we are convinced that supporting a programmable data plane in hardware will allow network security enhancements through customization at a level not previously possible.

ACKNOWLEDGMENT

Initial part of this work were funded by the RoN - Research on Networks- from SURFnet. We are particularly thankful to Ronald van der Pol and Marijke Kaat.

REFERENCES

[1] *NetFlow Performance Analysis*, Cisco Systems, Inc., 2005. [Online]. Available: https://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/secure-infrastructure/net_implementation_white_paper0900aecd80308a66.pdf

[2] P. Bosshart, D. Daly, G. Gibb, N. M. Martin Izzard, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," July 2014. [Online]. Available: https://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf

[3] *P416 Language Specification*, The P4 Language Consortium, May 2017. [Online]. Available: https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[4] *The P4 Language Specification*, The P4 Language Consortium, May 2017. [Online]. Available: https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[5] P4 Language Consortium. Code. https://p4.org/code/. Online; Accessed May 2017.

[6] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004. [Online]. Available: https://doi.org/10.1080/15427951.2004.10129096

[7] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[8] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[9] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*. Springer, 2006, pp. 684–695.

[10] O. K. Alexander Khalimonenko and E. Badovskaya, "Ddos attacks in q1 2018," 4 2018. [Online]. Available: https://securelist.com/ddos-report-in-q1-2018/85373/

[11] C. Sun, J. Fan, and B. Liu, "A robust scheme to detect syn flooding attacks," in *Communications and Networking in China, 2007. CHINACOM'07. Second International Conference on*. IEEE, 2007, pp. 397–401.