University of Amsterdam

System & Network Engineering

Securing Networks with P4

Joseph Hill, Paola Grosso

University of Amsterdam

February 14, 2018

Contents

1 Introduction

Conventional methods of tracking the flow of data through a network can be resource intensive. This is especially true when the examining of each packet has to be done by the control plane. Resource utilization 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[5], this is where instead of inspecting every packet, every N^{th} packet is inspected. 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.

1.1 NetFlow

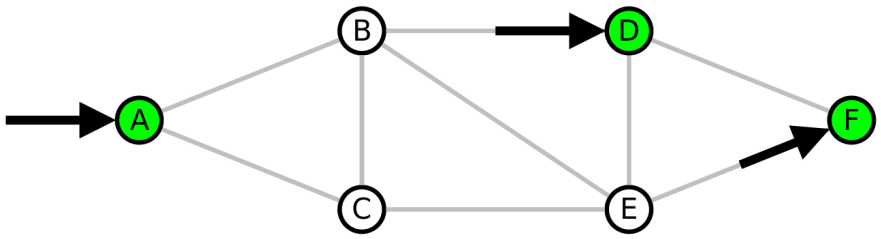

Figure 1: Example of an ambiguous path due to sampled NetFlow.

NetFlow is a commonly used approach to logging traffic in a network. NetFlow correlates traffic in to what it considers a flow based on certain predefined fields. Which fields are used depends on the version of NetFlow. While some devices can run NetFlow without sampling, sampling is often necessary when there are limited system resources available. Previous work with the CoreFlow framework found that sampled NetFlow in some circumstances did not provide sufficient information to identify the complete path of malicious traffic[4]. This can occur do the nature of sampled NetFlow. When sampling is used some devices along the path of a flow may not record it. This is especially a problem with short lived flows which may be missed in their entirety. Figure 1 demonstrates how sampled NetFlow may result in ambiguous information about the path of a flow. In the figure a flow transverses the network from node A to F. Due to sampling, the green nodes are the only ones that capture this flow. The version of NetFlow used here captures the ingress port but not the egress port. From the data logged it is not possible to tell the exact path of the traffic. For instance, between nodes A and D the flow may have transversed just node B or C then B. When load balancing over multiple paths, the problem is exacerbated. It would then be possible that some of the traffic from the flow went directly from D to F while some traffic also used a path from E to F without traversing D.

1.2 P4: Programming Protocol-Independent Packet Processors

P4 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. This research looks at how P4 can be used to help secure networks by enabling the tracking of the complete path of data as it moves through the network. Specifically, P4 solutions to tracking the path of data will be compared with how NetFlow is used to track flows with the goal of maximizing efficiency and accuracy.

2 P4 Language Structure

A P4 program starts by parsing a packet based on user defined headers. The protocol independent nature of P4 means that theses headers could 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 [6, 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[8, 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[8, pages 61-62].

There are two version of the P4 language specification. This research focuses on $P4_{14}$ which is the older version of the language but is currently better supported. $P4_{16}$ is the latest version of the language. It makes significant changes to the language 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[6, 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 $P4_{14}$.

3 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. The following P4 based solutions will look at a more cooperative means 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. The following three methods will explored:

- 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.

3.1 Hop Recording

Instead of keeping track of flows in each device as NetFlow does, the path can be captured in the packet itself. In this method of hop recording, 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 would not require any state to be kept in intermediate nodes. The logging information only exits on the edge nodes and is not duplicated. Only the last node even needs to do the work of parsing out the fields that identify the flow. To recreate the path only the egress node needs to be found which will contain the complete path. This is not a novel solution to recording the route of traffic. The Internet Protocol specification[7] has facilitates to do just this. Also a newer proposed method of recording network state in packets, In-band Network Telemetry (INT)[3], has facilities for recording each hop. The downsides of this approach are that a variable amount of data is added to each packet causing the packet to grow in size. This makes it difficult to know what maximum transmission unit (MTU) a sending device should use. Also, additional information may be necessary to resolve ambiguity when multiple links exist between the same two nodes.

This method is well suited to spine and leaf topologies where the majority of traffic is internal. In this topology traffic between two nodes has multiple equal length paths it could take. Here the amount of data added to each packet is limited and more predictable. Figure 2 shows a simplified spine and leaf network where nodes X through Z make up the spine and nodes A through F are the leaves. There are three different paths that could be taken between nodes A and E, but they are all exactly three hops.

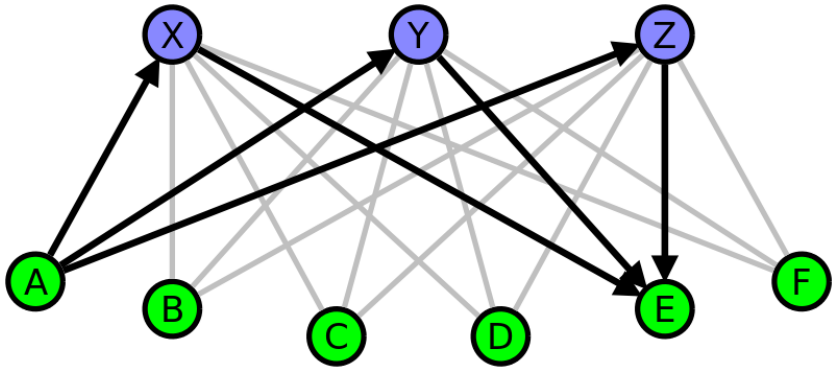

Figure 2: Paths in a spine and leaf network.

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.

3.2 Logging Forwarding State

This method is specific to certain IP networks. With a link-state routing protocol such as OSPF, each device knows enough about the network to determine how every other device will forward packets. Whenever the routing protocol is converged, all routers contain the same information in their link-state database. This means that whenever the network is converged, if given where a packet enters a network, the destination IP, and the link-state database, it is possible to determine the exact path the the packet will take through the network. This method uses this property of link-state routing protocols to determine the path of traffic. 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. If an invalid routing database digest is encountered then the path can not be determined. The last device logs where the packet entered the network, the routing database digest, and the flow determining fields. To later determine the path of a flow, the digest that was logged is used to look up the actual link-state database from the archive. That is then used to calculate the routing rules that were in place when the packet was being forwarded through the network. This information along with where the packet entered the network allows the complete path to be calculated.

This method is only viable if a link state routing protocol is in use and is the only factor

determining the routing of packets. If other routing information sources, such as static entries, can override the routing protocol then it will not be possible to determine the exact path from the link-state database alone. Also, when the network is not converged, inconsistent routing information exists in the network and packets may not be able to be tracked. So this is only practical in stable networks without frequent routing database changes. The amount of memory required to archive link-state databases must also be considered. Routing protocols that allow load balancing over multiple paths may be problematic depending on the load balancing algorithm in use. For instance if round robin load balancing is used then it would not be possible to determine which path a particular flow took. If load balancing is done based on characteristics of the packet then it may be possible to determine the path if the same fields are recorded as part of the flow data.

The P4 implementation of this method would require some work to be done outside of the P4 program. Each time the link-state database is updated a new digest must be calculated. The control plane would then need to expose this digest via a register. At least one node would also need to archive the link-state database. 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.

3.3 Dynamic Path Labeling (DPL)

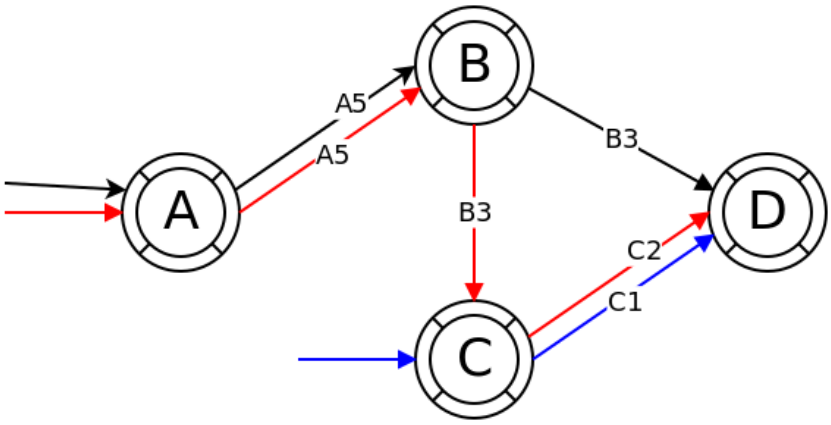

Figure 3: Labels being generated in DPL network.

This method borrows from the concepts of MPLS (Multiprotocol Label Switching). In MPLS labels are used to determine the path of a packet. This occurs at both the device and the network level. Each device forwards a packet based on the label it has and updates that label as it is forwarded. While labels are changed as a packet moves through the network, the initial label given to a packet effectively determines the complete path it will take through

the network. Instead of using a label to determine the path, this method assigns a label based on the path already taken. As shown in figure 3, each device creates a label for each unique combination of incoming label and ingress port that it sees. When there is no incoming label a null value is used. The label created incorporates the local NID to make it unique in the network. A table correlating incoming labels and ingress interface with local labels is maintained. Tables 1 to 4 show the corresponding DPL tables generated on each node. When forwarding a packet it is updated with the local label. 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 ingress interface can be used to resolve ambiguity when multiple paths connect two nodes. For example, from the final label D9 the label history of C2, B3, A5 could be determined, meaning the path of the flow was A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D.

Table 1: DPL table on node A.

Incoming Label	Ingress Interface	Local Label
Null	West	A5

Table 2: DPL table on node B.

Incoming Label	Ingress Interface	Local Label
A5	West	B3

Table 3: DPL table on node C.

Incoming Label	Ingress Interface	Local Label
Null	West	C1
B3	North	C2

Table 4: DPL table on node D.

Incoming Label	Ingress Interface	Local Label
B3	West	D4
C1	West	D7
C2	West	D9

This method is able to follow traffic no matter what path it takes through the network. Routing changes and load balancing over multiple links do no affect the ability to log the path of traffic. Even loops are possible to track. The major drawback of this method is that when a packet arrives with an previously unseen label a local label must be generated prior to forwarding the packet. This means that the first packet along each unique path will incur increased latency. This delay is also cumulative since there will be a delay on each node from where the new path diverges from known paths. Note that this delay only applies to the first packet on that path, and not to the first packet of each flow. A new flow that transverses a

know path will not incur any extra latency. It is expected that there will be far fewer unique paths in a network then there are unique flows.

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. The performance of the communication channel between the data plane and control plane and the control plane's ability to add an entry to the table will largely determine the delay the packet receives. However, this should not be an operation that happens frequently as it is only necessary the first time a new path is being transversed. On the last device along the path, after updating the label, the final label and the flow determining fields are sent to the control plane to be logged.

4 Bloom Filters with P4

In each of the path logging methods previously described, the data is retained on the last node along the path. Ideally all of this data would be collected at a central point. The last node would either need to send logging data every time a packet is seen, even when it is from the same flow, or it would need to keep a list of flows seen so that it would know when not to send redundant data. Alternatively Bloom filters could be used. A Bloom filter implemented in the data plane allows additional state to be tracked without involving the control plane. They 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 [2]. A Bloom filter is implemented with an array of bits and a predetermined number of hash functions. 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.

4.1 Parameters

The performance of a Bloom filter can be though 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.

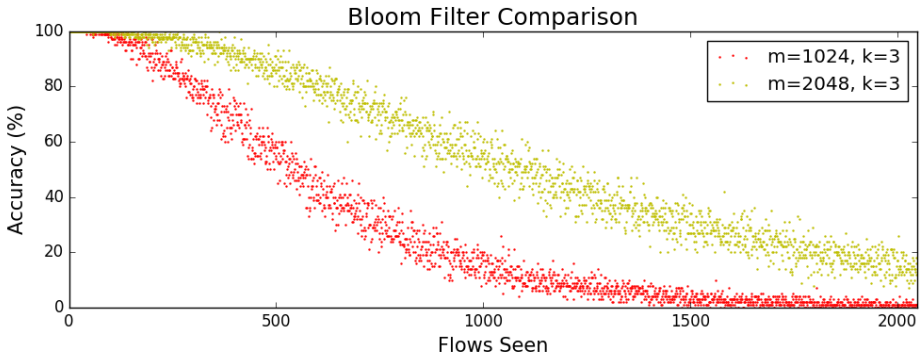

Figure 4: Bloom filters with different array sizes.

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 4 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 5 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[2]. 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)[1].

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

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

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 6 shows the performance of this Bloom filter in red. In blue the cumulative accuracy is also shown, which

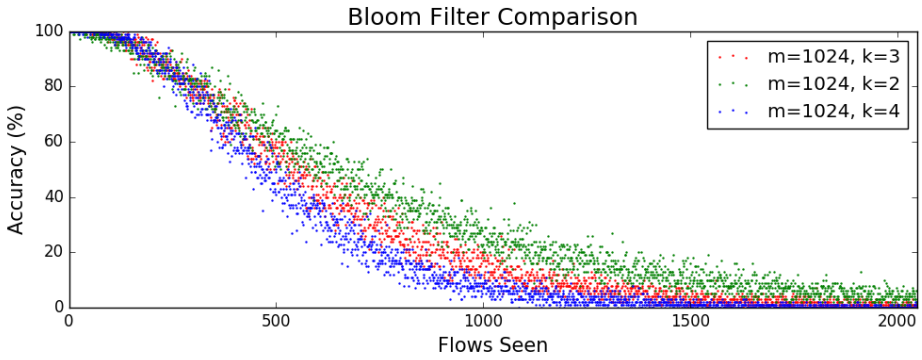

Figure 5: Bloom filters with a varying number of hash functions.

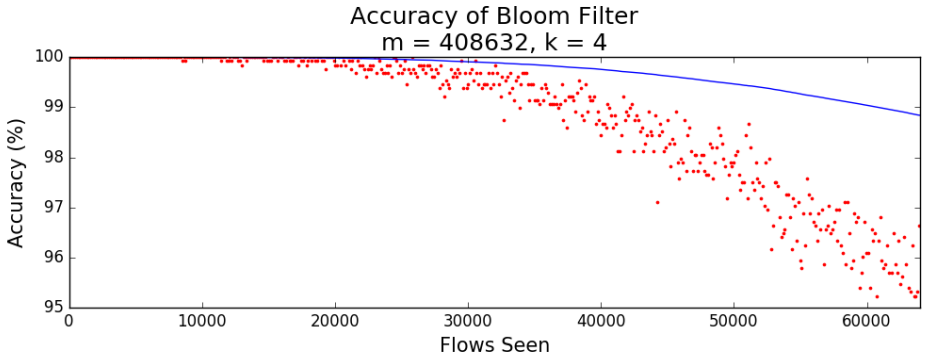

Figure 6: Bloom filter tuned for performance with cumulative average..

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.

4.2 P4 Implementation

The properties of Bloom filters make them well suited to be implemented in P4 without needing to interact with the control plane. The size of the Bloom filter array is hard coded in the P4 program. The bit array can then be implemented as a register.

```
#define BF_WIDTH 408632

register bloom_filter {
    width : 1;
    instance_count : BF_WIDTH;
}
```

In order to simulate havening multiple hash functions, a salt field is added to the hash calculation and a different salt value is used for each run of the hash calculation. The number of hash calculations to perform, and the salt values are also hard coded in the P4 program. The output of each hash function is modulated to get an index into the array. When adding a flow to the set, the corresponding cells of the array are set to 1. Since cells in the array are only ever changed from 0 to 1 there are no issues with concurrency.

```
#define BF_SALT1 0x7742
#define BF_SALT2 0x2eb7
#define BF_SALT3 0xf3b4

action bf_insert() {
    bf_set_bit(BF_SALT1);
    bf_set_bit(BF_SALT2);
    bf_set_bit(BF_SALT3);
}

action bf_set_bit(salt) {
    modify_field(bf_metadata.salt, salt);
    modify_field_with_hash_based_offset(bf_metadata.index, 0, bf_hash_func, BF_WIDTH);
    register_write(bloom_filter, bf_metadata.index, 1);
}
```

To query whether a flow is in the set, a 'in_set' flag is initialized to one. The cells corresponding to indexes derived from the hash functions are then each read and a bitwise AND operation is performed with the 'in_set' flag. If the flag is still one after all hashes have been checked the flow has been seen previously. This flag can then be used later to apply different handling to this packet.

```
action bf_query() {
    modify_field(bf_metadata.in_set, 1);
    bf_check_bit(BF_SALT1);
    bf_check_bit(BF_SALT2);
    bf_check_bit(BF_SALT3);
}

action bf_check_bit(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.bit_set, bloom_filter, bf_metadata.index);
    bit_and(bf_metadata.in_set, bf_metadata.in_set, bf_metadata.bit_set);
}
```

5 Conclusion

This research shows a number of scenarios where the ability to implement custom headers and packet processing rules in a network can enhance the security of that network. Being

able to program the data plane allows this to be done while maintaining a high level of performance. This is most useful when all the devices in a network are able to cooperate. The P4 specification provides a standard way of implementing this functionality in forwarding devices. 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[6, 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. Supporting a programmable data plane in hardware allows network security enhancements through customization at a level not previously possible.

References

[1] Flavio Bonomi et al. "An Improved Construction for Counting Bloom Filters". In: *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*. ESA'06. Zurich, Switzerland: Springer-Verlag, 2006, pp. 684–695. ISBN: 3-540-38875-3. DOI: 10.1007/11841036_61. URL: http://dx.doi.org/10.1007/11841036_61.

[2] Andrei Broder and Michael Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In: *Internet Mathematics* 1.4 (2004), pp. 485–509. DOI: 10.1080/15427951.2004.10129096. URL: https://doi.org/10.1080/15427951.2004.10129096.

[3] *In-band Network Telemetry (INT)*. The P4.org Applications Working Group. Dec. 2017. URL: https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[4] Ralph Koning et al. "CoreFlow: Enriching Bro security events using network traffic monitoring data". In: *Future Generation Computer Systems* 79 (2018), pp. 235–242. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0167739X17305952.

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

[6] *$P4_{16}$ Language Specification*. Version 1.0.0. The P4 Language Consortium. May 2017. URL: https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[7] Jon Postel. *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://rfc-editor.org/rfc/rfc791.txt.

[8] *The P4 Language Specification*. Version 1.0.4. The P4 Language Consortium. May 2017. URL: https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf.