Secure in-packet Bloom Filter forwarding on the NetFPGA

Adnan Hassan Ghani
Ericsson Research, NomadicLab
02420 Jorvas, Finland
adnan.ghani@ericsson.com

Pekka Nikander
Ericsson Research, NomadicLab
02420 Jorvas, Finland
pekka.nikander@ericsson.com

ABSTRACT

In-packet Bloom filters allow one to forward source-routed packets with minimal forwarding tables, the Bloom filter encoding the identities of the links the packet needs to be forwarded over. If the link identities are made content dependent, e.g. by computing the next-hop candidate link identifiers by applying a cryptographic function over some information carried in the packet header, the Bloom filters differ pseudo-randomly from packet-to-packet, making the forwarding fabric resistant towards unauthorised traffic.

In this paper, we describe our early implementation and testing of an in-packet Bloom filters forwarding node that implements cryptographically computed link identifiers. We have tested two different cryptographic techniques for the link-identity computation and thereby for making the forwarding decision. The algorithms have been implemented and tested on the Stanford NetFPGA. The performance and eifficiency of the algorithms is also briefly discussed.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Design

Keywords

Bloom filters, forwarding, multicast, security, denial-of-service resistance, NetFPGA

1. INTRODUCTION

While Bloom filters [2] are commonly used in several roles in networking applications [3], in-packet Bloom filters have only recently gained more attention [6, 11]. The basic idea in these works is to encode the packet path (or a multicast tree) into a small Bloom filter, carried in the packet header. In the approach by Jokela *etal.* [6], each network link was expected to have been assigned a statistically unique, unidirectional link identifier. A set of these link identifiers, forming the path or the tree, was then encoded into the in-packet Bloom filter. This basic method was implemented on the NetFPGA by Keinanen *etal.* [7]

In a followup paper, Esteve, Jokela, *etal.* [10] introduced an idea where the link identifiers computed dynamically. That is, instead of storing the names (or Bloom masks) of the outgoing links at a forwarding table, the forwarding node would dynamically compute the outgoing link identifiers. If the computation uses some information from the packet, the link identifiers, and thereby the in-packet Bloom filters, may be made flow or packet contents dependent. As a consequence, only authorised users, which have the required input parameters for sending packets along a specific path, are able to compute the appropriate in-packet Bloom filters for any given path. Hence, the method very effectively block unauthorised traffic, at the cost of parameter distribution. Gathering the input parameters for a source route and fast rerouting are out of scope for this paper.

From the security point of view, the in-packet Bloom filters act simultaneously as forwarding identifiers and forwarding capabilities [10]; introducing a DoS resistant forwarding service. Capabilities enable secure statements attached to packets, allowing forwarding nodes to easily check if a packet has been approved by the receiver. Any sender that has the approriate input parameters is able to compute the in-packet Bloom filter, encoding a number of dynamically computed link identifiers. When such a packet then arrives at a forwarding node, the node computes a number of candidate Bloom masks (i.e. link identifiers), using a loosely synchonized time-based shared secret and additional in-packet per-flow or per-packet information. The forwarding capabilities are thus expirable and packet or flow dependent. They do not require any per-flow network state or memory look-ups, at the cost the additional per-packet computation.

While expected to be secure, the performance of the dynamic-link-identifiers-based forwarding method needs to be checked in real hardware. The delay and the resource usage in the forwarding node needs to be examined. In this paper, we present our early results from implementing the dynamic-link-identifiers-based forwarding method on the NetFPGA, and report our early results on its applicability and performance.

The rest of this paper is organized as follows. First, in Section 2, we discuss the background and the problem. In Section 3, the design and architecture is discussed with two different possibilities. Section 4 describes the implementation details. Section 5 explains the evaluation and performance of our design. In Section 6, we briefly discuss the related work and Section 7 concludes the paper.

2. BACKGROUND

Our ultimate goal is to provide a generic forwarding mechanism where the forwarding identifiers are secure [10]. We

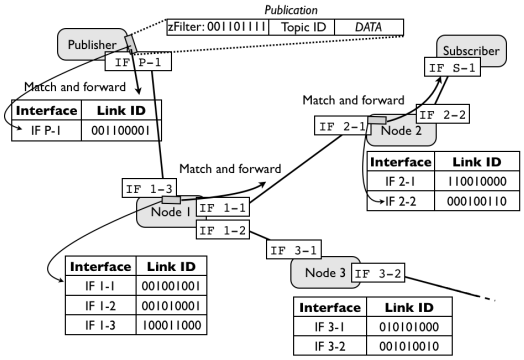

Figure 1: Example of link identifiers assigned for links, as well as a sending data with a zFilter from the Publisher to the Subscriber [6] (copied with permission)

presume that, in general, such a mechanism could be used for any source-routed system, including source-routed IP packets; however, the system we describe is more tailored for Denial-of-Service (DoS) resistant information-centric networks [13].

We build upon the forwarding mechanism described by Jokela *etal*. [6]. The basic idea is to do forwarding based on identifying links instead of nodes. As a simple case, consider a point to point case, where two nodes are connected with a single bi-directional link. This link will have two separate identifiers, each identifying the specific direction of packet flow. When the same case is mapped to multi-point scenario, statistically unique identifiers are assigned to all of the links.

When the link identifiers are encoded into a Bloom filters, they first need to be converted into a Bloom mask. The standard way for that is to compute k distinct hash functions over the identifier, defining k bit positions that are then set to 1 in the Bloom mask. However, for the simplicity of handling, for the most part we ignore this step and consider the link identifiers to be in the Bloom mask form already from the beginning.

Hence, for a (k, m) Bloom filter scheme, the length of each link identifier (Bloom mask) is m-bit, in which k bits are set to 1, with $k \ll m$. For example, if $m = 256$ and $k = 5$, the number of unique link identifiers $\approx m!/(m-k)!k! \approx 10^{10}$.

As in [13], we further assume a topology system which keeps track of forwarding nodes in the network and the identifiers of the links interconnecting the nodes. The system also keeps track of the potential senders and receivers in the network. Using this information, the topology system may create a graph representation of the network, with the edges annotated with the the link identifiers. This can then be used when packets need to be forwarded from a sender to a receiver (or group of receivers).

In [6], the topology system encodes the link identifiers along a path (or a tree) into a Bloom filter, and then gives the Bloom filter to the source node, which then places it into the packets; see Figure 1. In their work, Jokela et al denote the in-packet Bloom filter as a *zFilter*, a convention that we adopt below. It is good to notice that zFilters provide build-in DoS resistant capabilities.

When a packet reaches a forwarding node along the path, each candidate outgoing link identifier (Bloom mask) is AND-ed with the zFilter carried in the packet, and the result is compared with the link identifier. If there is a match, the packet is forwarded along the path; conversely, if there is no match, the packet is not forwarded along the link associated with that particular candidate outgoing link identifier. Furthermore, if there are matches with multiple candidate identifiers, the packet is by default forwarded along all of the matching links, thereby providing support for multicast; see Algorithm 1.

Algorithm 1: Forwarding method of LIPSIN

Input: Link IDs of the outgoing links; zFilter in the packet header

```
foreach Link ID of outgoing interface do
    if (zFilter & Link ID) ≡ Link ID then
        Forward packet on the link
    end
end
```

As usually with bloom filters, with the increase in the number of links encoded into a zFilter, there arises a possibility of false positives. While the Link ID Tag mechanism, introduced in [6], may be used to squeeze in some more link identifiers without causing too many or too bad false positives, the number of links within a zFilter has always a practical upper bound that depends on m, the length of the zFilter. They also describe a (perhaps somewhat hacky) method for overcoming this limitation, and also discuss a number of other practical considerations, such as loop prevention.

From security point of view, this basic mechanism is susceptible to a number of attacks. For example, an attacker may try to collect zFilters and guess a forwarding identifier based upon such collected information. That is, by analysis of zFilter bit patterns, an attacker may determine the probabilities of what bits are set to one on which partial graph. With having a large number of zFilters, source and sinks' information, an attacker may have success in constructing a valid zFilter.

In this paper, we explore some solutions to this security problem, and evaluate their performance penalty.

3. ARCHITECTURE

In this section, we present the details of the secure and dynamic link-identity-based forwarding approach. In particular, we briefly describe the two different cryptographic functions that we have used, one based on a self-synchronising stream cipher and the other on the standard AES block cipher. Our initial assumption was that using a self-synchronising stream cipher could allow use to perform more parallelised and thereby faster forwarding decisions. That assumption turned out to be false, for a number of reasons.

3.1 Secure In-Packet Bloom Filters

In essense, we have implemented the central ideas of the Esteve at paper [10]. Our solution dynamically computes the link identifiers on the basis of packet contents, the path the packet takes, and the node keys. In the following, we will use the term *zFormation* to designate our design, i.e. basically the dynamic computation of the link identifiers on per-packet packet-basis. The idea is that there is a function Z, essentially evaluated for each packet and for each poten-

tial outgoing interface, that gives out the indices of the k bits set to one in the m-bits long link identifier (Bloom mask).

The function Z can be defined as

$$O = Z(K, M, I) \quad (1)$$

wher the input parameters of Z are defined as follows:

1. K is a secret key that is changed periodically, e.g. once every few minutes, hours, or days.
2. M is a medium dynamic term that includes the incoming and outgoing interface indices.
3. I denotes some in-packet information that varies per packet, e.g. a counter that increases per packet basis.

For performance purposes, the key K is divided into three cryptographically separated parts, K_1, K_2 and K_3 which are created using a standard key derivation function:

$$K_i = KDF(K, L_i) \quad (2)$$

where the term L_i is a literal identifying the particular key.

The Key Derivation Function (KDF) is used to compute the three keys. These keys are used in the construction of zFormation in the following manner:

$$O_1 = F_1(K_1, S) \quad (3)$$

where S denotes any (semi-)static inputs to the function.

$$O_2 = F_2(K_2, O_1||M) \quad (4)$$

where || denotes concatenation. Furthermore, if there are multiple potential actively valid values for M, it may be necessary to precompute and cache a set of corresponding O_2 values. We call such a set of O_2 values as O_2 *value set*.

$$O = O_3 = F_3(K_3, O_2 \otimes I) \quad (5)$$

where $\otimes$ denotes exclusive OR.

The functions F_1 and F_2 can be computed off-line, before packet processing, using a strong algorithm, e.g. HMAC-SHA-256. The function F_3 needs to be performed on a per-packet bases, and thereby represents a compromise between security and performance. We use per-packet information, as an input value to the hash function, to make it infeasible to send other packets using an eavesdropped Bloom filter. That is, an active attacker may capture some packet and replay them a number of times, until one of the node keys is changed, but the attacker cannot send modified packets. When combined with per-packet caching or fingerprints, this prevents replay-based DoS attacks.

In this paper, we consider two constructions, using a self synchronising stream cipher and a block cipher function. The in-packet information I can be formed, for example, by using a packet a counter that is incremented once per packet, and then taking a cryptographic hash overc counter, using HMAC-SHA-256.

3.1.1 Self Synchronising Stream Cipher

A self-synchronising stream cipher consists of three main components, an initialising function, an output generating function Fc and a state update function. First, a state is initialised, typically S_o = 00...0 or some other fixed value is used. The decryption of some stream of bits C_0, C_1, ... proceed as follows

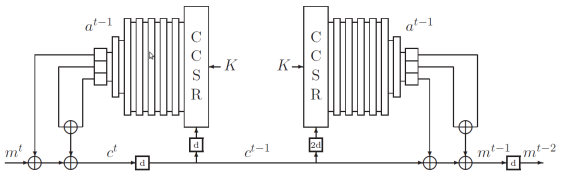

Figure 2: The Moustique stream cipher

Algorithm 2: The algorithm

foreach i **do**
  $P_i = C_i$ XOR Fc (K, S_i)
  $S_{i+1} = C_i$ || $(S_i \rightarrow_1)$
end

where $\rightarrow_i$ denotes right shift by i bits and || concatenation. Here, our F_3 corresponds to the function fc and K_3 corresponds to the key K, etc. The idea behind using a self-synchronising stream cipher is that if the synchronisation is lost, the state S_i will eventually recover as it is filled up again by received bits C_j. However, in our case we don't really need this property.

For us, the important property is to get fast and securely the number of bits that we need to determine the k bits needed for forming the outgoing link identifier. Self-synchronising stream ciphers have the nice property that they output bits on just a couple clock cycles after having been fed in the input. Since they usually work on a single-bit bases, they are fine for line-speed processing.

Unfortunately, the NetFPGA reference router pipeline processes bits in 64-bits words, thereby foiling at the NetFPGA some of the nice properties. To alleviate this, we plan to unroll the stream cipher in the future, aiming towards getting more than one bits out in a cycle. However, in this paper we only report the basic implementation, using the Moustique self synchronising stream cipher.

3.1.2 Moustique

Moustique is a single-bit self-synchronising stream cipher [5]. The Moustique cipher function is implemented in the form of stages. A conditional complementing shift register (CCSR) is followed by seven pipelined stages, as illustrated in Figure 2. The left half shows encryption and the right half shows decryption[1].

3.1.3 AES Block Cipher

A block cipher operates on fixed length bits called blocks. The reason behind choosing block cipher is as we have data coming in the form of 64-bits blocks in the NetFPGA. AES

[1]It must be noted that historically it has been hard to design fully secure self synchronising stream ciphers; therefore, the security of Moustique is perhaps not fully understood yet. On the other hand, for this particular application, we don't need the full power of typical cipher functions, but only a function that generates fast pseudo-random indices out of in-packet input data.

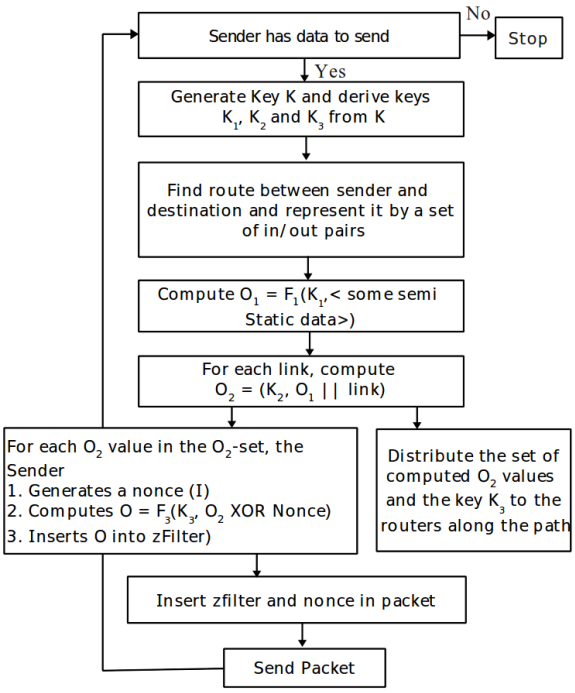

Figure 3: Flow diagram for sender's operation

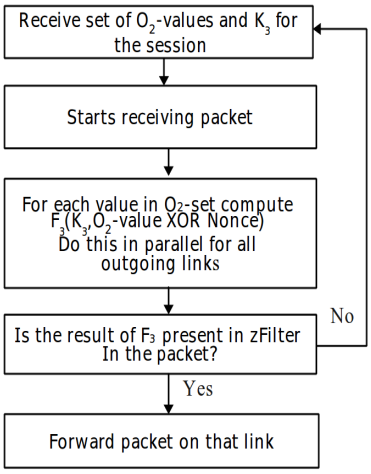

Figure 4: Flow diagram for forwarding operation

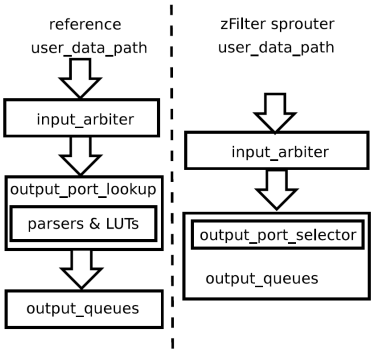

Figure 5: The reference switch and modified datapaths

was chosen as a cipher function. It has a fixed block size of 128 bits with key sizes of 128, 196 and 256. The minimum acceptable key size is 128 and can then be multiples of 64 bits. Further detail on AES can be found in [1].

3.2 Overall Flow of the design

The overall flow of the design is depicted in figures 3 and 4. Figure 3 shows the flow at the sender side and Figure 4 shows the flow at the forwarding node side.

At the sender side when sender has data to send, it generates keys K_1, K_2 and K_3. The route between sender and the receiver is find out and represented in a form of in/out pairs. O_1 and O_2 are computed using the corresponding keys K_1 and K_2, respectively. K_3 and the O_2 value set are distributed among the forwarding nodes on the path. For each O_2 value in the set, sender generates a nonce I and computes F_3. Inserts the result O, in the form of a Bloom mask, into the Bloom filter carried in the packet.

At the forwarding node side, it already receives O_2 value set and K_3. When it receives packet, it retrieves the nonce I from the packet and performs F_3 for each outgoing interface in parallel, giving out the candiate outgoing Bloom mask for each outgoing link. Using these Bloom masks as link identifiers, the node then implements the "usual" in-packet Bloom filters based forwarding (as described in [7]), to check whether the Bloom mask is present in the Bloom filter or not. If present, the packet is forwarded along the path; otherwise it is dropped.

4. IMPLEMENTATION

In this work, we have taken the previous implementation of in-packet Bloom Filter (iBF) based forwarding node [7] on a NetFPGA and optimized it to our needs, as explained below. The basic forwarding method remains unmodified. The difference here is instead of having fixed Link IDs (or Link ID Tags - LITs) we have dynamically computed identifiers of the links, on a per-packet basis.

The dynamic Link IDs are computed using zFormation. We have two implementations for the computation. One is using the Moustique stream cipher and the other one is using AES block cipher. In each case, the Link IDs are computed using i) In-packet information (I), ii) a periodically changing key (K_3) and iii) the outgoing interface index (O_2). The forwarding decision is simple binary AND and comparison operations, for the in-packet Bloom filter and the computed Link ID.

4.1 Forwarding Node

The adopted implementation utilizes only a limited set of modules from the Stanford reference switch, without modifying the rest as shown in figure 5

The main functionality is implemented in module called `output_port_selector`, where the forwarding decision takes place and the packet is placed on the correct output queue. For computing the dynamic Link IDs, a separate module `moustique` is implemented for the stream cipher, and `aes_cipher_top` for the block cipher. For each link, these modules are instantiated, in parallel with the `output_port_selector` module. The basic structure of `output_port_selector` is

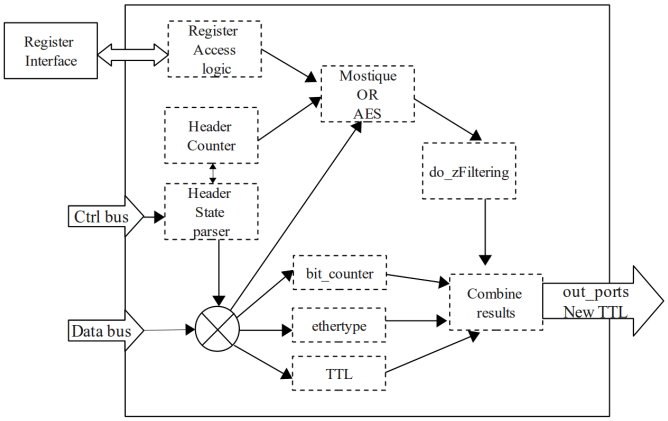

Figure 6: output_port_selector module structure

depicted in Figure 6.

4.2 Packet Forwarding Operations

Prior to sending packets, the computed key K_3 and the O_2 value set are written into the NetFPGA registers from the user space. In our current implementation, the key K_3 and the O_2 values in the value set are both 256 bits each, as a result of HMAC-SHA-256 computed at the software side. In our current design, O_2 consists of four values, each representing one port of NetFPGA. One 32-bit port is used for writing these values into the registers.

Packets arrive in 64-bit pieces at each clock cycle. From the `input_arbiter` module, packets are sent into the SRAM and to the `output_port_selector` module, for computing the dynamic link IDs and then taking the forwarding decision. Along-with the forwarding decision that takes place in the `do_zfiltering` logic block, the three parallelized operations take place for the packet goodness verification, i.e. `bit_counter`, `ethertype`, and `TTL` checks; see [7] for the details.

For computing the dynamic Link IDs using F_3, two separate implementations were made, one for the Moustique stream cipher and the other for AES block cipher. Dynamic Link ID for each port is calculated separately using Moustique and AES. Both of them are explained as follows:

Moustique. As the 96 bits of K_3 register gets value from the user space, a control signal `start_initialization` is set to 1 and the initialization vector bits are applied at the cipher input for 105 clock cycles. The implementation then goes to a hold state, until a packet arrives. In the packet header, there comes the in-packet information (I). I is XORed with each value in the O_2 value set (currently one for each port) and then applied to the cipher input of the Moustique module, with a control signal `start_mostique` set to 1. Moustique is instantiated four times, oncefor each outgoing port, for the NetFPGA in a parallel manner.

Our current Moustique implementation performs the ciphering in a single-bit fashion. Hence, the number of clock cycles to perform whole ciphering depends upon k (cf. section 2). In our case, when $k = 5$ and $m = 256$, we need to perform only 40-bits of decryption, and hence it will take 40 clock cycles with the current implementation. With unrolling, we expect to get this down to maybe 5 cycles, depending on the details of the propagation delays.

As soon as the decrypted data is ready, `decrypted_data_ready` is set to 1 by `mostique` for one clock cycle, so that decrypted data can be read by `output_port_selector`.

AES. AES block cipher, with the key and block sizes of 128, is used for the computation of F_3. We used the OpenCores AES implementation[2]. As the in-packet information I arrives, it is XORed with the values in the O_2 value set (one for each port). The data and the key K_3 are loaded into the input of the cipher function, and `start_AES` is set to 1. AES is also instantiated four times for each outgoing port of the NetFPGA in aparallel manner.

The AES block cipher performs complete encryption sequence in 12 clock cycles, where the initial key expansion takes 1 clock cycle, 10 rounds take 10 clock cycles, and the output stage takes 1 clock cycle.

As the encryption is finished, `decrypted_data_ready` is set to 1 and the `output_port_selector` reads the encrypted data.

The Bloom mask is then computed for each outgoing link. As in our case $m = 256$, each 8-bits of the decrypted data, from Moustique or AES, gives an index in Bloom mask where a 1 should be written.

In `do-zfiltering`, the actual matching is done for each outgoing link. For each interface, we have a single bit forming a bit-vector. These bits are set to 1 initially. Matching is done for each Bloom mask and in-packet Bloom filter (iBF) using AND and comparison operations. If there is a mismatch for a particular link, the corresponding bit gets zero. At the end, when the matching is finished for each Bloom mask, the bit vector shows the interfaces to forward the packet. Wherever we have one in the bit vector, the packet is forwarded on that interface. But still we have some other checks from the three verification functions.

The `bit_counter` module counts number of ones in the iBF. This is done to avoid attacks of setting all bits to one in iBF. The maximum allowed number of ones in a iBF is a constant value. If the iBF contains more number of ones than the constant value, the packet gets dropped. This module is implemented using only wires and logic elements. It takes 64 bits input and returns number of ones. It means for 256 bits iBF it takes 4 clock cycles to count the number of ones.

Currently, our iBF-based packets are identified using `0xacdc` as the ethertype. This is checked upon the arrival of the packet. TTL is also checked to avoid loops in the network.

If any of the three verification checks or the iBF matching itself fails, the packet is dropped. All the operations are shown in reference to clock cycles in 7 and 8, for Moustique and AES respectively.

4.3 Management Software

From the user space, the management software computes keys K_1, K_2 and K_3 using HMAC-SHA-256. Similarly, defining outgoing interfaces and then computing O_1 and the O_2 value sets using HMAC-SHA-256. It writes key K_3 and O_2 value set into the registers in NetFPGA card. At the software side it also computes F_3 by using K_3, O_2 and generates nonce I for each packet. This is done for each interface and then iBF is computed and packed into the packet.

[2]http://opencores.org/project,aes_core

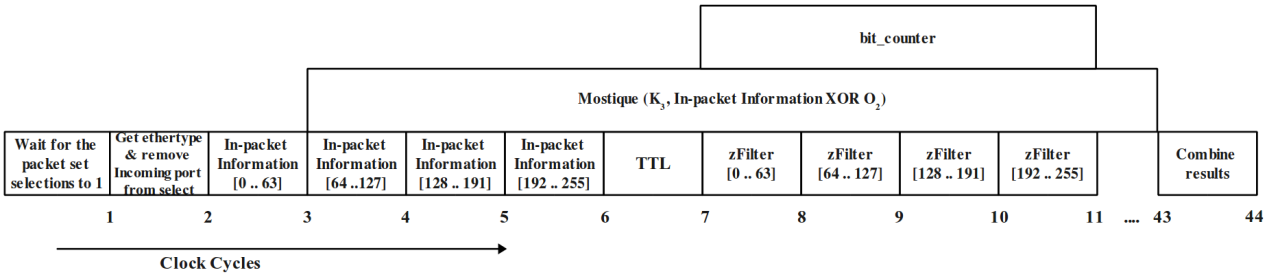

Figure 7: Flow Diagram for Moustique

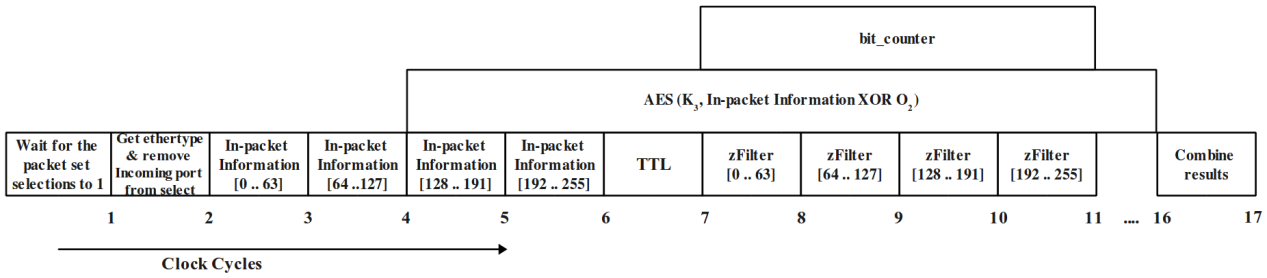

Figure 8: Flow Diagram for AES

The software can send customizable packets to the NetFPGA card. It controls the delay between the transmitting packets, packets size, Time-to-live (TTL) in packet header, computing nonce I, defining iBF and ethernet protocol field.

5. EVALUATION

We now study some of the performance results. For testing the functionality, the two interfaces of a test host were connected to the NetFPGA. One interface was used to send and the other was used to receive packets. Several scripts were run to test forwarding mechanism, verifying set bits in the zFilter, TTL verificaton and ethertype checking. All these were working as expected.

5.1 Performance

The packet traversal times were measured in our test environment. Packets were sent at the rate of 25 packets/second. Sending and receiving operations were implemented in the FreeBSD kernel. Table 1 shows the measurement results with plain wire and one NetFGPA. The measurements were taken with Moustique, AES and then for LIPSIN separately on the NetFPGA. The packet format "new" and "old" refer to packet header with in-packet information and without it. These formats can also be noticed from the flow diagrams for Moustique, AES, and LIPSIN in Figures 7, 8 and 9. The readings were taken for 10 000 samples.

The delay caused by Moustique is 320ns (40 clock cycles for 40 bits) with k set to 5 and m set to 256. The delay caused by AES is 96ns (12 clock cycles). After this, matching is performed only in a single clock cycle. These delays are quite small compared to the measured 3μs overall delay of the whole NetFPGA.The numbers presented in Table 1, the average delay for Moustique and AES measured agrees the expected results. As can be seen Moustique has average delay of 15,272ns and AES has 15,057ns. The measured difference between the two techniques is quite close to the expected results.

Comparing Moustique and AES, with the increase in k and m set to 256, the bits to compute increase with a multiple of 8 in Moustique. Each bit requires 1 clock cycle and hence the clock cycles also increase in multiples of 8. For AES, upto 128 bits the clock cycles remain same that is 12. 128 bits means that with AES k can be set to 16 without any additional performance penalty. Hence, it became clear that the need for having the k bit indices before performing the zFilter comparison and the 64-bits nature of the NetFPGA data path make AES a faster choice.

Table 1: Latency measurement results

Path and Packet format	Average Latency	Standard Deviation
Wire (new)	12,784ns	4,448.96ns
NetFPGA with Mostique (new)	15,272ns	4,991.28ns
NetFPGA with AES (new)	15,057ns	3,756.86ns
Wire (old)	12,549ns	4,867.34ns
NetFPGA with LIPSIN (old)	14,627ns	4,204.58ns

6. RELATED WORK

The first implementation of the zFilters on a NetFPGA was presented in [7]. It used the static forwarding tables containing Link identifiers (or LITs). In this paper, we have have repeated the performance measurements of [7] using NetFPGA 2.0, making the numbers comparable with ours.

What comes to other related work, we only highlight some of the most important ones. In SANE [4], authentication to use communication paths can only be achieved from a central controller in the form of capabilities. The main difference is that we compress the source route by utilizing

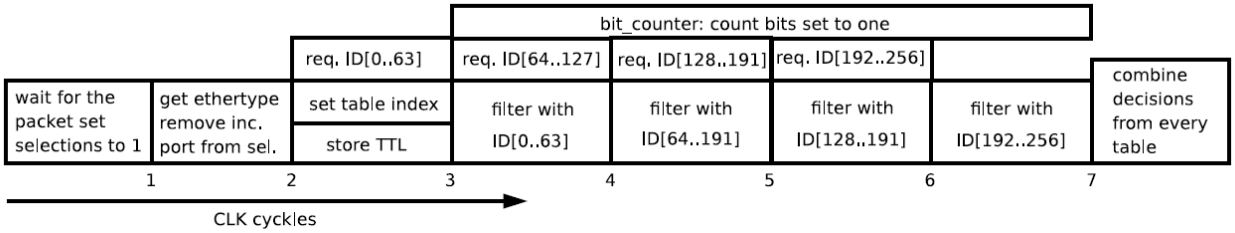

Figure 9: Flow Diagram for LIPSIN

in-packet Bloom filters.

In [8], Ju and Lockwood described their FPGA-based IPsec implementation. It discusses implementation of various cryptographic algorithms, like AES, HMAC-MD5 and HMAC-SHA-1, in an FPGA, and for packet processing purposes. From our work point of view, it gives some cryptographic background.

Path pinning in SNAPP [9] decouples routing and forwarding. When setting up the connections, the routers add their decision into the packet header: a next hop and a Message-Integrity Code over it using its secret key. Finally, the sender will get a source routing-like path towards the receiver. When forwarding, routers can see their decision and verify that it has not been tampered. Similar to our work, forwarding is stateless; the biggest difference is that our scheme uses just a single fixed-sized data structure, the Bloom filter, to encode both the packet path and the authorisation information.

A recent work by Seehray et al, ICING [12], has similar goals to ours: only allowing packets to be sent with the forwarders' and the destinations' consent. Similarly to us, they explore the feasibility and performance of the protocol with a NetFPGA implementation. Their conclusion is that the verification algorithm while forwarding can still maintain line-speed.

7. CONCLUSIONS

In this paper, we have described our early implementation for a source-routing-based forwarding mechanism that is resistant to forwarding-identifier-guessing attacks. In a forwarding fabric based on such a mechanism only authorised nodes are able to send packets; packets sent with guessed forwarding identifiers will be dropped with high probability.

We have briefly described two different implementations, one using the Mostique self-synchronising stream cipher, and the other the AES block cipher function. Contrary to our inital assumption that a stream cipher might be faster as it can efficiently produce a partial result, it turned out that the block-cipher-based implementation is faster in practise. While unrolling the stream cipher might help to give more bits out on each cycle, also the block cipher can be unrolled. In any case, the results show the time taken by the cryptographic operations is negligible compared to the overall NetFPGA forwarding delay.

Acknowledgements

We want to express our gratitude to Mats Näslund, Karl Norrman, and Jukka Ylitalo, who greatly contributed to some of the ideas that this work is based upon.

8. REFERENCES

[1] Advanced Encryption Standard (AES). In *FIPS PUB 197*, 2001.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 1970.

[3] A. Z. Broder and M. Mitzenmacher. Survey: Network Applications of Bloom Filters. *Internet Mathematics*, 2004.

[4] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, , and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security Symposium*, 2006.

[5] J. Daemen and P. Kitsos. The Self-synchronizing Stream Cipher Moustique. In *Springer-Verlag*, 2008.

[6] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander. LIPSIN: Line speed Publish/Subscribe Inter-Networking. In *SIGCOMM*, 2009.

[7] J. Keinanen, P. Jokela, and K. Slavov. zFilter Sprouter - Implementing zFilter based Forwarding Node on a NetFPGA. In *NetFPGA Developers Workshop*, 2009.

[8] J. Lu and J. Lockwood. IPSec Implementation on Xilinx Virtex-II Pro FPGA and Its Application. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 37*, 2005.

[9] B. Parno, A. Perrig, and D. Andersen. SNAPP: Stateless Network-authenticated Path Pinning. In *ACM ASIACCS '08*, 2008.

[10] C. E. Rothenberg, P. Jokela, P. Nikander, M. Sarela, and J. Ylitalo. Self-routing Denial-of-Service Resistant Capabilities using In-packet Bloom Filters. In *the 5th European Conference on Computer Network Defense (EC2ND)*, 2009.

[11] C. E. Rothenberg, C. Macapuna, F. Verdi, M. MagalhÃčes, and A. Zahemszky. Data Center Networking with In-packet Bloom Filters. In *SBRC 2010*, May 2010.

[12] A. Seehray, J. Naousz, M. Walfishy, D. Mazi'eresz, A. Nicolosix, and S. Shenker. A policy framework for the future Internet. In *HOTNETS*, 2009.

[13] A. Zahemszky, A. Csaszar, P. Nikander, and C. Esteve. Exploring The PubSub Routing/Forwarding Space. In *International Workshop on the Network of the Future*, 2009.