Adnan Hassan Ghani

Secure In-packet Bloom Filter Based Forwarding on a Reusable Network Hardware Design

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 07.05.2011

Thesis supervisor:

Prof. Jyri Hämälainen
Aalto University
School of Electrical Engineering

Thesis instructor:

D.Sc. (Tech.) Pekka Nikander
NomadicLab, LM Ericsson AB Oy.

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEEERING

ABSTRACT

Author: Adnan Hassan Ghani

Title: Secure In-packet Bloom Filter Based Forwarding on a Reusable Network Hardware Design

Date: 07.05.2011 **Language:** English **Number of pages:** 8+56

Department: Department of Communications and Networking
Professorship: S-72 Communication Engineering

Supervisor: Prof. Jyri Hämälainen
Instructor: D.Sc. (Tech.) Pekka Nikander

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 unauthorized traffic.

The implementation and testing of in-packet bloom filter forwarding node that uses cryptographically computed link identifiers are discussed in this thesis. Two different cryptographic techniques are tested 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 efficiency of the algorithms is also briefly discussed.

Keywords: Publish/Subscribe, Bloom filters, forwarding node, security, denial-of-service resistance, NetFPGA

ACKNOWLEDGEMENTS

First of all thanks to Almighty Allah for giving me courage and strength to do my Master's thesis successfully.

This Master's thesis was completed at L.M. Ericsson AB Oy. This work was supported by TEKES as part of the Future Internet program of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

My heartfelt appreciation goes to my instructor Pekka Nikander for his attention, guidance, insight and support during this work. I feel very fortunate to work under his instructions. I would like to express my sincere gratitude to my supervisor Professor Jyri Hämäläinen for showing great interest in my work and guiding me throughout the process.

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

Special thanks to Benny, Teemu, Somaya, Andras and Jimmy for their invaluable support throughout the thesis. My gratitude to all my friends at Aalto University, School of Science and Technology.

Finally, but with greatest regard I express my indebtedness towards my parents, siblings and my wife Saima for their overwhelming love and support at every step of my life.

Adnan Hassan Ghani

Jorvas, May 2011

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACK	Acknowledgment
AES	Advanced Encryption Standard
ARK	Add Round Key
BF	Bloom Filter
BS	ByteSub
CCSR	Conditional Complementing Shif Register
CPU	Central Processing Unit
CTRL	Control
DDR	Double Data Rate
DoS	Denial of Service
FID	Forwarding Identifier
FIFO	First In First Out
FPGA	Field Programmable Gate Array
GMAC	Gigabit Ethernet Media Access Controllers
HMAC	Hash-based Message authentication code
iBF	In-packet Bloom Filter
IOCTL	Input/Output Control
IP	Internet Protocol
ISE	Integrated Synthesis Environment
IV	Initialization vector
KDF	Key Derivation Function
LID	Link Identifiers
LIPSIN	Line Speed Publish/Subscribe Inter-Networking
LIT	Link Identity Tags

MAC	Message Authentication code
MAC	Media Access Control
MC	Mix Columns
MGT	Multi Gigabit Transceivers
MPLS	Multi-Protocol Label Switching
NIST	National Institute of Science and Technology
NSA	National Secure Agency
PCI	Peripheral Component Interconnect
PID	Packet Identifier
RAM	Random Access Memory
RDY	Ready
REG	Register
SATA	Serial Advanced Technology Attachment
SDRAM	Synchronous Dynamic Random Access Memory
SHA	Secure Hash Algorithm
SR	Shift Rows
SRAM	Static Random Access Memory
TCP	Transmission Control Protocol
TTL	Time To Live
WR	Write
XOR	Exclusive OR

1 INTRODUCTION

While Bloom filters [19] are commonly used in several roles in networking applications [20], in-packet Bloom filters have only recently gained more attention [1][22]. 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 *et al*. [1], 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 *et al.* [18]

In a follow up paper, Esteve, Jokela, *et al.* [21] 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 authorized 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 blocks unauthorized traffic, at the cost of parameter distribution. Gathering the input parameters for a source route and fast rerouting are out of scope for this thesis.

From the security point of view, the in-packet Bloom filters act simultaneously as forwarding identifiers and forwarding capabilities [21]; 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 appropriate 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 synchronized time-based shared secret and additional in-packet per-flow or per-packet information. The forwarding capabilities are thus expiable 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 need to be examined. In this thesis, we present our 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 the thesis is organized as follows:

Chapter 2, background, where publisher/subscriber based internetworking is discussed. Its types, architecture, forwarding mechanism on the basis of bloom filters are covered here. It is followed by introduction to Stanford NetFPGA, its specifications and reference pipeline.

Chapter 3 discusses the encryption techniques that include *Advanced Encryption Standard, Hash based Message Authentication Code* and *self-synchronizing stream ciphers.* In particular "*Moustique"* is discussed for *self-synchronizing stream ciphers.*

Chapter 4 covers the design part. Secure in-packet bloom filter's architecture and construction is discussed. It is explained using the network example and flow diagrams.

In Chapter 5, the implementation details of the design are discussed. NetFGPA's reference model's implementation and its modification for our design are discussed. It also includes forwarding node's implementation details.

Chapter 6 discusses the testing, evaluation and performance results. Different testing and verification cases are presented. Their result evaluation and performance comparison is also part of this chapter.

Chapter 7 concludes the thesis.

The idea is also presented in a paper "Secure in-packet bloom filter forwarding on a NetFPGA" authored by myself, Adnan Hassan Ghani and Pekka Nikander in the proceedings of *1st European NetFPGA Developers Workshop*, University of Cambridge, Computer Laboratory, Cambridge, UK, Sep. 9–10th, 2010. [27] Chapter 4, 5 and 6 are based on our paper.

zFormation's NetFPGA wikipage is also written by me and can be found at [26]. The tests presented in Chapter 6 are also discussed at [26].

2
BACKGROUND

This chapter covers the basic concepts of *Publish/Subscribe based internetworking*. Its types, architecture and forwarding mechanism are discussed in details. Later comes description of *Stanford NetFPGA* platform. Its specifications and reference pipelines are also mentioned.

2.1 Publish/Subscribe Internetworking

Publish/Subscribe communication paradigm has got a lot of attention in the last few years for distributing information in the wide area networks. The two main components in this system are publishers and subscribers. Publishers submit information to the system and subscribers express interest in specific types of information [1]. The characteristics of this paradigm are that the communication is decoupled in space, time and flow. Decoupled in space, time and flow means Publisher and Subscriber do not need to know each other, do not to be up at the same time and sending/receiving does not block the participants.

The main schemes in which publish/subscribe systems can be divided are as follows:

1. *Topic-based Publish/Subscribe*
2. *Content-based Publish/Subscribe*

2.1.1 Topic-based Publish/Subscribe

In a *topic-based system*, the subscriber expresses interest to specific topics and receives the events related to those particular topics. These topics correspond to a separate logical channel that connects each publisher to all interested subscribers.

Some systems that come in this model are *COBRA Notification Service* [2], *Bayeux* [3] , *SCRIBE* [4] and *iBus* [5].

2.1.2 Content-based Publish/Subscribe

In a *content-based system*, messages are only delivered to a subscriber if the attributes or contents of those messages match constraints defined by the subscriber. It can be defined also as, a filter is defined over the attributes of the notification. The subscriber is responsible for classifying the messages. Some of the systems that come in this model are *SIENA* [6], *LeSubscribe* [7], *Ready* [8] and *Gryphon* [9].

2.1.3 Architecture

The architecture presented in "LIPSIN: Line Speed Publish/Subscribe internetworking" [1] is defined here as my work is based on the same architecture.

The Publish/Subscribe architecture can be defined in a layered and recursive approach. The higher layers are utilizing the lower layers' rendezvous, topology and forwarding functions. At the bottom of the whole architecture lies the "Forwarding or more" shown in Figure 1.

There are two parts of the structure. One is data and the other is control plane. In the control plane, the information about the whole network structure is with the topology system. It creates and spreads this information within the network. Next to topology system is Rendezvous system. It lies on top of the topology system. It deals with the matching between the publishers and subscribers. When there is a publication and it has some subscribers, a request from rendezvous system is sent to the topology system to build a logical forwarding tree. This forwarding tree is basically the path from the current location to the subscriber. It also shares the information about the forwarding with the publisher. [1]

The data plane is responsible for forwarding functionality. It also deals with the transport functions including error detection and traffic scheduling. [1] Some

other functions are also involved i.e. opportunistic caching and lateral error correction but these are out of scope. Our main focus will be on forwarding layer.

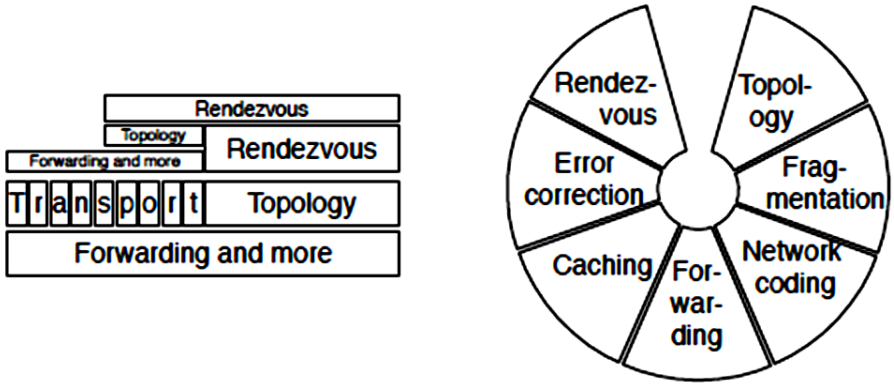

Figure 1. Rendezvous, Topology and Forwarding [1]

2.1.4 Recursive bootstrapping

Bootstrapping is done to invoke the initial connectivity in the network. For this, the rendezvous and topology systems are bootstrapped. It takes place from the lower layer towards the upper layers. At the lower layer, the connectivity takes place from any node towards the rendezvous system in the pub/sub network. The lower layer provides the information to the topology management functions. They exchange information about the connectivity similar to the other routing protocols. Hence, a network graph is mapped. Similar procedure occurs for the rendezvous system to advertise themselves. [1]

2.1.5 Forwarding on Bloom link identifiers

According to *LIPSIN* [1], the links are identified instead of nodes. This means instead of giving names to the nodes, the links are given names. Forwarding takes place on the basis of Bloom-filter-based approach. The forwarding identifiers are created by encoding the link identifiers by the topology system. It also creates new states at the forwarding nodes if required.

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 $k << m$. For example, if $m = 256$ and $k = 5$, the number of unique link identifiers $\approx m!/(m-k)!k! \approx 10^{10}$.

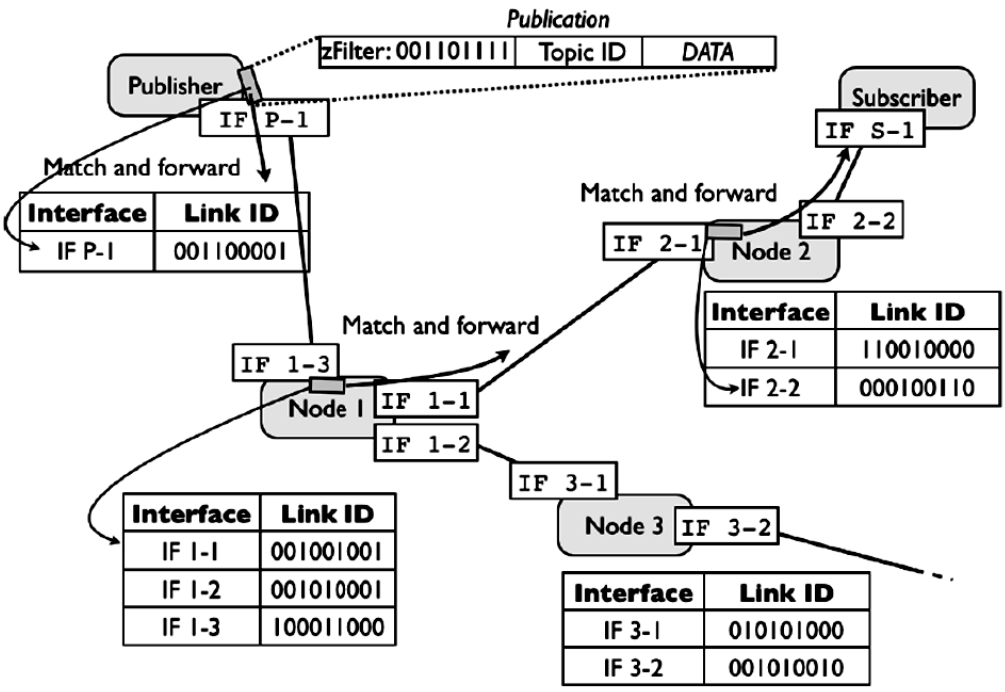

Figure 2. Example of Link IDs assigned for links, as well as publication with a *zFilter*, built for forwarding the packet from the Publisher to the Subscriber [1].

As in [9], 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 link identifiers. This can then be used when packets need to be forwarded from a sender to a receiver (or group of receivers).

In [1], 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 2. In their work, Jokela *et al.* denote the in-packet Bloom filter as a *zFilter*.

When a packet reaches a forwarding node along the path, each candidate outgoing link identifier (Bloom mask) is ANDed 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.

```
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
```

Algorithm 1. Forwarding method of LIPSIN [1].

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 [1], 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*.

2.1.6 Forwarding in TCP/IP based networks

In case of IP, LIPSIN can be considered as another underlying forwarding fabric like Ethernet and MPLS. Upon the arrival of the IP packet into the LIPSIN fabric, a header with a *zFilter* is appended in the start at the entering node. It is then removed at the leaving node. In case of unicast traffic, a pre-computed *zFilter* specifies the specific leaving node and packet is forwarded on that path. For multicast, the entering router of the source needs to keep track of the joins received on multicast group through the edge routers. Hence, it also knows the leaving edges where the packet should be forwarded. Having this information a *zFilter* can be constructed for suitable links. [1]

2.1.7 Link IDs and LITs

To reduce the false positives, Link ID Tags (LITs) are introduced in addition to Link IDs. A signle Link ID is replaced with *d* distinct LITs that is shown in Figure 3. Hence, different candidate *zFilters* can be constructed and one best can be selected among them in terms of optimized for false positive rate, compliance with network policies and false positive rate. [18]

It gives birth to *d* forwarding tables. Each table contains LIT entries for active Link IDs. An index in the packet header determines which forwarding table should be used to perform matching as shown in Figure 4. Its construction is similar to single Link ID. The only difference is *d* distinct candidate filters are calculated. They have equivalent representation of delivery tree. By this way the false forwarding number is minimized. [18]

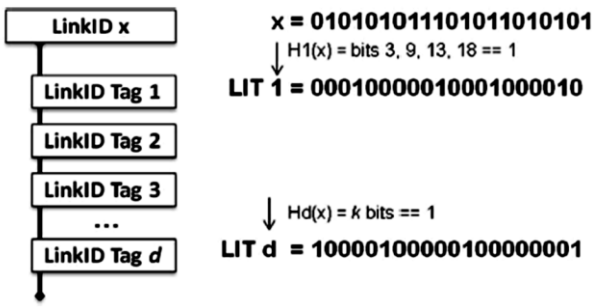

Figure 3. One Link ID to *d* distint LITs [1]

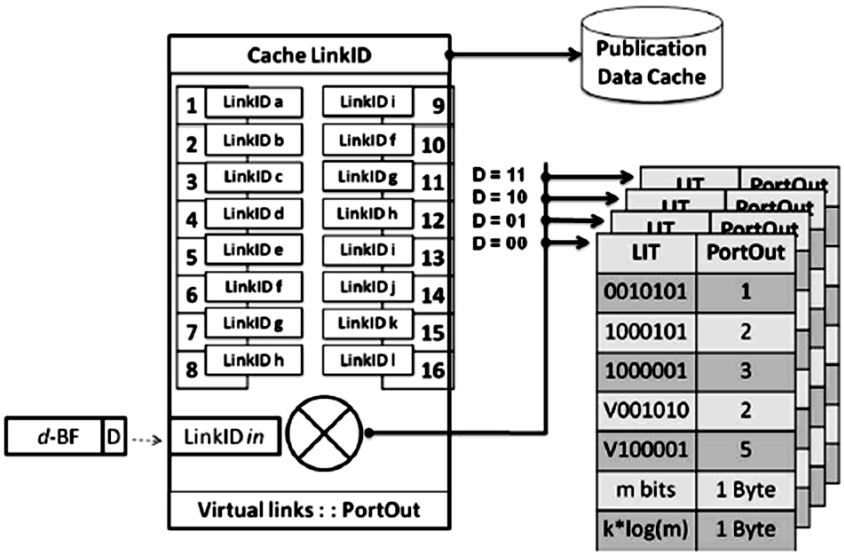

Figure 4. Outgoing interfaces equipped with d forwarding tables, indexed by value in pakcet header [1]

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 thesis we explore some solutions to this security problem and evaluate their performance penalty. The design is discussed in Chapter 4.

2.2 The Stanford NetFPGA Platform

The NetFPGA is an open source hardware platform through which students and researchers are able to build networking systems. They run on line-rate. It enables to build reusable designs [12].

The NetFPGA's existence came at Stanford University where the first version with 10Mb/s Ethernet was designed in 2001. In 2005-2006, the second version

with 1Gb/s was developed. Its working version came into existence in 2007 that was tested by the users of ten universities. With the support of industry like Cisco, Google, Huawei, Juniper, Agilent, Micron, Cypress and Broadcom the beta program started in the start of 2008. The cards were built and launched through Digilent Inc. The open-source gateware, software and hardware were made available on the website. At the moment around 1000 NetFPGA boards are available at around 150 universities and research institutions in different parts of the world.

2.2.1 Specifications

The NetFPGA platform contains one Xilinx Virtex-II Pro 50 FPGA alongwith a Xilinx Spartan FPGA. Former is programmed with user-defined logic while the latter is used to program the former and takes care of PCI interface as well. It contains two *Static RAMs* (SRAMs) and two *Double Data Rate* (DDR2) SDRAM Devices. SRAM operates synchronously and SDRAM operates asynchronously with the NetFPGA. It also contains a quad-port physical layer transceiver (PHY). Using these, NetFPGA can send and receive packets as it provides interface to four twisted-pair Ethernet cables. Two Serial ATA (SATA) enable multiple NetFPGAs to to exchange data at high speed in the platform. Figure 5 shows the NetFPGA. [12]

The NetFPGA library includes a verilog design that instantiates four Gigabit Ethernet Media Access Controllers (GMACs). It also interfaces the logic to SRAM and DDR2 memory. The modules inside use *First-in-First-Out* (FIFO) protocol. The circuit is implemented into the FPGA using standard Computer Aided Design tools. The simulation is run using Mentor Graphics ModelSim tool. It is synthesized using Xilinx ISE tools. The PCI interface enables to program the Virtex NetFPGA. [11]

Figure 5. Photo of a NetFPGA [11]

The NetFPGA release can be divided into three main components. The first one is the kernel module. It is used to communicate with the NetFPGA hardware. The bitfiles to program FPGA through PCI interface, communicating with the register interface from software is all done through kernel module. Second is the utility software that includes read and write programs. The third one is the reference pipeline. These all can be seen in the Figure 6.

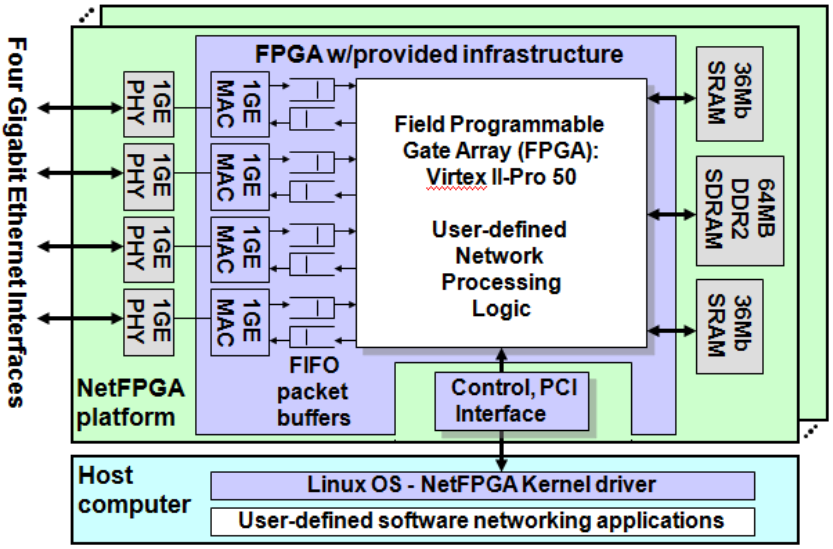

Figure 6. Detailed specifications of NetFPGA [13]

2.2.2 Reference Pipeline

Different sources of packet arrival into the NetFPGA are: i) four network interfaces through Gigabit Ethernet, ii) the host CPU through PCI interface and

iii) other NetFPGAs those are connected by Multi-Gigabit Transceivers (MGT) via SATA connectors. On arrival into the module, the required operations are performed according to user logic. Then an output port lookup module places the packet into the required output queue on which it is meant to be forwarded.

Two buses interconnect the Modules in the pipeline. Those are the packet bus and the register bus.

The 64-bit packet bus sends data from one module to the other. The packet is divided into 64-bit words and sent in series between the modules. The part or whole of a packet can be stored into the FIFOs in each module if required to do some processing. The sum of clocks to process and stream the packet in each module gives us total latency. The register bus can be used by the software to change the hardware registers. *ioctl* calls can be used to access the registers from software and they appear to I/O registers to software. The registers are connected in a chain style and accessible from each module. Figure 7 shows the pipeline structure.

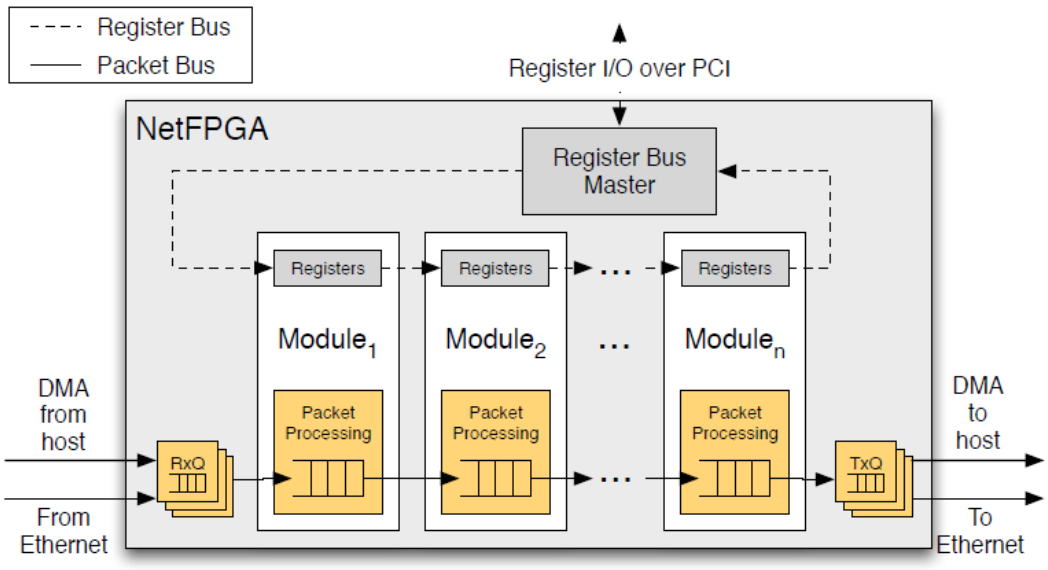

Figure 7. Reference pipeline [12]

3

ENCRYPTION TECHNIQUES

In this chapter the encryption techniques are discussed. The chapter starts with detailed description of block cipher *Advanced Encryption Standard* (AES). Next comes *Hash-based Message authentication* using secure *Hash Algorithm* (HMAC-SHA). Finally, *Self-synchronizing stream ciphers* are discussed. In particular a single bit stream cipher *Moustique* is described with its architecture and construction.

3.1 Advanced Encryption standard

Advanced Encryption Standard (AES) also known as *Rijndael* (named after its Belgian co-inventors Joan Daemen and Vincent Rijmen) is symmetric-key encryption standard. It was announced by *National institute of Standards and Technology* (NIST).

It is based upon the properties of finite field of 256 elements. Each byte can be associated with a unique element of this field. As the field elements can be multiplied and added, this association makes it possible to add and multiply bytes. Similarly, each byte has a multiplicative inverse in a field. Due to all these properties, it's possible to encode entire bytes using nonlinear matrix operations at a time [14].

AES uses a fixed block size of 128 bits and a key size of 128, 196 or 256 bits.

Here we will discuss only the case where key size is 128 bits. It comprises of several transformation rounds to perform diffusion of the bits. Particularly for our case it's ten rounds. Hence, a 16 bytes plaintext is converted into 16 bytes ciphertext by going through ten rounds of transformations where each round has

its own derived key from the original key. It operates on 4x4 array of bytes that is known as state.

3.1.1 Overview of the algorithm

1. Key expansion The W-matrix is computed from the key. The first four columns of the original keyword matrix is added to the input data consisting of 16 bytes arranged in a 4x4 matrix.
2. Nine Rounds of transformations
    - ***i.*** ***ByteSub (BS)*** According to a lookup table each byte is replaced with another byte.
    - ***ii.*** ***ShiftRows (SR)*** Each row is shifted cyclically a certain number of steps.
    - ***iii.*** ***MixColumns (MC)*** Columns of the state are mixed in a special order.
    - ***iv.*** ***AddRoundKey (ARK)***
3. Final round (MC is not applied in this round)
    - ***i.*** ***ByteSub***
    - ***ii.*** ***ShiftRows***
    - ***iii.*** ***AddRoundKey***

Now we'll explain the above steps in detail Using *S-Box* for the code The *S-Box* for the *AES* code is given by the 16x16 matrix.

The *S-box* is used to replace a byte with a coded byte. It is done as if the input byte is *($C_7C_6 \ldots C_0$)*, then the number in row $\sum_{j=4}^{7} c_j 2^{j-4}$ and in column $\sum_{j=0}^{3} c_j 2^j$ of S-box is the integer representation of the new byte. For example, input byte is 10101100, then the row is first four bits 1010 = 10 (in integer) and column is last four bits 1100 = 12. So the corresponding number from the above matrix is 145 = 10010001. [14]

$$\begin{pmatrix}
99 & 124 & 119 & 123 & 242 & 107 & 111 & 197 & 48 & 1 & 103 & 43 & 254 & 215 & 71 & 118 \\
202 & 130 & 201 & 125 & 250 & 89 & 71 & 240 & 173 & 212 & 162 & 175 & 156 & 164 & 114 & 192 \\
183 & 253 & 147 & 38 & 54 & 63 & 247 & 204 & 52 & 165 & 229 & 241 & 113 & 216 & 49 & 21 \\
4 & 199 & 35 & 195 & 24 & 150 & 5 & 154 & 7 & 18 & 128 & 226 & 235 & 39 & 178 & 117 \\
9 & 131 & 44 & 26 & 27 & 110 & 90 & 160 & 82 & 59 & 214 & 179 & 41 & 227 & 47 & 132 \\
83 & 209 & 0 & 237 & 32 & 252 & 177 & 91 & 106 & 203 & 190 & 57 & 74 & 76 & 88 & 207 \\
208 & 239 & 170 & 251 & 67 & 77 & 51 & 133 & 69 & 249 & 2 & 127 & 80 & 60 & 159 & 168 \\
81 & 163 & 64 & 143 & 146 & 157 & 56 & 245 & 188 & 182 & 218 & 33 & 16 & 255 & 243 & 210 \\
205 & 12 & 19 & 236 & 95 & 151 & 68 & 23 & 196 & 167 & 126 & 61 & 100 & 93 & 25 & 115 \\
96 & 129 & 79 & 220 & 34 & 42 & 144 & 136 & 70 & 238 & 184 & 20 & 222 & 94 & 11 & 219 \\
224 & 50 & 58 & 10 & 73 & 6 & 36 & 92 & 194 & 211 & 172 & 98 & 145 & 149 & 228 & 121 \\
231 & 200 & 55 & 109 & 141 & 213 & 78 & 169 & 108 & 86 & 244 & 234 & 101 & 122 & 174 & 8 \\
186 & 120 & 37 & 46 & 28 & 166 & 180 & 198 & 232 & 221 & 116 & 31 & 75 & 189 & 139 & 138 \\
112 & 62 & 181 & 102 & 72 & 3 & 246 & 14 & 97 & 53 & 87 & 185 & 134 & 193 & 29 & 158 \\
225 & 248 & 152 & 17 & 105 & 217 & 142 & 148 & 155 & 30 & 135 & 233 & 206 & 85 & 40 & 223 \\
140 & 161 & 137 & 13 & 191 & 230 & 66 & 104 & 65 & 163 & 45 & 15 & 176 & 84 & 187 & 22
\end{pmatrix}$$

S-Box

Representing input data

As input is 16 bytes (128 bits) then they can be arranged into 4x4 matrix in following order

$$A = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}$$

ByteSub Transformation

The *ByteSub* Transformation is non-linear and hence, resistant to both linear and differential attacks. In this step each byte in *A* is replaced using the S-Box according to the procedure stated above and the new Matrix can be represented as follows

$$B = \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}$$

The ShiftRow Transformation

This linear transformation causes diffusion of the bits. Row *j* of the matrix is shifted cyclically to the left by *j* offsets so the new matrix is of the form

$$C = \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,1} & B_{1,2} & B_{1,3} & B_{1,0} \\ B_{2,2} & B_{2,3} & B_{2,0} & B_{2,1} \\ B_{3,3} & B_{3,0} & B_{3,1} & B_{3,2} \end{pmatrix}$$

The MixColumn Transformation

This transformation causes strong diffusion where each byte in matrix *C* is represented as elements of F256 and multiplied by matrix M

$$MC = \begin{pmatrix} \alpha & \alpha+1 & 1 & 1 \\ 1 & \alpha & \alpha+1 & 1 \\ 1 & 1 & \alpha & \alpha+1 \\ \alpha+1 & 1 & 1 & \alpha \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,1} & B_{1,2} & B_{1,3} & B_{1,0} \\ B_{2,2} & B_{2,3} & B_{2,0} & B_{2,1} \\ B_{3,3} & B_{3,0} & B_{3,1} & B_{3,2} \end{pmatrix}$$

For example the first column of the matrix *C* is of the form

$$\begin{pmatrix} 10000001 \\ 00000000 \\ 00001001 \\ 00101010 \end{pmatrix}$$

Then the first in the left uppermost position of MC will be computed as: $\alpha(\alpha^7 + 1) + (\alpha + 1)(0) + (1)(\alpha^3 + 1) + (1)(\alpha^5 + \alpha^3 + \alpha) = \alpha^8 + \alpha^5 + 1 = \alpha^5 + \alpha^4 + \alpha^3 + \alpha = 00111010$

RoundKey Addition

Round key addition is the modulo2 addition (*XOR*) of MC with the Round Key at the end of j^{th} round. Round Key is obtained by means of key schedule. Key schedule can be explained by considering a W-Matrix. The initial key of 16 bytes

are placed in first four columns of W-Matrix which is 4 x 44. The other columns of W-matrix are generated from first four columns in following manner. As we will start computing from the fifth column then say $j \geq 4$ and column $j + 1$ is w_j. If j is not a multiple of 4, then

$$\boldsymbol{w_j = w_{j-4} + w_{j-1}}$$

where addition is modulo 2. If j is multiple of 4 then following steps should be followed

1) Replace every byte in column $\boldsymbol{w_{j-1}}$ with a byte from S-Box using ByteSub Transformation.
2) A vector $\boldsymbol{n_{j-1}}$ is created by moving the top byte to the bottom and every other byte one place up in the matrix obtained in step 1.
3) The new column matrix $\boldsymbol{w_j}$ is computed as

$$\boldsymbol{w_j = w_{j-4} + n_{j-1} + v_j}$$

Where $\boldsymbol{v_j}$ can be given as

$$\begin{pmatrix} \alpha^{(j-4)/4} \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

3.2 HMAC-SHA-256

Hash-based Message authentication code (HMAC) is used for the calculation of a *message authentication code* (MAC) on the basis of a cryptographic hash function with a secret key. MAC has the capability to check both the data integrity and authenticity of the message. Different hash functions like *MD5, SHA-1, SHA-2,* etc. can be used to compute HMAC and hence the resulting algorithm is known as *HMAC-MD5, HMAC-SHA-1, HMAC-SHA-2,* etc. Its strength depends upon the hash function used, hash output length and the secret key. HMAC can be mathematically defined as

$$HMAC(K, m) = H((K \oplus opad) \parallel H((K \oplus ipad) \parallel m)$$

where *H()* is the hashing function that we will discuss later on. *K* is the secret key, *m* is the message, ‖ is concatenation. *opad* is the outer padding constant and its value is 0x5c times blocksize while *ipad* is inner padding with value 0x36 times block size. [15]

Secure Hash Algorithm (SHA)

SHA-0, SHA-1 and SHA-2 are the set of cryptographic hash functions. They are designed by *National Secuity Agency* (NSA) and published by NIST. SHA-1 avoids some weaknesses from SHA-0 by correcting an error otherwise they are quite similar. SHA-2 is quite different and stronger than the both. SHA-2 consists of four hash functions on the basis of different digest sizes i.e. 224, 256, 384 and 512. Table 1 below gives the comparison between the different SHA functions. These were calculated on a Intel Core 2.18 GHz under 32-bit Vista. [15] The pseudo code for SHA-256 can be found in Appendix A.

Algorithm And Variant		Output size (bits)	Internal state size (bits)	Block size (bits)	Max message size (bits)	Word size (bits)	Rounds	Operations	Collisions found
SHA-0		160	160	512	$2^{64}-1$	32	80	+,and,or,xor,rot	Yes
SHA-1		160	160	512	$2^{64}-1$	32	80	+,and,or,xor,rot	Theoratical attack (2^{51})
SHA-2	**SHA-256/224**	256/224	256	512	$2^{64}-1$	32	80	+,and,or,xor,shr,rot	None
	SHA-512/384	512/384	512	1024	$2^{128}-1$	64	80	+,and,or,xor,shr,rot	None

Table 1. Performance Comparison of SHA functions [15]

SHA-256 and SHA-512 are novel hash functions. SHA-256 is computed with 32 bit words while SHA-512 is computed with 64 bits words. The number of rounds, shift amounts and additive constants for the computation differ but still the

structure is similar. SHA-224 and SHA-384 use different initial values from the first two and can be simply seen as their truncated versions.

3.3 Self-synchronizing stream cipher

A stream cipher or a state cipher performs encryption on a plaintext bits with a pseudorandom cipher bit stream by an exclusive-or (XOR) operation. The plaintext bits are encrypted one at a time. During the encryption process, the successive bits get transformed. In self-synchronizing stream cipher, to compute the pseudorandom bits i.e. keystream, it uses several of previous *N* ciphertext bits. The idea behind using a self-synchronizing stream cipher is that if the synchronization is lost, the state will eventually recover as it is filled up again by received bits cipher bits.

In stream cipher, if m^t is the plaintext and z^t is the keystream, then the resulting ciphertext c^t can be given as [16]

$$c^t = m^t \oplus z^t$$

Where $\oplus$ is the exclusive-or. Similarly the decryption can be performed as

$$m^t = c^t \oplus z^t$$

The keystream z^t is determined by

$$z^t = F_c[K](c^{t-n_m} \dots c^{t-1})$$

Where last n_m ciphertext bits and cipher key K of n_k bits take part in the computation of cipher function *Fc* and determine z^t. When computing for the first time, we don't have any previous nm ciphertext bits so we need an *initialization vector* that can be given as

$$c^{1-n_m} \dots c^o = initialization\ vector$$

The encryptor and decryptor both should have initialization vector. Figure 8 shows the block diagram of the encryptor and decryptor of a self synchronizing stream cipher.

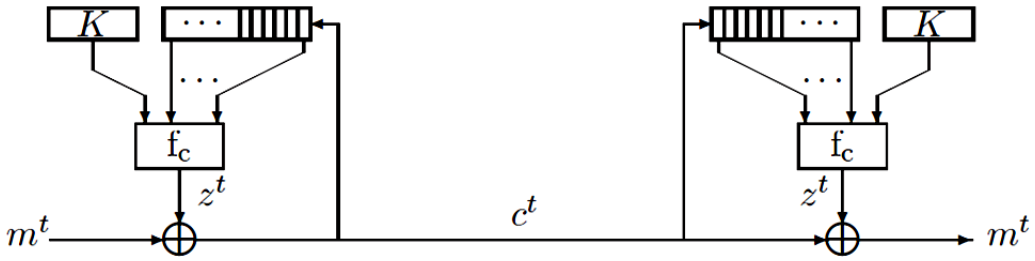

Figure 8. The self synchronizing stream cipher [16]

The cipher function architecture can be built using the idea of pipelining and *conditional complementing shift registers* (CCSR). Figure 9 shows different stages from *bs* to *Gi* with shift registers. The encryption speed has limitation of the stages, so the stages should be simple with small propagation delay. [16]

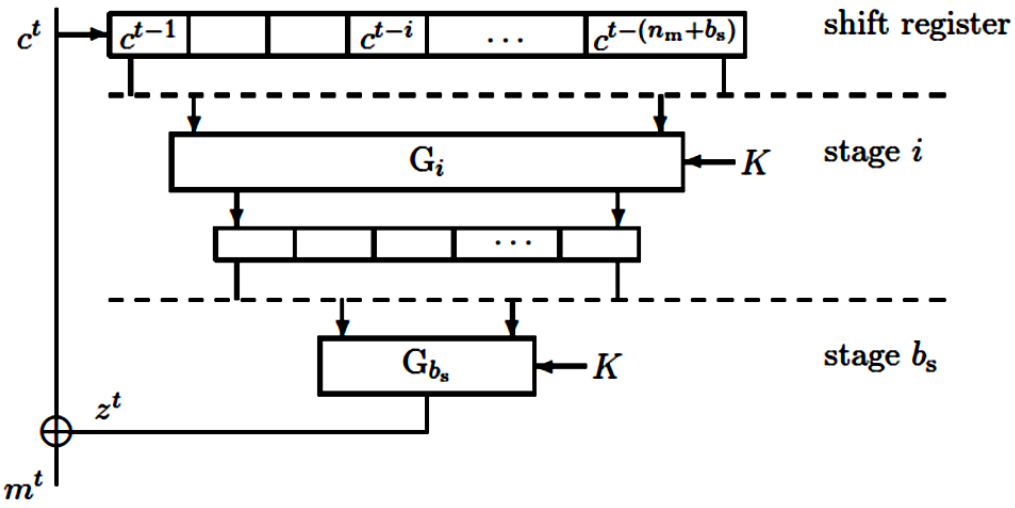

Figure 9. Self synchronizing stream cipher with cipher function consisting of stages [16]

By replacing the shift register with finite state machine will improve the propagation property. If q is the internal state and G the state updating transformation then

$$q^{t+1} = G(q^t, c^t)$$

3.3.1 Moustique cipher function

Now we will explain a single bit self-synchronizing stream cipher and its specifications. So we have symbol size n_s = 1, key size n_k = 96, input memory n_m = 105. Cipher function delay b_s = 9.

CCSR's 128 bits are partitioned in 96 cells and denoted by q_j where j ranges from 1 to 96. The number of bits allocated to the cells is dependent upon the value of j and denoted by n_j given in Table 2 and shown in Figure 10. The bits within the cell are denoted by q_i^j with $0 \leq i \leq n_j$. Moustique has 8 internal stages denoted by a^i. The first stage is the CCSR and is denoted by a^o with length 128. $a^1$to a^5 have length 53. a^6 has length of 12 and a^7 has length 3. [17] The encryption and decryption using moustique is given is Figure 12.

Range of *j*	n_j
1 – 88	1
89 – 92	2
93 – 94	4
95	8
96	16

Table 2. Number of bits per cell

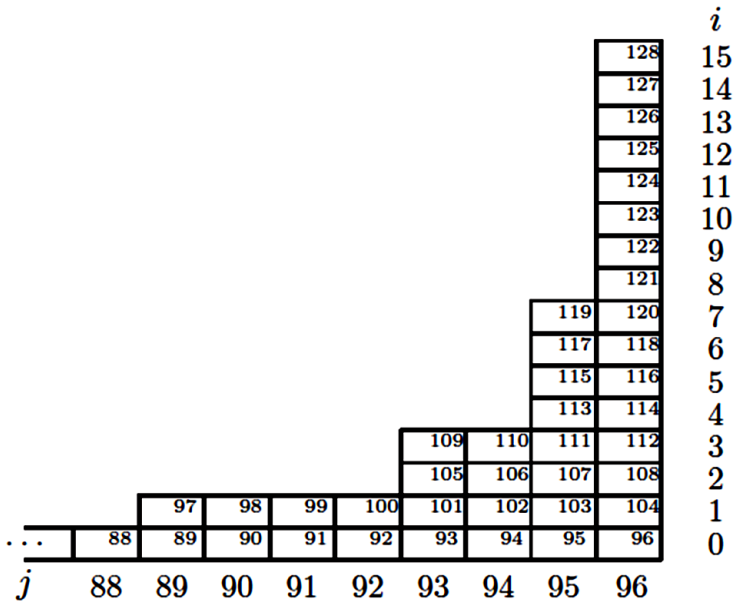

Figure 10. CCSR Expansion [17]

There are three state updating functions which consist of basic addition and multiplication. Those functions are given below and there circuits are shown in Figure 11.

$$g_0(a,b,c,d) = a + b + c + d$$

$$g_1(a,b,c,d) = a + b + c(d+1) + 1$$

$$g_2(a,b,c,d) = a(b+1) + c(d+1)$$

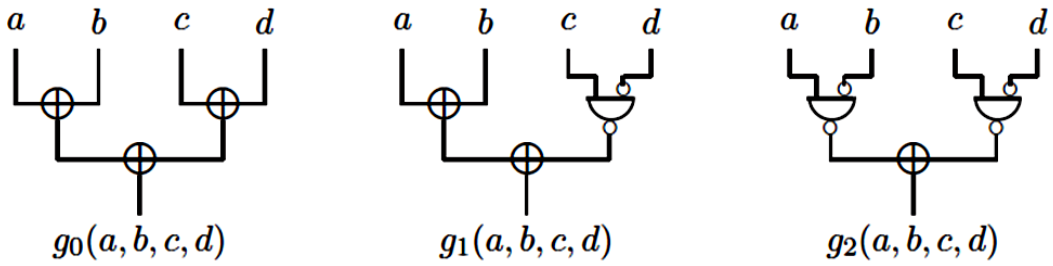

Figure 11. State updating functions [17]

The CCSR bits are calculated using following

$$q_j^i \Leftarrow g_x\left(q_{i \bmod n_{j-1}}^{j-1}, k_{j-1}, q_{i \bmod n_v}^{v}, q_{i \bmod n_w}^{w}\right)$$

with $v \geq 0$, $w < j-1$. The values for x, v and w for all combinations are specified in Table 3 except for $j \leq 2$, $j = 96$ and $i > 1$. For $j \leq 2$ the q^v *and* q^w are taken to be 0. The 15 bits q_i^{96} with $i > 0$ is given by

$$q_j^{96} \Leftarrow g_2\left(q_{i \bmod 8}^{95}, q_0^{95-i}, q_{i \bmod 4}^{94}, q_{1 \bmod n_{94-i}}^{94-i}\right)$$

Index	Function	v	w
$(j-i) \bmod 3 = 1$	g_0	$2(j-i-1)/3$	$j-2$
$(j-i) \bmod 3 = 2$	g_1	$j-4$	$j-2$
$(j-i) \bmod 6 = 3$	g_1	0	$j-2$
$(j-i) \bmod 6 = 0$	g_1	$j-5$	0

Table 3. Function, v and w values [17]

Output	Equation			Input
$a_i^1, 0 \leq i < 53$	$a_{4i \bmod 53}$	$\Leftarrow$	$g_1(a_{128-i}, a_{i+18}, a_{113-i}, a_{i+1})$	$a_i^0, 1 \leq i < 128$
$a_i^2, 0 \leq i < 53$	$a_{4i \bmod 53}$	$\Leftarrow$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^1, 0 \leq i < 53$
$a_i^3, 0 \leq i < 53$	$a_{4i \bmod 53}$	$\Leftarrow$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^2, 0 \leq i < 53$
$a_i^4, 0 \leq i < 53$	$a_{4i \bmod 53}$	$\Leftarrow$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^3, 0 \leq i < 53$
$a_i^5, 0 \leq i < 53$	$a_{4i \bmod 53}$	$\Leftarrow$	$g_1(a_i, a_{i+3}, a_{i+1}, a_{i+2})$	$a_i^4, 0 \leq i < 53$
$a_i^6, 0 \leq i < 12$	a_i	$\Leftarrow$	$g_1(a_{4i}, a_{4i+3}, a_{4i+1}, a_{4i+2})$	$a_i^5, 0 \leq i < 53$
$a_i^7, 0 \leq i < 3$	a_i	$\Leftarrow$	$g_0(a_{4i}, a_{4i+1}, a_{4i+2}, a_{4i+3})$	$a_i^6, 0 \leq i < 12$

Table 4. Bit updating functions for the stages [17]

Table 4 has the bit updating functions for the stages. In the equations if the index is out of bound then 0 should be taken for those values.

The keystream bit can be given as

$$z = a_0^7 + a_1^7 + a_2^7$$

So

$$p \Leftarrow g_0(c, a_0^7, a_1^7, a_2^7)$$

and

$$c \Leftarrow g_0(g, a_0^7, a_1^7, a_2^7)$$

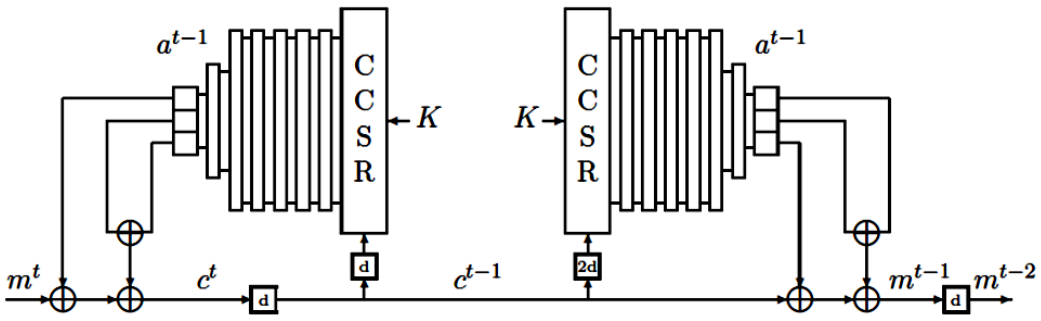

Figure 12. Encryption and decryption using moustique [17]

4 DESIGN

In this chapter, we present the detailed architecture 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-synchronizing stream cipher and the other on the standard AES block cipher. Our initial assumption was that using a self-synchronizing stream cipher could allow us to perform more parallelized and thereby faster forwarding decisions. That assumption turned out to be false, for a number of reasons. Those reasons are explained in the coming sections. [27]

4.1 Secure in-packet Bloom filters

As described in section 2.1, "the 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". [27]

Here we will discuss the solution to this problem with our basic concept of the secure in-packet bloom filters and their construction based on [27].

Our design 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 basis. The idea is that there is a function Z, essentially evaluated for each packet and for each potential outgoing interface, which 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

$$O = Z(K, M, I)$$

where the input parameters of Z are defined as follows:

1. *K* is a semi static 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.

As far as the security requirements are concerned, semi static part *K* needs to be very strong i.e. it should be impracticable to predict Bloom Filter by reusing the older *K* values if they get changed. *I* can be weak, i.e. the attackers are allowed to infer partial information with some controllable probability as long as attacks are not trivial.

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* (KDF):

$$K_i = KDF(K, L_i)$$

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:

$$O_1 = F_1(K_1, S)$$

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

$$O_2 = F_2(K_2 , O_1 || M)$$

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

$$O = O_3 = F_3(K_3 , O_2 \oplus I)$$

where ⊕ denotes exclusive OR.

4.2 Reasons for choosing Encryption techniques

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 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. [27]

We consider two constructions, using a self-synchronizing 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 over counter, using HMAC-SHA-256. [27]

The idea behind using a self-synchronizing stream cipher is that if the synchronization is lost, the state will eventually recover as it is filled up again by received bits. However, in our case we don't really need this property. [27]

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-synchronizing 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 single-bit bases, they are fine for line-speed processing. [27]

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 work we only report the implementation, using the *Moustique* (see section 3.3.1). The other block cipher technique that we have considered is Advanced Encryption Standard AES: see section 3.1.

4.3 Functionality using example

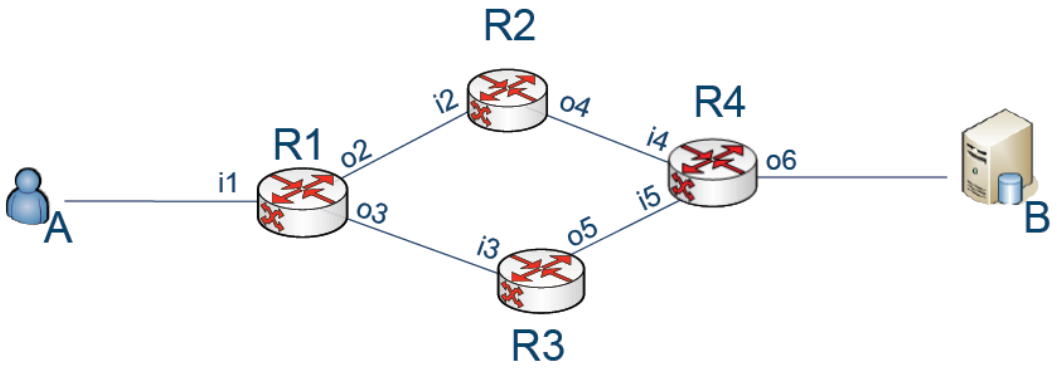

Figure 13. Bloom filter based routing

A concrete example is given to explain the functionality. In relation to Figure 13, assume the user A wants to send a packet to the server B through the network. As a consequence of this factorization of F, the calculation of O_3 can be performed by the user A. To facilitate this, the "name server" would provide user A, not with the final bloom filter BF, but rather with a set of O_2 values which describes the path in the network for the particular "session" or "publication" and the key K_3. The user A would then compute the BF for each packet according to the following principle:

For each *O_2-value in the set received from "name server"* **do**:

Generate information *I* (could be a random value)
Compute $O = F_3(K_3, O_2\text{-}value \,||\, I)$
Insert *O* in *BF*

User A next inserts the constructed BF and the generated information *I* in the packet and sends it to router R_1.

The router R_1 has pre-computed O_2 values for each of its outgoing links for each of the flows it is aware of.

Upon receipt of the packet, router R_1 computes O_3 for each of its links (represented by the O_2 values in this computation) based on the information *I* received in the packet and checks if O_3 is a member of the BF also received in the packet. If the O_3 value is present in the BF, the router R_1 forwards the packet on the corresponding link. It is here assumed that R1 can obtain the O_2 value from the context information C, e.g. the link names as exemplified in Figure 13, and K_2 and K_3 from the master key *K*, shared with the "name server", by applying the KDF function. As an alternative to sharing the master key *K* with the "name server", the "name server" may distribute the derived keys K_1, K_2 and K_3 to the router R_1 (and similarly to other routers).

4.4 Flow of the design

The example discussed in section 4.3 uses a "name server". As our main focus is on forwarding so we have adopted the design in such a form that the name server part is also done by the sender.

The overall flow of the design is depicted in figures 14 and 15. Figure 14 shows the flow at the sender side and Figure 15 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. [27]

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 candidate 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 section 2.1.5), 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.[27]

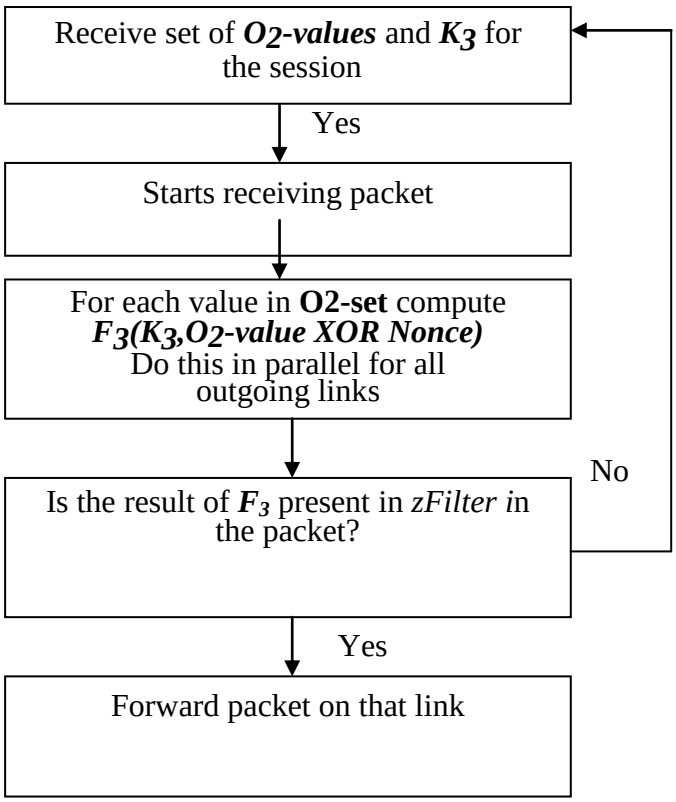

Figure 14. Flow diagram of a forwarding node operation

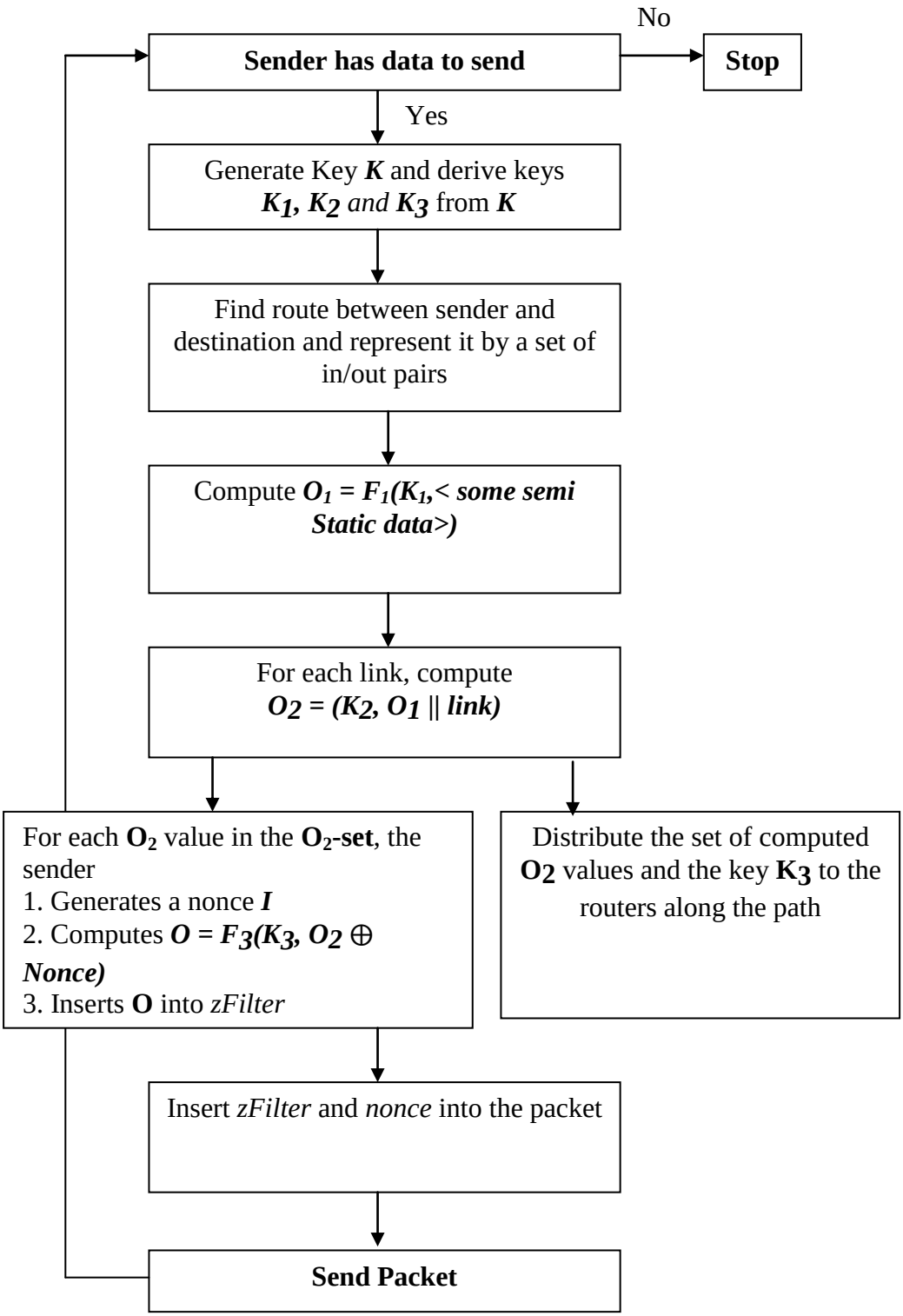

Figure 15. Flow diagram of a sender's operation

4.5 Packet Header Format

The header of the forwarding packet is depicted in Figure 16 in 32 bit format. First 14 bytes are MAC layer header followed by forwarding header. The forwarding header can be divided into two parts. The first header carries nonce and the second one carries bloom filter. The important fields for the header are header length, nonce, *d*, time-to-live (TTL) and BF.

The length of the header for each part is defined in header length field that is one byte long.

Nonce *I*, changes per-packet and 256-bits are reserved for this. A counter that is encrypted using HMAC-SHA-256 is 256 bits long.

d defines which of the link ID tags should be used. This was required for the previous implementation and is of no use for the current design.

TTL takes one byte and 3 bytes are reserved for future use.

BF, can also be referred as *forwarding Identifier* (FID) takes 256-bits. Payload follows the header.

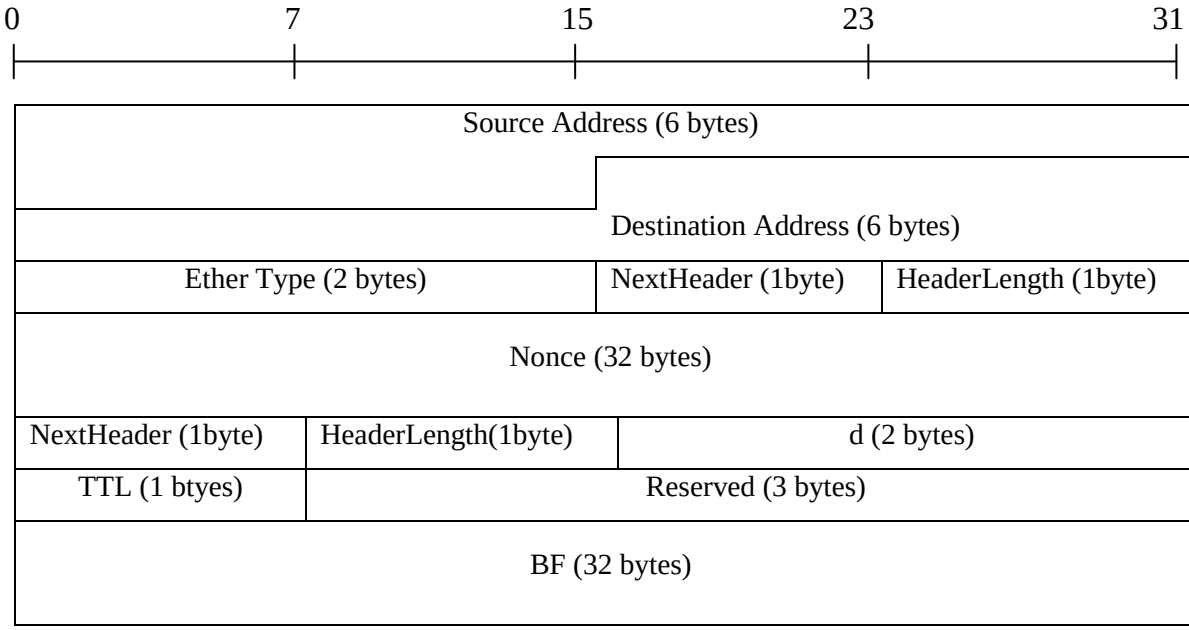

Figure 16. Forwarding header format

5
IMPLEMENTATION

This chapter covers the implementation of the design described in chapter 4. It describes how reference implementation of NetFPGA is taken and modified for our own design. The detailed implementation description of forwarding node in hardware is given. It also briefly covers the software management part. This chapter is also discussed in our paper [27].

In this work, we have taken the previous implementation of zFilter based forwarding node on a NetFPGA [18] and optimized it to our needs of implementing zFormation, 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*: see section 4.1.

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. The implementation of Name Server is out-of-scope for the thesis. Our main and initial emphasis was to get a forwarding node working. Its implementation details are as follows.

5.1 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 17.

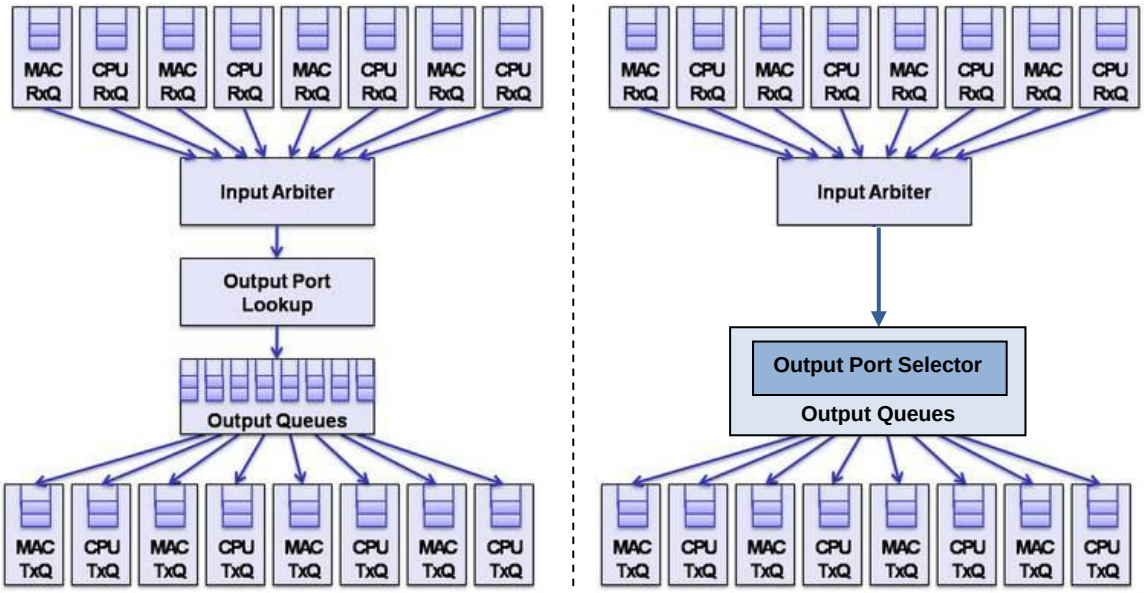

Figure 17 Reference datapath [13] on left and modified datapath on right

5.1.1 Modified datapath

The modules that we have used in our implementation from the reference implementation and our own designed are only discussed here. In Figure 17 we have several queues named as **`Rx_queues`** followed by **`input_arbiter`**. Module **`output_queues`** follows **`input_arbiter`** and has a sub-module **`output_port_selector`**. This **`output_port_selector`** contains our main implementation of *zFormation*. Finally we have **`Tx_queues`**.

Rx Queues

These are the queues which receive packets from the IO ports. Those ports include Ethernet and PCI over DMA. As can be seen in Figure 17, there are total eight receiver queues, four for MAC and four for CPU DMA. These queues are interleaved in such a fashion that port 0 is MAC, port 1 is CPU, port two is again MAC and port 3 is CPU and so on. These

ports are connected to **user_data_path** which contain rest of the modules. [13]

Input arbiter

The functionality of **input_arbiter** is to select the Rx queue from which the packets should be taken and then forwarding them to the next module in the flow. [13] In the reference design **output_port_lookup** was the next coming module. But in our design, **input_arbiter** is directly connected to **output_queues**.

Output_Queues

Module **output_queues** has a sub-module **output_port_selector**. The packets come in **output_queues** and stored in a RAM until the decision to forward on which Tx queue is taken in module **output_port_selector**. The forwarding decision is based on *zFormation*. 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 from the **output_port_selector** module. Further details on **output_port_selector** are discussed in section 5.2.

Tx Queues

Tx queues send packets out on the IO port received from the **output_queues**. They have the same alignment as Rx queues i.e. they are interleaved in the same fashion.

5.1.2 Packet bus

Here we will discuss the signaling details that take place between the pipeline's modules and the registers.

The speed at which the 64 bit wide user_data_path is running is 125MHz. This means that the maximum bandwidth that could be achieved is 8Gbps. Packets arrive at different modules and pushed to the other modules using synchronous FIFO like protocol. The signals which are used are **`write (WR), ready (RDY), data (DATA)`** and **`control (CTRL)`**. The sizes of these signals are **`WR`** one bit, **`RDY`** one bit, **`DATA`** 64 bits and **`CTRL`** 8 bits. Let's assume module *j* has packet that it wants to push forward to next module *j+1*. When the module *j+1* is ready to accept data, it asserts the **`RDY`** signal. On this module *j* places the packet on **`DATA`** bus with required information on **`CTRL`** bus and asserts the **`WR`** signal. Module *j* accepts the data on this and keeps on receiving data until it deasserts the **`RDY`** signal. It should be done one clock cycle prior to intentions of not receiving data anymore. [23]

The **`CTRL`** bus has two purposes. One is to distinguish between module headers and the second is to indicate the last byte of packet. It has non-zero value for different headers until it receives the payload. For data part in the packet it goes to zero. Finally, when comes the end of the packet, the **`CTRL`** bus gets the value of the last byte. Hence, giving indication to the end of the packet with last byte in the last word. It is shown in Figure 18. The **`CTRL`** bus has 0xXX and 0xYY (some non-zero) value for different headers until the packet data starts when it get 0x00 value. The last word has 0x40 that in binary is 0b01000000. This shows the 7 byte is the last byte in the last word. [13]

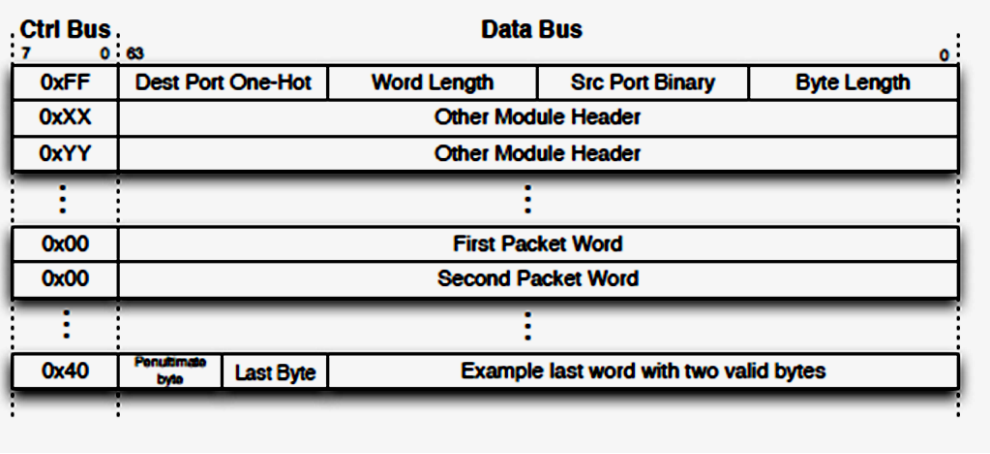

Figure 18. General format of the packet passing on the packet bus [23]

5.1.3 Register bus

The register bus is designed in the form of a chained pipeline that gives access to the host to change values in their own modules. The 32 bits wide register bus also runs at 125MHz. There are two set of signals. One for incoming requests and the second for outgoing replies. For incoming requests the registers are: **`REG_REQ_IN`**, **`REG_ACK_IN`**, **`REG_RD_WR_L_IN`**, **`REG_ADDR_IN (23-bits)`**, **`REG_DATA_IN (32-bits)`**, **`REG_SRC_IN (2-bits)`**. For outgoing replies the registers are: **`REG_REQ_OUT`**, **`REG_ACK_OUT`**, **`REG_RD_WR_L_OUT`**, **`REG_ADDR_OUT (23-bits)`**, **`REG_DATA_OUT (32-bits)`**, **`REG_SRC_OUT(2-bits)`**.[13]

For request/reply **`REG_REQ_IN`** and **`REG_REQ_OUT`** are set to high for one clock cycle. Signals **`REG_RD_WR_L_IN`** and **`REG_RD_WR_L_OUT`** show the request/reply is read or write. For read it is high and for write it is low. Signals **`REG_ACK_IN`** and **`REG_ACK_OUT`** should be low for request and high for reply. "The **`REG_SRC_IN/OUT`** signals are used by register request initiators to identify the responses that are destined to the requestor. Each requestor should use a unique value as their source address." [13]

`REG_ADDR_IN` carries the right address of the module for which the request is made. The module looks for the address in this register. If it matches it performs the request. Once the request is finished, it places the processed data on **`REG_DATA_OUT`** and sets the **`REG_ACK_OUT`**. Rest output signals get all input values. All this is done in a single clock cycle. For the modules if **`REG_ADDR_IN`** doesn't match it forwards all the inputs to the output registers set. [13]

5.2 Packet Forwarding Operations

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 given in Figure 19.

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. One 32-bit port is used for writing these values into the registers. [27]

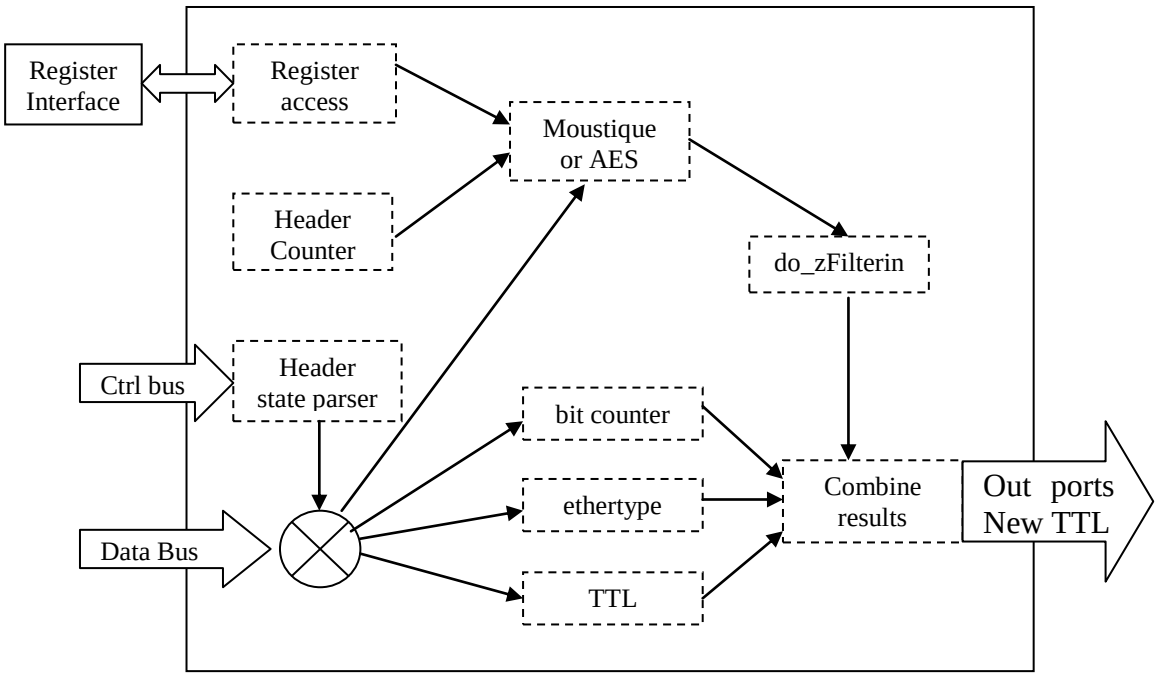

Figure 19. output_port_selector module structure

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. [27]

For computing the dynamic Link IDs, two separate implementations were made, one for the *Moustique* stream cipher and the other for *AES* block cipher. Both of them are explained as follows.

5.2.1 Moustique

For **`Moustique`**, as the 96 bits of K_3 register and 256 bits of O_2 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_moustique`** set to 1. **`Moustique`** is instantiated four times, once for each outgoing port, for the NetFPGA in a parallel manner. [27]

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 (see section 2.1.5). 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`** signal is set to 1 by **`moustique`** for one clock cycle, so that decrypted data can be read by **`output_port_selector`**. [27]

5.2.2 AES

In the block cipher case, AES, with the key and block sizes of 128, is used for the computation of F_3. We used the **OpenCores AES implementation**.[24] As the in-packet information I arrives, it is exclusive-ORed with the values in the O_2 value set. The data and the key K_3 are loaded into the input of the cipher function, and **`start_AES`** is set to 1. [27]

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. [25] The clock timings are given in Figure 20.

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

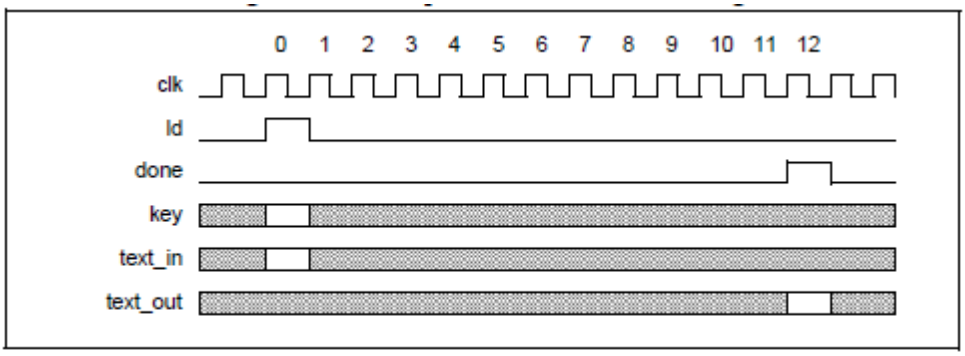

Figure 20. AES cipher core timing [25]

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.

5.2.3 do-zfiltering

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. [27]

5.2.4 bit_counter

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. [27]

5.2.5 Ethertype and TTL

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. As the packets are placed into the output-queues **`TTL`** is also decremented. [27]

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 Figure 21 and Figure 22, for *Moustique* and *AES* respectively.

5.3 5.3 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 *zFormation* i.e. 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. [27]

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. [27]

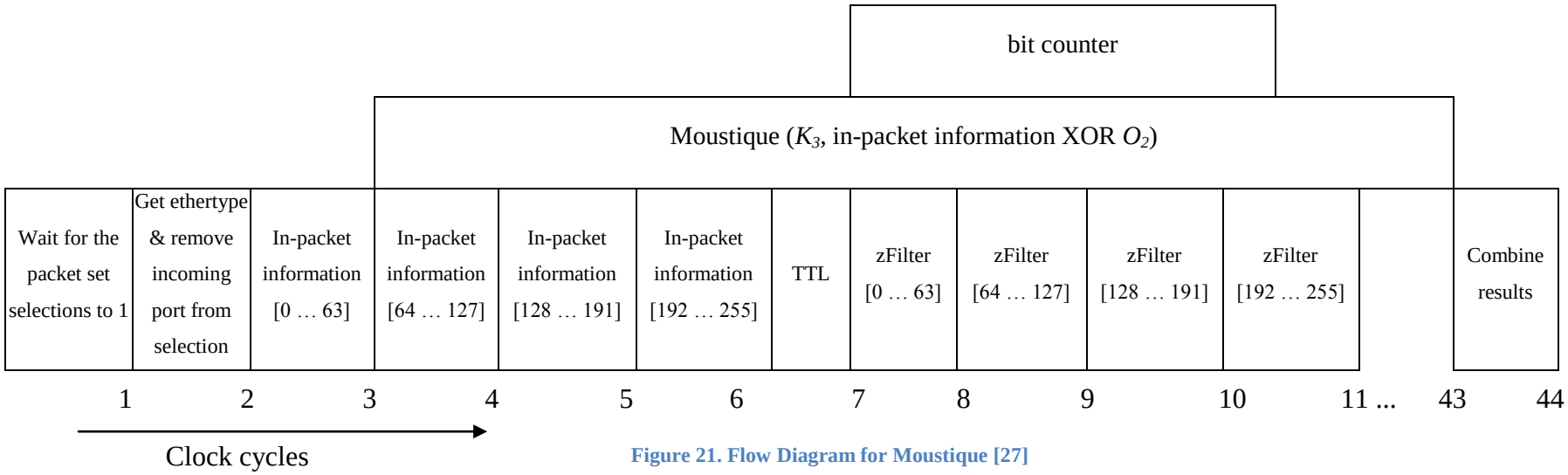

Figure 21. Flow Diagram for Moustique [27]

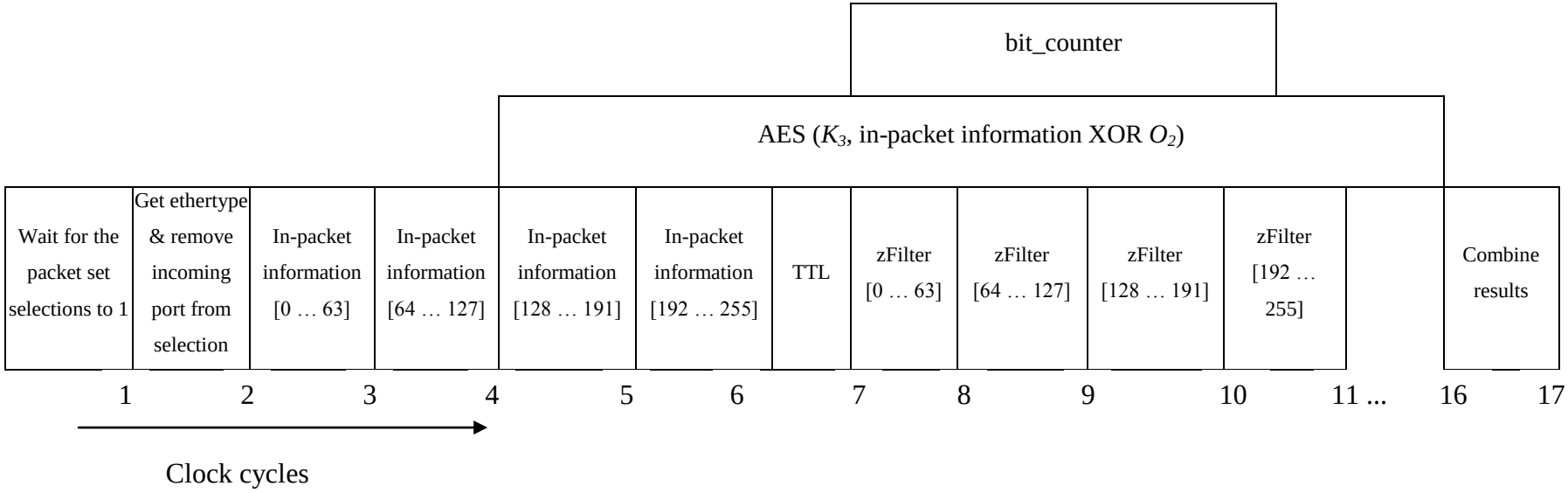

Figure 22. Flow Diagram for AES [27]

6
EVALUATION

In this chapter we will discuss several testing and verification results for the implementation. Two type of testing is given. One is regression tests and the other is simulation tests. The performance is also discussed on the basis of delay caused by each encryption technique for the computation of *zFormation*. This chapter also concludes the thesis.

6.1 Regression Tests

The regression tests verify the functionality of the hardware component of the *zFormation*. As there are two implementations. First, regression tests for *Moustique* and later the same steps for *AES*. In order to run the tests we need to connect the cables as described in each of the regression test below. All the procedure should be done while being root. The path and procedure for regression test is also defined at [26].

For these tests the bitfiles are downloaded into the NetFPGA and instructions can be found at [26][13]. In each of the regression tests some verification are done by sending pre-computed packets. These packets are sent using different ports and as per computation, expected to receive on particular ports and hence, verifying the correct implementation of the forwarding node. We have done testing for each and every interface individually to check the right implementation in the hardware. All these are defined in each of the regression test below. Each test checks for arrival of packets on port defined, *ethertype* check, *TTL* check and maximum number of 1s defined in the link IDs. Before sending the packets the values for O_2 and K_3 are also written into the registers of the NetFPGA using our management software.

Test 1: Test for receiving packets on each port (Broadcast)

Connect interfaces *eth1* and *eth2* of a sending node to any of the ports from *nf2c0* to *nf2c3* on the NetFPGA. The shell script is used to send ten packets from interface *eth1*. The pre-computed packet information iBF for this test includes all the link indices *M* on the path while calculating O_2. Hence, there should be a match for all the *output queues* and are received on all the ports on NetFPGA except the one from which it receives the packets. When packets are sent using interface *eth1* they are received on *eth2* and vice versa.

As an output received for this test, 10 packets were received from interface *eth2* when sent using interface *eth1* and vice versa. 10 packets were sent with wrong ethertype and we didn't receive any packet. For 10 packets, TTL was set to zero and again we didn't receive any packets. To test the limitation of No. of 1's in the iBF, 10 packets were sent with 1s exceeding the limitation so that it may not forward the packets sent by an attacker just by writing all 1s. Although there would be a match with the Link ID but the check for 1s doesn't allow the packets to get forward and we didn't receive any packets for this check also..

Hence, all the checks for ethertype, TTL, number of 1s limitation and broadcast passed.

Test 2: Test for receiving packets only on port 2 (nf2c1)

Connect interface *eth2* to *nf2c1* and eth1 to any other port. This test will send 10 packets from interface *eth1*. The pre-computed packet information iBF for this test matches with only *nf2c1* and *eth2* receives the packets. Here, only the path for interface *nf2c1* is added in O_2. This test will fail if *eth2* is connected to any other port.

In the output, we received 10 packets only when the receiving interface *eth2* was connected to *nf2c1*. All the three checks for ethertype, TTL and number of 1s exceeding limit were passed also. The test got failed when eth2 was connected to some other port than *nf2c1*. This verifies the right implementation of the *zFormation.*

Test 3: Test for receiving packets only on port 3 (nf2c2)

Connect interface *eth2* to *nf2c2* and *eth1* to any other port. This test will send 10 packets from eth1. The pre-computed packet information iBF for this test matches with only *nf2c2* and *eth2* receives the packets. This test will fail if *eth2* is connected to any other port.

In the output, we received 10 packets only when the receiving interface *eth2* was connected to interface *nf2c2* on the NetFPGA. All the three checks for ethertype, TTL and number of 1s exceeding limit were passed also. The test got failed when eth2 was connected to some other port than *nf2c2* since we didn't receive any packets on other nodes as the packets were not supposed to get forward.

Test 4: Test for receiving packets only on port 4 (nf2c3)

Connect interface *eth2* to *nf2c3* and *eth1* to any other port. This test will send 10 packets from *eth1*. The pre-computed packet information iBF for this test matches with only *nf2c3* and *eth2* receives the packets. This test will fail if *eth2* is connected to any other port.

In the output, we received 10 packets only when the receiving interface *eth2* was connected to interface *nf2c3* on the NetFPGA. All the three checks for ethertype, TTL and number of 1s exceeding limit were passed also. The test got failed when eth2 was connected to some other port than *nf2c3* since we didn't receive any packets on other nodes as the packets were not supposed to get forward.

Test 5: Test for receiving packets only on port 1 (nf2c0)

Connect *eth2* to *nf2c0* and *eth1* to any other port. This test will send 10 packets from eth1. The pre-computed packet information iBF for this test matches with only *nf2c0* and *eth2* receives the packets. This test will fail if *eth2* is connected to any other port.

In the output, we received 10 packets only when the receiving interface *eth2* was connected to interface *nf2c0* on the NetFPGA. All the three checks for ethertype, TTL and number of 1s exceeding limit were passed also. The test got failed when eth2 was connected to some other port than *nf2c0* since we didn't receive any packets on other nodes as the packets were not supposed to get forward.

As we discussed the output of all the regression tests, that verifies the correct implementation of our forwarding node in the hardware. All the tests got passed as were supposed to perform for each included path and other mentioned checks.

6.2 Tests for simulation

There are 10 different tests for simulation verification. These are written using Perl script and can be found at [26]. The software tool they need is ModelSim for testing of Verilog code. Each script has the capability to generate packets with computing the in-packet information separately for each packet using the cryptographic technique it uses. It calculates the in-packet information dependent upon keys K_1, K_2 and K_3 with O_1, O_2 and O_3. It writes the required information that is K_3 and O_2 values into the registers in NetFPGA. Caculates the *nonce I* and *F3*, packs them into the packet's header and sends using different ports.

First four tests are done to verify the broadcast functionality with verifying the packets are not received from the same port from where sent to avoid loops.

Test 1: Sending packets using port 1 and expecting on all other ports

Ten packets were sent using port 1 that is *nf2c0*. They are expected to arrive on all the other ports, except port 1 from which they are sent to avoid loops, as their paths are included when calculating the in-packet bloom filter (iBF).

The simulation output verified the arrival of ten packets on each output port except port 1 from where the packets were sent. This output verifies the correct functionality of the implementation for broadcast.

Test 2: Sending packets using port 2 and expecting on all other ports

Ten packets were sent using port 2 that is *nf2c1*. They are expected to arrive on all the other ports, except port 2 from which they are sent to avoid loops, as their paths are included when calculating the in-packet bloom filter.

The simulation output verified the arrival of ten packets on each output port except port 1 from where the packets were sent. This output verifies the correct functionality of the implementation for broadcast.

Test 3: Sending packets using port 3 and expecting on all other ports

Ten packets are sent using port 3 that is *nf2c2*. They are expected to arrive on all the other ports except port 3, from which they are sent to avoid loops, as their paths are included when calculating the in-packet bloom filter.

The output for this simulation showed arrival of ten packets on the ports except port 3 proving test 3 passed.

Test 4: Sending packets using port 4 and expecting on all other ports

Ten packets are sent using port 4 that is *nf2c3*. They are expected to arrive on all the other ports, except port 4 from which they are sent to avoid loops, as their paths are included when calculating the in-packet bloom filter.

The result for this test also got passed as all the ports received 10 packets each except port 4 where zero packets were received.

Test 5: Sending packets using port 1 and port 2 is not included in the path

Ten packets are sent using port 1 that is *nf2c0*. They are expected to arrive on all the other ports except the port 1 and port 2 because port1 is used to send the packets while port 2 is not included in the path while calculating iBF. Hence, there will be a mismatch at port 2. So no packets will arrive on port 1 and port 2.

The output of the simulation also showed the same results of receiving ten packets on port 3 and port 4 each and zero packets on port 1 and port 2.

Test 6: Sending packets using port 1 and port 3 is not included in the path

Ten packets are sent using port 1 that is *nf2c0*. They are expected to arrive on all the other ports except the port 1 and port 3 because port1 is used to send the packets

while port 3 is not included in the path while calculating iBF. Hence, there will be a mismatch at port 3. So no packets will arrive on port 1 and port 3.

The test also passed as ten packets were received on port 2 and port 4 and no packets were received on port 1 and port 3.

Test 7: Sending packets using port 1 and port 4 is not included in the path

Ten packets are sent using port 1 that is *nf2c0*. They are expected to arrive on all the other ports except the port 1 and port 4 because port1 is used to send the packets while port 4 is not included in the path while calculating iBF. Hence, there will be a mismatch at port 4. So no packets will arrive on port 1 and port 4.

The output of this simulation showed that this test also got passed by receiving ten packets on each port 2 and port 3 and no packets on port 1 and port 4.

Test 8: Test for Ethertype

Ten packets are sent using port 1 that is *nf2c0*. Ethertype is set different from `0xacdc`. Hence, no packets should arrive on any of the ports.

The output also verified the test by receiving zero packets on any of the ports. Hence, check for ethertype is working properly.

Test 9: Test for TTL

Ten packets are sent using port 1 that is *nf2c0*. TTL is set to 0. Hence, no packets should arrive on any of the ports.

The output of this simulation test verified the functionality by not getting any packet on any port.

Test 10: Test for different keys

Ten packets are sent using port 1 that is *nf2c0*. Key K_3 to calculate iBF is different from the key saved into the registers. There should be a mismatch and hence, no packets should arrive on any of the ports.

The output of this test verifies that the keys for particular flow for calculation of F_3 are working fine. If there is a mismatch in the secret key the traffic is not authorized and hence no forwarding takes place. Here, we didn't receive any packets on any of the port.

6.3 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 5 shows the measurement results with plain wire and one NetFGPA. The measurements were taken with *Moustique*, *AES* and then for *LIPSIN* [1] 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 21, 22 and 23 respectively. The readings were taken for 10 000 samples. [27]

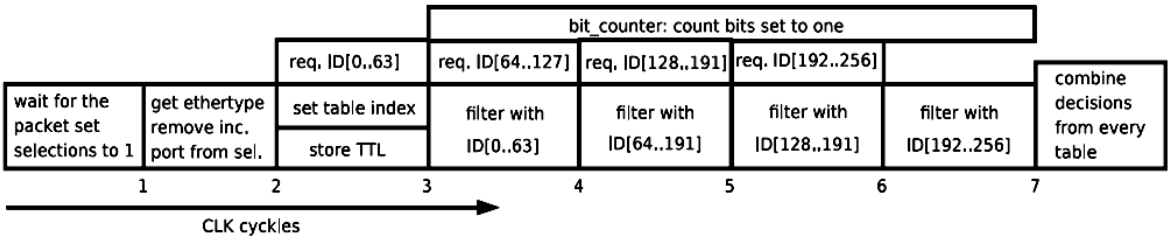

Figure 23. Flow diagram for LIPSIN [18]

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 6.1, 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. [27]

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 mean 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. [27]

In comparison to LIPSIN, although the delay time is greater for the computation of *zFormation* but it gives us more secure and dynamic way to compute the link IDs.

Path and packet format	Average Latency	Standard deviation
Wire (new)	*12,784ns*	*4,448.96ns*
NetFPGA with Moustique (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	*14,627ns*	*4,204.58ns*

Table 5. Latency measurement results [27]

7
CONCLUSION

In this thesis, we have described our early design and implementation for a source-routing-based forwarding mechanism. Initial goal was to construct the mechanism that is dynamic and secure. Along with providing these functionalities, to measure the delays caused by the forwarding node in computation of secure methods in real hardware.

The dynamic property was needed to avoid maintaining static tables to store the names of the outgoing links. Instead, forwarding node computes the forwarding link identifiers on the basis of some information from the packet, the link identifiers, and thereby the in-packet Bloom filters, may be made flow or packet contents dependent.

To ensure security, the design described in chapter 4 takes care of resistance against forwarding-identifier-guessing attacks. In a forwarding fabric based on in-packet bloom filters, only authorized 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 *Moustique* self-synchronizing stream cipher, and the other the *AES* block cipher function. Our initial assumption was to use self-synchronizing stream cipher because of their fast nature. For this, a single bit stream cipher *Moustique* was chosen. Contrary to our initial 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 practice. The measurement results obtained from the hardware implementation in the NetFPGA proved *AES* to be a better choice. 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.

8

REFERENCES

[1] P. Jokela, A. Zahemszky, C. Esteve, S. Arianfar, and P. Nikander. "LIPSIN: Line speed publish/subscribe inter-networking". In *SIGCOMM*, 2009.

[2] Object Management Group COBRA event service specification, version 1. 1. OMG Document formal/2000-03-01, 2001.

[3] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R., Kubiatowicz, J. "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination". In *11th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.

[4] Castro, M., Druschel, P., Kermarrec, A., Rowston, A. "Scribe: A large-scale and decentralized application-level multicast infrastructure". In *IEEE Journal on Selected Areas in Communications 20(8)*, October 2002.

[5] Altherr, M., Erzberg, M.,Mafieis, S. "iBus: A software bus middleware forthe java platform". In *Proceedings of the International Workshop on Reliable Middleware Systems*, pp. 49-65, October 1999.

[6] SIENA Web Site: http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-946-03.pdf (5/7/2011)

[7] Preotiuc-Pietro, R., Pereira, J., Llirbat, F., Fabret, F., Ross, K., Shasha, D. "Publish/subscribe on the web at extreme speed". In *Proc. of ACM SIGMOD, Conf. on Management of Data. Cairo, Egypt,* 2000.

[8] Gruber, R. E., Krishnamurthy, B., Panagosf, E. "The architecture of the readyevent noti_cation service". In *Proceedings of The International*

Conference on Distributed Computing Systems, Workshop on Middleware, Austin, Texas, 1999.

[9] Gryphon Web Site: http://www.research.ibm.com/distributedmessaging/gryphon.html (5/7/2011)

[10] Zahemszky, A. Csaszar, P. Nikander, and C. Esteve. "Exploring the pubsub routing/forwarding space". In *International Workshop on the Network of the Future*, 2009.

[11] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo, "NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing". In *MSE '07:Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[12] G. Adam Covington, Glen Gibb, Jad Naous, John W. Lockwood, Nick McKeown "Encouraging Reusable Network Hardware Design". In proceedings of *Microelectronic Systems Education, 2009. MSE '09. IEEE International Conference on* 25-27July 2009.

[13] NetFPGA User Guide: http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/Guide (5/7/2011)

[14] Aiden A. Bruen, Mario A. Forcinito, "Cryptography, Information Theory and Error-Correction: a handbook for the 21st century". *Wiley Interscience Publication,* 2005.

[15] Secure Hash Algorithm on Wikipedia: http://en.wikipedia.org/wiki/SHA-2 (5/7/2011)

[16] Joan Daemen, Paris Kitsos, "The self-synchronizing stream cipher MOSQUITO": *eSTREAM documentation, version 2,* 2005

[17] Joan Daemen, Paris Kitsos, "The self-synchronizing stream cipher MOUSTIQUE, 2006

[18] Jari keinänen, Petri Jokela, Kritistian Slavov, “Implementation of zFilter based forwarding node on a NetFPGA”, *NetFPGA Developers Workshop*, August 13-14, 2009, Stanford, CA.

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

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

[21] 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.

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

[23] Jad Naous, Glen Gibb, Sara Bolouki and Nick McKeown, “NetFPGA: Reusable Router Architecture for Experimental Research”. In *PRESTO ’08: Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[24] OpenCores AES project: http://opencores.org/project,aes_core (5/7/2011)

[25] Rudolf Usselmann, “Advanced Encryption Standard/Rijndael IP core” Rev. 1.1. 2002

[26] zFormation on a NetFPGA wikipage: http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/ZFormationPSrouter (5/7/2011)

[27] Adnan Hassan Ghani and Pekka Nikander, “Secure In-Packet Bloom Filter Forwarding on a NetFPGA” in *1st European NetFPGA Developers Workshop*, University of Cambridge, Computer Laboratory, Cambridge, UK, Sep. 9–10th, 2010.

APPENDIX A

Pseudocode for the SHA-256 algorithm follows.

Note 1: All variables are unsigned 32 bits and wrap modulo 2^{32} when calculating
Note 2: All constants in this pseudo code are in big endian

Initialize variables
(first 32 bits of the fractional parts *of the square roots of the first 8 primes 2..19):*

```
h[0..7] :=
   0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f,
0x9b05688c, 0x1f83d9ab, 0x5be0cd19
```

Initialize table of round constants
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):

```
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,
0x59f111f1, 0x923f82a4, 0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74,
0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f,
0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3,
0xd5a79147, 0x06ca6351, 0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354,
0x766a0abb, 0x81c2c92e, 0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,
0xd6990624, 0xf40e3585, 0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3,
0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa,
0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

Pre-processing:

```
append the bit '1' to the message
append k bits '0', where k is the minimum number >= 0 such that the
resulting message
    length (in bits) is congruent to 448 (mod 512)
append length of message (before pre-processing), in bits, as 64-bit
big-endian integer
```

Process the message in successive 512-bit chunks:

```
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[0..15]
```

Extend the sixteen 32-bit words into sixty-four 32-bit words:

```
    for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18)
xor (w[i-15] rightshift 3)
```

```
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor
(w[i-2] rightshift 10)
        w[i] := w[i-16] + s0 + w[i-7] + s1
```

Initialize hash value for this chunk:

```
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7
```

Main loop:

```
    for i from 0 to 63
        s0 := (a rightrotate 2) xor (a rightrotate 13) xor (a
rightrotate 22)
        maj := (a and b) xor (a and c) xor (b and c)
        t2 := s0 + maj
        s1 := (e rightrotate 6) xor (e rightrotate 11) xor (e
rightrotate 25)
        ch := (e and f) xor ((not e) and g)
        t1 := h + s1 + ch + k[i] + w[i]
        h := g
        g := f
        f := e
        e := d + t1
        d := c
        c := b
        b := a
        a := t1 + t2
```

Add this chunk's hash to result so far:

```
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h
```

Produce the final hash value (big-endian):

```
digest = hash = h0 append h1 append h2 append h3 append h4 append h5
append h6 append h7
```