HE-MT6D: A Network Security Processor with Hardware Engine for Moving Target IPv6 Defense (MT6D) over 1 Gbps IEEE 802.3 Ethernet

Joseph Lozano Sagisi

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Engineering

Joseph G. Tront, Chair

Patrick R. Schaumont

Randolph C. Marchany

May 10, 2017

Blacksburg, Virginia

Keywords: IPv6 Security, Moving Target Defense, Network Security Processor, FPGA, Moving Target IPv6 Defense, System on Chip, Packet Processor

Copyright 2017, Joseph L. Sagisi

HE-MT6D: A Network Security Processor with Hardware Engine for Moving Target IPv6 Defense (MT6D) over 1 Gbps IEEE 802.3 Ethernet

Joseph L. Sagisi

ABSTRACT

Traditional static network addressing allows attackers the incredible advantage of taking time to plan and execute attacks against a network. To counter, Moving Target IPv6 Defense (MT6D) provides a network host obfuscation technique that dynamically obscures network and transport layer addresses. Software driven implementations have posed many challenges, namely, constant code maintenance to remain compliant with all library and kernel dependencies, less than optimal throughput, and the requirement for a dedicated general purpose hardware. The work of this thesis presents Network Security Processor and Hardware Engine for MT6D (HE-MT6D) to overcome these challenges. HE-MT6D is a soft core Intellectual Property (IP) block developed in full Register Transfer Level (RTL) and is the first hardware-oriented design of MT6D. Major contributions of HE-MT6D include the complete separation of the data and control planes, development of a nonlinear Complex Instruction Set Computer (CISC) Network Security Processor for in-flight packet modification, a specialized Packet Assembly language, a configurable and a parallelized memory search through tag-based Hybrid Content Addressable Memory (HCAM) L1 write-through cache, full RTL Network Time Protocol v4 hardware module, and a modular crypto engine. HE-MT6D supports multiple nodes and provides 1,025% throughput performance increase

over earlier C-based MT6D at 863 Mbps with full encapsulation and decapsulation, and it matches bare wire throughput performance for all other traffic. The HE-MT6D IP block can be configured as an independent physical gateway device, built as embedded Application Specific Integrated Circuit (ASIC), or serve as a System on Chip (SoC) integrated submodule.

HE-MT6D: A Network Security Processor with Hardware Engine for Moving Target IPv6 Defense (MT6D) over 1 Gbps IEEE 802.3 Ethernet

Joseph L. Sagisi

GENERAL AUDIENCE ABSTRACT

Traditional static network addressing allows attackers the incredible advantage of taking time to plan and execute attacks against a network. One approach to counter this effect is dynamic addressing through Moving Target Defense, which the Department of Homeland Security Cyber Security Division (CSD) designated as one of the fourteen primary Technical Topic Areas for securing federal networks and the larger Internet. A specific application for Internet Protocol version 6 (IPv6) networks is Moving Target IPv6 Defense (MT6D). This provides tunneling and dynamic cryptographic network address translation, where new addresses are cryptographically generated every few seconds. The work of this thesis presents a Network Security Processor and Hardware Engine for MT6D (HE-MT6D). HE-MT6D is the first hardware-oriented implementation of MT6D developed in full Register Transfer Level (RTL) logic and provides 1,025% performance increase over earlier C-based MT6D at 863 Mbps full duplex throughput. It also provides support for multiple nodes. The HE-MT6D Intellectual Property (IP) block is modular for maximum flexibility towards system deployment: it can be configured as an independent physical gateway device, built as embedded Application Specific Integrated Circuit (ASIC), or serve as a System on Chip (SoC) integrated submodule.

Acknowledgments

I would like to express my deep gratitude to Dr. Joseph G. Tront, Randy C. Marchany, and Patrick R. Schaumont for their mentorship and inspiration throughput my years of research at Virginia Tech. I would also like to thank my colleagues at the IT Security Office and Lab (current and past) for their unwavering support, patience, wisdom, and general humor, which made daily life at the beautiful campus of Virginia Tech even more pleasurable. Also, special thanks goes to my additional thesis proof readers Mark DeYoung and Sarah Talty for their patience and feedback. I would like to thank the academic mentors of my younger years for helping cultivate within me the seed of curiosity: Mrs. Sherri Houp, Dr. Michael Mosley, Dr. Scot Ransbottom, and Dr. Michael Brownfield, to name a few. To my family and friends, thank you for your undying love and support.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Background and Motivation 3

1.3 Moving Target IPv6 Defense (MT6D) Concepts 4

1.4 Related Work 9

1.5 Research Objectives 11

1.6 Research Questions 12

1.7 Methodology and Organization of Thesis 13

2 Feasibility Study 1: Embedded MT6D **16**

2.1 Embedded MT6D 17

2.2 Lower Layer Integration: Physical and Data Link Layers 21

2.2.1 Placement of Hardware-based MT6D 22

2.2.2 Understanding and Maximizing Throughput 25

2.2.3 Adjustments 27

2.3 Conclusion 28

3 Feasibility Study 2: IPV6 Communications 29

3.1 Internet Protocol version 6 (IPv6) 30

3.1.1 Addressing model 31

3.1.2 Resolving MAC Addresses 33

3.1.3 IPv6 Packet Structure 34

3.1.4 User Datagram Protocol (UDP) 35

3.1.5 Internet Control Message Protocol for IPv6 (ICMPv6) Type 1-4 Errors 36

3.2 Packet Case Types 38

3.2.1 Unmodified Communication Across a Router 39

3.2.2 Selective Modification of Multiple IPv6 Communication Protocols . . 41

3.2.3 Accounting for 13 Different Packet Case Types 46

3.3 Conclusion 47

4 Design: Overview **49**

4.1 Overall Architectural Model 50

4.2 Control and Data Plane Separation 53

4.3 Stream-based Packet Buffering 54

4.4 Process Flows 56

4.4.1 Time Synchronization 56

4.4.2 System Initialization 57

4.4.3 System Maintenance 59

4.4.4 Processing Datapath 61

4.5 Conclusion 63

5 Implementation: a System of Systems **64**

5.1 Statistics and initialization Subsystem 66

5.2 Time Subsystem. 66

5.3 Rotation Coprocessor 69

5.4 Hash Engine 73

5.5 Memory Subsystem 74

5.5.1 Reconfigurable Hybrid Content Addressable Memory (CAM) (HCAM) 75

5.5.2 Shared Memory 80

5.6 MT6D Processing Cores 81

5.6.1 Field Extractor Module 82

5.6.2 Broker Module 83

5.6.3 Packet Assembler Module 87

5.6.4 Processing an Example Packet 93

5.7 Datapath 94

5.8 Conclusion 96

6 Evaluation of Hardware Engine for HE-MT6D) 98

6.1 Evaluation Platform 98

6.2 Network Communications Performance 100

6.3 Memory Search 107

6.4 Hash Engine Performance 110

6.5 FPGA Resources 111

6.6 MT6D Performance 112

6.7 Summary and Future Work 116

6.8 Conclusion 118

7 Conclusion and Future Work **119**

7.1 Conclusion 119

7.2 Future Work 120

Bibliography **122**

Appendix A Code Repository **129**

Appendix B Network Security Processor and Hardware Engine for MT6D (HE-MT6D) Architecture **130**

List of Figures

1.1 The similarity betwen Moving Target IPv6 Defense (MT6D) and Frequency Hopping Spread Spectrum (FHSS) technologies. 5

1.2 The MT6D tunneling process. 8

1.3 Optional payload encryption feature. 8

1.4 MT6D deployment strategies. 9

2.1 Embedded MT6D design architecture. 18

2.2 Placement of Network Security Processor and Hardware Engine for MT6D (HE-MT6D) in the Reconciliation Sublayer (RS) between the Data Link and Physical layers of the IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Local Area Network (LAN) model. 23

2.3 Illustration of HE-MT6D implemented in-line with the Reduced Gigabit Media-Independent Interface (RGMII) 25

2.4 Size of and fields within a basic Ethernet Frame. 25

3.1 IPv6 address format. 32

3.2 Standard IPv6 packet format. 34

3.3 The User Datagram Protocol (UDP) Extension Header (EH) format. 35

3.4 A typical Internet Control Message Protocol for IPv6 (ICMPv6) Type 1-4 Error Packet. 37

3.5 ICMPv6 Error types. 38

3.6 Packet exchanges used to start and continue communication over IPv6. . . . 39

3.7 IPv6 packet transformations with HE-MT6D gateways inserted. 42

4.1 The basic topology of HE-MT6D. 51

4.2 Overall system architecture with separation of data and control planes. . . . 51

4.3 General concept of in-stream processing. 55

4.4 Time synchronization process flow. 56

4.5 System initialization process flow. 58

4.6 System maintenance process flow. 60

4.7 Datapath process flow. 62

5.1 Expanded system architecture. 65

5.2 Initialization and Statistics subsystem. 66

5.3 The Time subsystem. 66

5.4 NTP time formats. 67

5.5 Network Time Protocol v4 (NTPv4) Module design. 68

5.6 NTPv4 packet format. 69

5.7 Rotation Coprocessor Subsystem. 70

5.8 Modular Hash Engine design. 74

5.9 The Hybrid Content Addressable Memory (HCAM) Module Design. 76

5.10 Overview of an HCAM internal search engine. 78

5.11 HCAM internal search engine in detail. 78

5.12 On-chip shared memory. 80

5.13 MT6D packet processing core design. 81

5.14 The Broker Module. 83

5.15 Single-stage pipeline nonlinear Packet Assembler Module. 87

5.16 The Datapath. 94

6.1 Evaluation platform. 99

6.2 Experimental setup diagrams. 101

6.3 User Datagram Protocol (UDP) connectionless performance results. 102

6.4 Transmission Control Protocol (TCP) connection-oriented performance results. 103

6.5 Overflow buffer statistics taken after bi-directional UDP throughput testing. 104

6.6 Walking through a Translation Lookup Request (TLRQ) 108

6.7 Datapath and the Translation Lookup Request (TLRQ) and Translation Lookup Response (TLRS) buffers. 110

6.8 Hash engine performance. 111

6.9 Wireshark capture of `ping` over MT6D. 112

6.10 A visual representation of IPv6 addresses captured from constant pings between two nodes over HE-MT6D during a course of six hours. 113

6.11 SignalTap capture of a packet being decapsulated, Part 1. 114

6.12 SignalTap capture of a packet being decapsulated, Part 2. 114

B.1 System Architecture. 131

List of Tables

1 Notation xxv

2.1 Enumeration of the functional dependencies required to run MT6D on top of the Linux kernel 20

2.2 Basic Ethernet Frame Measurements 26

3.1 Thirteen currently identified packet case types. 47

5.1 The Rotation Table. 71

5.2 Shared Routing Table (SRT) data structure entries. 72

5.3 Reconciliation Decision Matrix. 89

5.4 Fundamental Reduced Instruction Set Computer (RISC) instructions. 90

5.5 CISC instructions for hardware execution. 91

5.6 Packet Assembly Trajectories. 92

6.1 Field-Programmable Gate Array (FPGA) Resources consumed by HE-MT6D with both inbound and outbound HCAMs set to 7 bits of collision resistance. 111

Listings

2.1 Embedded MT6D initialization and rotation maintenance. 20

5.1 Reconciliation Cycle 89

Acronyms

6LoWPAN IPv6 over Low power Wireless Personal Area Networks. 10

ARP Address Resolution Protocol. 33

ASIC Application Specific Integrated Circuit. iii, iv, 2, 3, 11–13, 16, 24, 25, 28, 63, 96, 116, 119

Avalon-ST Avalon Streaming interface. 55, 94

CAM Content Addressable Memory. viii, 50, 75

CC Channelizer and Checksum arbiter. 95, 96

CIDR Classless Inter-Domain Routing. 31

CISC Complex Instruction Set Computer. ii, 47, 87, 90–92, 117, 120

CSD Cyber Security Division. 2

CSMA/CD Carrier Sense Multiple Access with Collision Detection. xi, 22, 23

DDIO Double Data Rate I/O. 18, 19

DHCP Dynamic Host Configuration Protocol. 4, 31

DHS Department of Homeland Security. 2

DHT Distributed Hash Table. 10

DLL Data Link Layer. 22, 27, 28

DNS Domain Name Server. 30, 31

DRAM Dynamic RAM. 54

EH Extension Header. xii, 7, 8, 35–37, 82, 109

EPD Error Packet Discard. 95

FHSS Frequency Hopping Spread Spectrum. xi, 4, 5

FIFO first in, first out. 50, 55, 81, 94, 95, 105

FOVF Final Overflow Buffer. 95, 104, 116

FPGA Field-Programmable Gate Array. xvi, 2, 3, 17, 26, 54, 64, 99, 100, 111, 117, 118

GBIC Gigabit Interface Converter. 24

GMII Gigabit Media-Independent Interface. 23, 24

HAL Hardware Abstraction Layer. 17, 20

HCAM Hybrid Content Addressable Memory. ii, viii, xiii, xvi, 59, 70, 75–81, 83, 84, 96, 107–109, 111, 117, 120

HDL Hardware Description Language. 17

HE-MT6D Network Security Processor and Hardware Engine for MT6D. ii–iv, ix–xii, xiv, xvi, 3, 9, 11–17, 22–30, 35, 36, 38, 39, 41–43, 46–49, 51–54, 56–58, 60, 64–67, 73, 81, 87, 94, 96, 98–121, 130, 131

ICMPv6 Internet Control Message Protocol for IPv6. vii, xii, 7, 14, 32, 34–41, 43–47, 52, 86, 93, 107, 109

IID Interface Identifier. 6, 31–33, 59, 72, 73, 75, 76, 79, 84, 85, 106, 108, 114

IoT Internet of Things. 1, 3, 10

IP Intellectual Property. ii–iv, 3, 9, 19, 25, 73, 96, 99, 119

IPsec Internet Protocol Security. 11, 35, 64

IPv4 Internet Protocol version 4. 1, 4, 30–33

IPv6 Internet Protocol version 6. iv, vii, xii, xiv, 1, 2, 4–8, 13, 14, 19, 21, 28–49, 52, 59, 63, 65, 75, 77, 81–85, 91, 93, 94, 96, 106, 107, 109, 113–115, 117, 118, 120

ISA Instruction Set Architecture. 90, 120

ISP Internet Service Provider. 32

JTAG Joint Test Action Group. 19, 100

LAN Local Area Network. xi, 23

M2M Machine-to-Machine. 1

MAC Media Access Control. vii, 8, 17, 19, 22, 25, 27, 31, 33–35, 40, 58, 66, 91, 95, 106, 113

MDIO Management Data Input/Output. 95

MII Media Independent Interface. 9, 23, 24, 28

MT6D Moving Target IPv6 Defense. ii–iv, vi, ix, xi, xiii–xv, 2–5, 7–14, 16–29, 35, 36, 41–45, 47, 51–53, 57, 58, 63, 65, 67, 70–73, 76, 80–82, 86, 89, 91, 93, 96, 100–104, 108–110, 112–117, 119–121

MTD Moving Target Defense. 2

MTU Maximum Transmission Unit. 27, 34, 35, 37, 38, 40, 44, 47, 86, 91, 93

NA Neighbor Advertisement. 33, 34, 40, 41, 106

NDP Neighbor Discovery Protocol. 30, 33, 39, 41, 106, 107

NIC Network Interface Card. 9, 100

Node ID Node Identifier. 63, 71, 77–79, 83, 84, 108

NS Neighbor Solicitation. 33, 40, 41, 106

NSP Network Security Processor. 2, 10, 11, 13, 49, 96, 117, 120

NSTC National Science and Technology Council. 2

NTP Network Time Protocol. xiii, 57, 67

NTPv4 Network Time Protocol v4. ii, xiii, 8, 31, 56–58, 61, 66–69, 94, 120

OS Operating System. 11, 20

OSI Open Systems Interconnection. 13, 16, 22–24, 67, 69

OVF Overflow Buffer. 95, 96

PCS Physical Coding Sublayer. 24

PHY physical. 9, 17, 19, 22, 24, 27, 28, 58, 66, 95, 96

PLL Phase-Locked Loop. 18, 19, 98

PMA Physical Medium Attachment. 24

PMD Physical Medium Dependent. 24

pps packets per second. 26

RA Router Advertisement. 32, 33

RGMII Reduced Gigabit Media-Independent Interface. xi, 24, 25, 95, 96

RISC Reduced Instruction Set Computer. xv, 17, 90

RMII Reduced Media-Independent Interface. 24

RS Router Solicitation. 32

RS Reconciliation Sublayer. xi, 22–25, 28

RTL Register Transfer Level. ii, iv, 2, 3, 12–14, 16, 17, 20, 28, 49, 54, 118–120

RX receive. 51–53

SFD Start of Frame Delimiter. 25

SFP Small Form-factor Pluggables. 24

SG-DMA Scatter-Gather Direct Memory Access. 18–20, 54, 55, 88

SGMII Serial Gigabit Media-Independent Interface. 24

SHA256 Secure Hash Algorithm. 19, 73, 74, 110, 120

SLAAC Stateless Address Auto-Configuration. 32

SoC System on Chip. iii, iv, 3, 10, 119

SRAM Static RAM. 54

SRT Shared Routing Table. xv, 6, 7, 19, 52, 58–60, 62, 70, 72, 75, 80, 81

SSH Secure Shell. 115

SSL Secure Sockets Layer. 11, 64

TCP Transmission Control Protocol. xiv, 34, 39, 41, 44, 102, 103, 116

TLRQ Translation Lookup Request. xiv, 36, 83, 84, 108–110

TLRS Translation Lookup Response. xiv, 83, 84, 108–110

TSE Triple Speed Ethernet. 19, 20, 95, 96, 98

TX transmit. 51–53, 104

UART Universal Asynchronous Receiver/Transmitter. 19, 100

UDP User Datagram Protocol. vii, xii–xiv, 6–9, 34–36, 41, 54, 64, 69, 77, 82, 91, 92, 101, 102, 104, 105, 107, 109, 110, 112, 116

UTP unshielded twisted pair. 22, 103

VLAN Virtual LAN. 27

XGMII 10-Gigabit Media-Independent Interface. 24

Table 1: Notation

Symbol	Definition
d	direction $\in$ D={O for Outbound, I for Inbound}
$\mathcal{P}_d$	Original packet with direction d
$\mathcal{P}'_d$	MT6D encapsulated packet with direction d
$\{a_s, p_s\}$	144-bit address set of both the 128-bit IPv6 IP address and corresponding potential 16-bit MT6D port within p_s packet $\mathcal{P}_d$
$\mathcal{A}$	The set of all 144-bit addresses $\{a_0, a_1, ..., a_f\}$ within packet $\mathcal{P}_d$, where f is the last set. Typically, a_0 is the source address set, a_1 is the destination address set, a_2 is theICMPv6 Type 1-4 error invoked source address, and a_3 is the ICMPv6 Type 1-4 error invoked destination address
n_i	Node ID for node i, $n_i \in \mathcal{N}$
$\mathcal{N}$	The set of all protected node IDs
α'_i	$= \{\sigma_i, \phi_{i,j(t)}\}$, full 144-bit MT6D IPv6 address with UDP port of node i
$Q_d(a_n)$	MT6D lookup request on address set a_n
$R_d(a_n)$	MT6D lookup response on address set a_n
$\mathbf{P}_i$	$=\{\sigma_i, k_i, \delta_i\}$ Profile of node i
α_i	$= \{\sigma_i, \beta_i\}$, full 128-bit IPv6 address of node i
σ_i	Node i's subnet
β_i	Node i's original Interface Identifier (IID)
δ_i	Node i's rotation interval
k_i	Node i's session key
h_i	Hash select. Used to select the hashing algorithm to be used.
U	Current Unix time
j_{max}	Number of rotations supported per node t
$j(t)$	Rotation number at time t. j_{prev}, j_{curr}, j_{next} are also used to describe the previous, current, and next rotation pointers.
val_j	Valid attribute of rotation j
val	Array of all valid bits val_j
$t_{exp}(i)$	Current expiration time of n_i
$\epsilon_i(t)$	$= U - U \mod \delta_i$, border floor time at time t
$\phi_{i,j(t)}$	Node i's corresponding translated IID pair at time t. The IID pair includes both the IID and UDP port
H	Suite of cryptographic hash functions
w_n	HCAM bank width (in number of nodes)
$\eta_{d,r,c}$	tag located at row r and column c in HCAM with direction d
c_d	column pointer for HCAM hit with direction d
r_d	row pointer for HCAM hit with direction d
μ_d	number of collision resistance tag bits chosen for HCAM with direction d

Chapter 1

Introduction

Connecting billions of smart objects is one of the biggest digital revolutions of the 21st century, known as the Internet of Things (IoT) [42, 8]. Direct Machine-to-Machine (M2M) connections provide cyber-physical bridging of intelligent devices that increasingly automate the world we live in. Applications are vast and growing, and IoT nodes are already seen in smart power grids, retail, public service management, e-healthcare, home area networks, intelligent transportation systems, utility management, environmental monitoring, smart cities, and industrial automation [49, 4]. Enabling these devices on a massive scale is Internet Protocol version 6 (IPv6), as its predecessor Internet Protocol version 4 (IPv4) simply does not have the address space. One of the features of and drawbacks to IPv6 is its goal to provide end-to-end transparency, whereby any globally routable interface address can be reached from anywhere in the world [12]. This simplifies device connections and fosters technological innovation in areas such as peer-to-peer communication but also raises security concerns and

challenges [42]. Security is never a single solution, but rather, a multi-layered defense-in-depth technique. One such layer of defense is Moving Target Defense (MTD), which seeks to make attack surfaces less deterministic, less homogeneous, and less static [40]. The National Science and Technology Council (NSTC) and the US Department of Homeland Security (DHS) Cyber Security Division (CSD) both recognize MTD as a technical topic area for national cybersecurity defense strategy [40, 14].

Previous work has been done on a specific MTD technique called Moving Target IPv6 Defense (MT6D) that performs MTD with IPv6 cryptographic network address translation and tunneling. The research presented in this thesis seeks to advance MT6D beyond the current state of the art to facilitate Application Specific Integrated Circuit (ASIC) design of this technology and eventual widespread deployment to provide additional defense in depth to IPv6 networks.

1.1 Problem Statement

This research asks the question: is it possible to implement MT6D in full Register Transfer Level (RTL) logic as a gateway device, and on top of that, is it possible to do so at full line rate speeds? This thesis hypothesizes that the implementation is possible and that the speed condition can be met with the current technologies.

To support this hypothesis, the goal of this research is to design, implement, and evaluate an MT6D RTL Network Security Processor (NSP) hardware gateway in an Field-Programmable

Gate Array (FPGA) as this prototyping device enables future transition of MT6D into an ASIC.

1.2 Background and Motivation

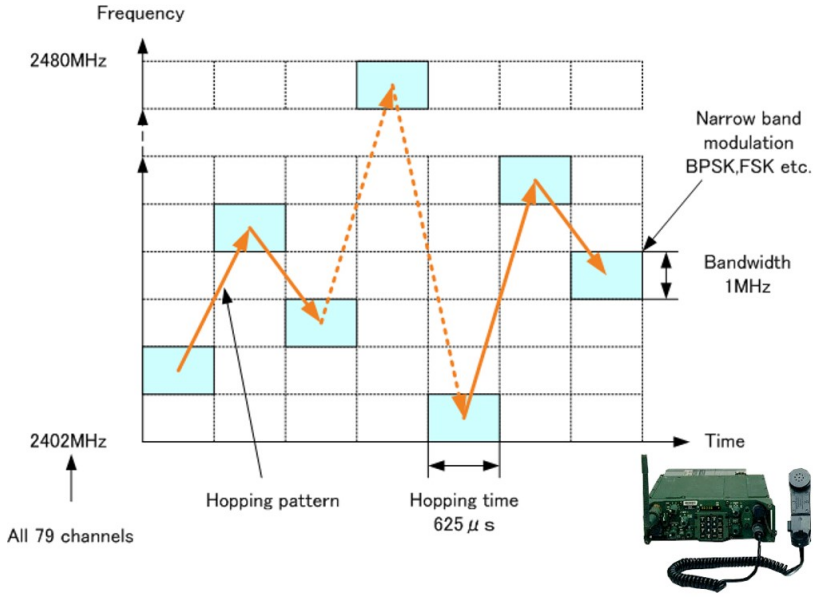
All developments of MT6D so far have been software implementations, the downside of which are dependency limitations, physical size, process efficiency, and throughput. For example, Hardman's implementation requires `iptables` no later than v1.4.10 and is limited to a 32-bit computing architecture due to requirements of the library `libssl` [24]. Additionally, the computing platform used in that specific implementation had dual 1 Gbps Ethernet ports but the system was only able to push a maximum of 167.4 Mbps, even without MT6D processing applied. Server grade equipment can be used but would increase deployment and maintenance costs. An ideal solution would be a low cost ASIC that would intercept and modify a standard protocol, which can then be built as embedded hardware or a standalone network device. A lightweight ASIC design can find itself embedded in IoT devices, routers, or other consumer electronics. In order to develop an ASIC, an RTL-based design must first be accomplished. HE-MT6D meets this need and provides more: HE-MT6D is a modular Intellectual Property (IP) block that can be configured not only as an fully independent ASIC gateway device, but also built as embedded ASIC as well as a System on Chip (SoC) integrated submodule.

1.3 Moving Target IPv6 Defense (MT6D) Concepts

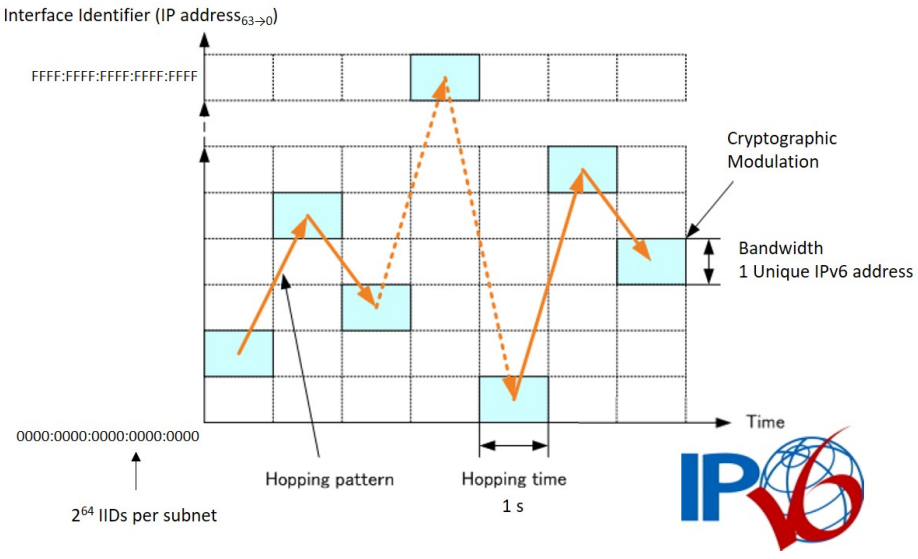
Most computing environments today rely on static network addressing schemes, which are vulnerable to long term probing, observation, and attack. Dynamic Host Configuration Protocol (DHCP) systems exist, but even these are still relatively static due to long lease times. MT6D presents a paradigm shift by constantly moving the logical location of a target, obfuscating the network attack surface, every few seconds. This movement is akin to Frequency Hopping Spread Spectrum (FHSS) in the radio communications field, whereby a sender and receiver's channel frequency hops in a psuedorandom manner upwards of 100 times a second as compared in Fig. 1.1.

MT6D was developed to apply the same concept over the vastness of IPv6 address space, albeit with a much slower hopping period of every few seconds, making it unlikely for an attacker to observe traffic long enough to identify and attack a protected IPv6 interface—if the attacker can even rendezvous with it in the first place. IPv6 is much more vast than its predecessor protocol. IPv4 addresses are 32-bits wide (4 billion combinations), while IPv6 addresses are 128-bits wide, which provides for $3.4 \cdot 10^{38}$ combinations.

MT6D is a tunneling protocol [16]. It is peer-to-peer based and assumes knowledge of a shared session key k_i, the unobscured protected interface addresses α_i, and a shared time rotation interval δ_i. This tuple $\{k_i,\ \alpha_i,\ \delta_i\}$ is known as the profile $\mathbf{P}_i$ of the protected node [16]. Each protected host has an encapsulation and decapsulation process. During the encapsulation process, the original headers of packet $\mathcal{P}_d$ and the rest of the packet are



(a)



(b)

Figure 1.1: The similarity betwen Moving Target IPv6 Defense (MT6D) and Frequency Hopping Spread Spectrum (FHSS) technologies. (a) FHSS modulates its communications medium frequency over time in order to provide resistance against narrow band interference, eavesdropping, and signal jamming [9]. (b) Similarly, MT6D modulates the transmission medium Internet Protocol version 6 (IPv6) address space over time to provide resistance against surveillance, unauthorized accesses, and denial of service attacks.

packaged as a User Datagram Protocol (UDP) datagram [16]. UDP is a connectionless transport protocol. The IPv6 header source and destination address of the encapsulated MT6D header consists of the original subnetwork prefix σ_i and a cryptographically obscured IID pair $\phi_{i,j(t)}$, derived as a function of the original base IID β_i, a shared session key k_i, and the current floor time border $\epsilon_i(t)$ where $j(t)$ is the active rotation at some time t [16].

Given the original 128-bit IPv6 address of a given node

$$\alpha_i = \{\sigma_i, \beta_i\} \tag{1.1}$$

The new IID pair is constructed as

$$\phi_{i,j(t)} = H(\beta_i || k_i || \epsilon_i(t))_{0 \to 63} \tag{1.2}$$

And new 144-bit IPv6 address and UDP port concatenation as

$$\alpha_i' = \{\sigma_i, \phi_{i,j(t)}\} \tag{1.3}$$

The border floor time is calculated as the current modular period of Unix time based on the profile rotation time δ_i:

$$\epsilon_i(t) = U - U \mod \delta_i \tag{1.4}$$

To maintain rotation records, a Shared Routing Table (SRT) maintains at least the previous,

current, and next rotation pairs $\phi_{i,j_{prev}}$, $\phi_{i,j_{curr}}$, $\phi_{i,j_{next}}$ calculated for $n_i \in \mathcal{N}$ at the time floor $\epsilon_i(t+\delta_t)$, where n_i represents each protected node interface within the set of of all protected node interfaces $\mathcal{N}$, t is some specified Unix time, and δ_i is n_i's rotation interval [16]. The SRT also holds the configuration data for each node: the original base address σ_i, β_i, symmetric session key k_i, rotation interval δ_i.

The encapsulation process is shown in Fig. 1.2. The standard packet will have two IPv6 addresses (basic source and destination), but it is important to realize that packets may have several more nested within Extension Header (EH) options and upper layer payloads. For example, Internet Control Message Protocol for IPv6 (ICMPv6) error packets (addressed in Section 1.5) include as much of a copy of its invoking packet as possible and should be accounted for to prevent data leakage as well as as to facilitate the IPv6 communications protocol. Only the packet types identified in Chapter 3 are considered for the scope of the research presented in this document.

It is important to realize that a packet may have up four or more IPv6 address sets, a_s, but the standard packet will have two, as basic IP header source and destination address. Any ICMPv6 error packets have additional set within its payload as will be discussed in Section 3.1.5.

The MT6D payload can be optionally encrypted to prevent deep packet inspection correlation by an attacker as shown in Fig. 1.3 [16]. The result is a standard IPv6 UDP packet with standard headers (40 bytes IPv6 basic, 8 bytes UDP, and modified original packet as payload). Since the original IPv6 addresses are removed, a standard encapsulated packet is

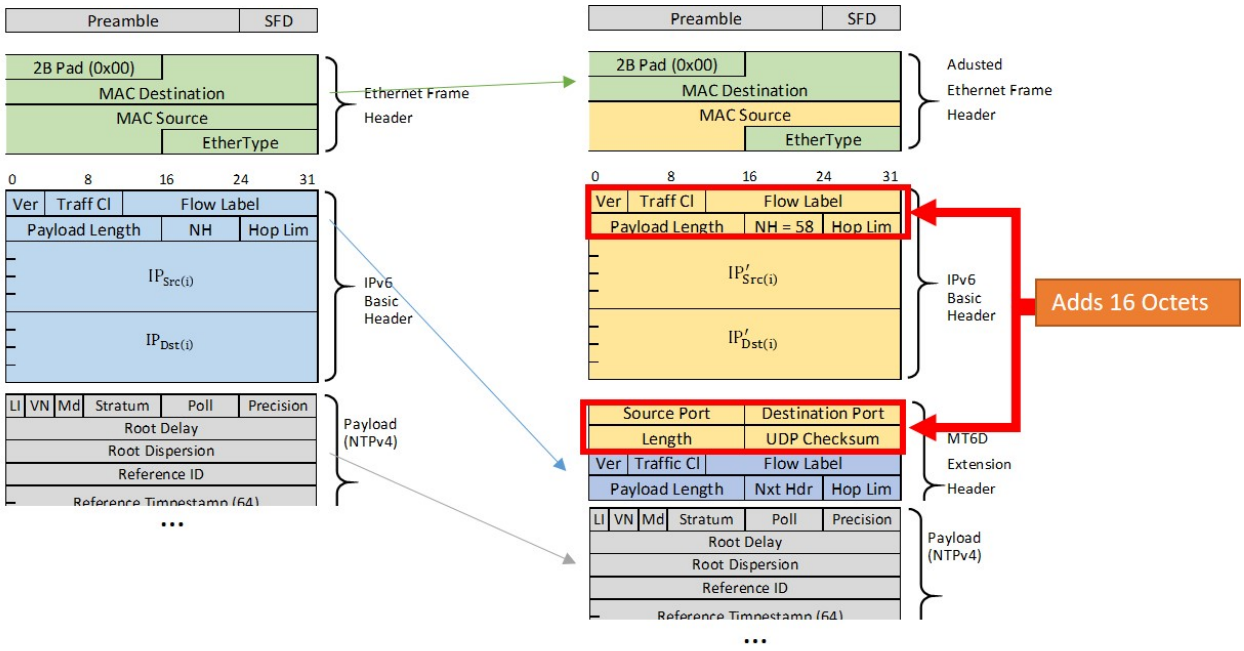
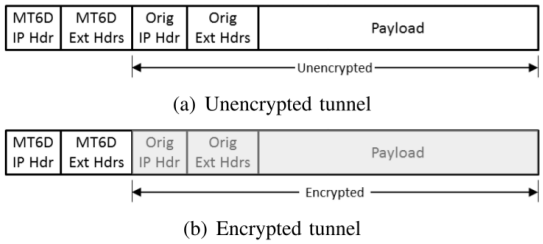


Figure 1.2: The MT6D tunneling process. An MT6D tunnel encapsulates the original packet with with a new IPv6 basic header. The Ethernet header is preserved, with to ability to optionally replace the Media Access Control (MAC) Source address. The original IPv6 basic header is preserved as a payload item, immediately after an inserted MT6D Extension Header (EH), which takes on the same format as a UDP EH. Original IPv6 addresses are then completely removed. The original packet payload then follows. Here, Network Time Protocol v4 (NTPv4) is the example upper layer payload [16].



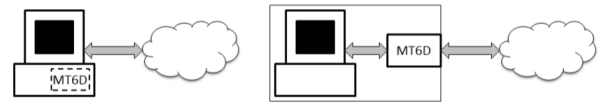
(a) Unencrypted tunnel

(b) Encrypted tunnel

Figure 1.3: Optional payload encryption feature. The original IPv6 packet becomes the tunneled payload. This payload may be encrypted to prevent deep packet inspection [16].

16 octets longer due to the insertion of the UDP header and preservation of the first 8 octets of the original packet [23].

MT6D can either be carried out on host as embedded software or hardware, or through an independent gateway hardware platform as seen in Fig. 1.4. The Intellectual Property (IP) presented in HE-MT6D targets the standard Media Independent Interface (MII) physical (PHY) sublayer and can be deployed as either embedded hardware as illustrated in Fig. 2.3 or an independent gateway device. For evaluation, HE-MT6D is implemented on the Terasic DE2-115 Cyclone IV development board as an independent gateway device.



(a) MT6D embedded on the host device in software (b) MT6D implemented as a stand-alone gateway device

Figure 1.4: MT6D deployment strategies. MT6D may be deployed as either an embedded or external solution [16].

1.4 Related Work

To better understand the purpose and direction of HE-MT6D, it is necessary to understand a bit about previous work in the field.

In 2012, Dunlop et al. pioneered MT6D using Python. Throughput was just under 16 Mbps, with approximately 2.1% to 2.6% packet loss experienced due to the Network Interface Card (NIC) becoming temporarily disabled during the address rebinding process [17].

In 2013, Hardman optimized an MT6D gateway implementation in C to utilize Linux kernel resources and more efficient libraries. These optimizations yield a gateway device that provides 156.7 Mbps straight pass-through traffic without MT6D engaged and 84.2 Mbps with MT6D engaged on a 1 Gbps network. Packet losses were much less at 0.0-0.9% depending upon the topology [23]. The hardware used in the implementation was the single core Marvell Kirkwood 88F6281 SoC ARM CPU, with 512 MB DDR2 800 MHz RAM and two gigabit Ethernet ports [24, 36]. Hardman showed that C- and kernel-based implementations are much more efficient, but near 1 Gbps throughput performance would require perhaps a server-class platform.

In 2015, Sherburne implemented MT6D in resource constrained IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) wireless sensor nodes to demonstrate viability in protecting IoT sensor nodes [45]. Results showed low memory (less than 3 KB) and insignificant current draw overhead for a client based system (smaller than the measured margin of error), but do point out that the symmetric key system employed poses a challenge for the scaling required for large IoT networks.

In 2015, Morrell et al. demonstrated a server-client method to provide initial configuration management and distribution. The method works by securely and anonymously obtaining initial configurations via blind rendezvous to a Redis key-value store simulating a BitTorrent Mainline Distributed Hash Table (DHT). The server-client method sought to removing the need for pre-shared static configurations and increasing mobility [38].

NSPs are application-specific processors that offload computationally intensive network packet

operations in hardware. They are often used in Internet Protocol Security (IPsec) and Secure Sockets Layer (SSL) protocol acceleration [7, 50, 22, 20, 34]. They have developed to be faster and more efficient over the years. HE-MT6D research designs an NSP for MT6D.

HE-MT6D research fulfills the next step towards design of an ASIC for MT6D. It pushes the state of the art by providing even higher MT6D throughput with less silicon resources. It also seeks to provide relief in key distribution, deployment, maintenance, and management challenges by surreptitiously providing MT6D services to any number of clients at a centralized location at the border of any network. It can work in tandem with a server-client system, as it only executes node configuration data.

1.5 Research Objectives

The primary goal of HE-MT6D is to move MT6D into hardware in such a manner that supports the ultimate design of an ASIC MT6D gateway device. The research for HE-MT6D presented here also takes up the challenge of doing so a line rate speeds. To support these goals, the following sub-goals for HE-MT6D design and performance are enumerated:

1. It must perform functionally. Developing and integrating kernel, Operating System (OS), driver, library, and user functions all in hardware is a challenging an non-trivial endeavor. Control flow among all the independent subsystems may be a challenge.

2. It must be unobtrusive. It should not interfere with the passage of non-protected

traffic. MT6D should be applied successfully to only qualified traffic.

3. It must be high performance. It should perform as close to line rate speeds of 1 Gbps as possible.

4. It must require low resources. It should use as few silicon components as necessary, which would aide in transitioning to an ASIC design that may be implemented on small devices. External components such as off-chip RAM should be avoided if possible.

5. It should support multiple nodes. As a gateway device, the developed system should be flexible enough to scale to support multiple endpoints.

6. It should be full RTL. This is a design constraint given from the hypothesis. Relaxation of this constraint is allowed during system initialization; a coprocessor can be used to write to initialization registers, but said coprocessor must not contribute to the functioning of HE-MT6D after initialization is complete.

1.6 Research Questions

This thesis asked the following four research questions to facilitate meeting the primary objective as well as accomplish supporting sub-goals:

1. HE-MT6D is a departure from the software versions (Python and C) and involves several layers of abstraction down to the bit level. What support systems are taken for

granted at the user space in Python and C implementations and need to be emulated or recreated in RTL? Elements no longer available in HE-MT6D include threads, kernel, library dependencies, firmware, hardware, etc.

2. Keeping in mind the ultimate goal of an ASIC, at what layer of abstraction within the Open Systems Interconnection (OSI) 7 Layer Model should hardware-based MT6D design be targeted? How do neighboring layers need to be implemented or managed?

3. Previous implementations relied on network libraries to handle the IPv6 network stack. What protocols are taken care of by the network stack? What packets need to be generated to initiate and maintain communications during MT6D operation?

4. The HE-MT6D implementation is in essence an NSP. How can an architecture be fully designed using RTL logic? What challenges might be faced during the design process? How can the design be optimized to reasonably maximize data and packet throughput as well as allow for a scalable number of client nodes?

1.7 Methodology and Organization of Thesis

The rest of this document seeks to answer these four research questions in order support the hypothesis presented in Section 1.1. The first three research questions are answered in two feasibility studies. The last research question is answered in the final chapters of Design, Implementation, and Evaluation of HE-MT6D. The hypothesis is assessed in the chapter

Conclusion.

Feasibility Study 1 looks to answer Research Questions 1 and 2. It studies porting MT6D onto an embedded platform and works to identify all the layers of abstraction provided by the Linux kernel, dependent libraries, user space programming, and general purpose commercial off the shelf hardware and firmware. This study unearths what architectural elements and subsystems are necessary to build the framework that the RTL version of MT6D will execute on top of.

Feasibility Study 2 looks to answer Research Question 3. It studies the IPv6 communications stack and works to understand the communications medium that MT6D is meant to intercept and manipulate: IPv6 packets. Supporting packets, this study also dives below the user space requires understanding and manipulating ICMPv6 messages, to allow for communications setup and tear down, as well as accommodating for specific control messages that may contain sensitive information. This study ascertains which IPv6 address sets to extract, what parameter fields to extract, how to handle them, how to manipulate packets while maintaining protocol integrity, and how to identify each packet case type in order to determine how to handle each type as it is received.

From here, the thesis delves into the chapters of Design, Implementation, and Evaluation. These chapters answer Research Question 4 through practical design and implementation of HE-MT6D and evaluation of its performance. Chapter 4 Design reviews the overall processes that must take place to maintain global time synchronization for all HE-MT6D platforms when deployed, the process of system initialization, the process of system maintenance, and

the process of datapath processing. Chapter 5 Implementation goes over the HE-MT6D subsystems,the modules deployed into each of the subsystems, and system integration. Feedback from the two feasibility studies is used to develop the final system architecture. Chapter 6 Evaluation evaluates system performance according to the established design objectives.

Chapter 2

Feasibility Study 1: Embedded MT6D

This chapter sets out to answer the first two research questions with a feasibility study:

1. HE-MT6D is a departure from the software versions (Python and C) and involves several layers of abstraction down to the bit level. What support systems are taken for granted at the user space in Python and C implementations and need to be emulated or recreated in RTL? Elements no longer available in HE-MT6D include threads, kernel, library dependencies, firmware, hardware, etc.

2. Keeping in mind the ultimate goal of an ASIC, at what layer of abstraction within the OSI 7 Layer Model should hardware-based MT6D design be targeted? How do neighboring layers need to be implemented or managed?

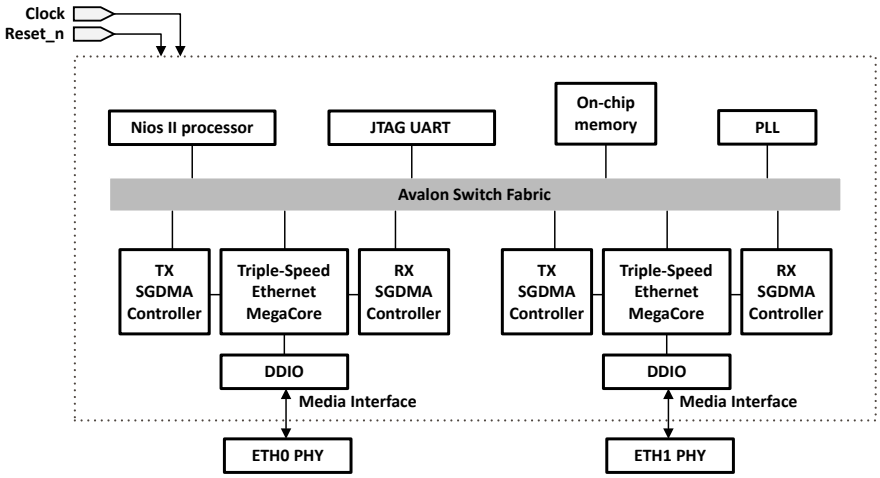
Feasibility Study 1 focuses on Research Question 1 and endeavors to recreate C-based MT6D on an embedded platform without Linux kernel support. Section 2.1 summarizes the design,

build, and evaluation of an embedded form of MT6D. The work presented does not fully employ end-to-end MT6D but emulates enough of the protocol to discover what systems and dependencies are needed for MT6D to operate at the RTL. In the course of conducting this feasibility study, Section 2.2, which addresses at what layer of abstraction to target design for HE-MT6D, naturally arose and answers Research Question 2. Section 2.3 reflects on the lessons learned during the execution of Embedded MT6D on how to driver the lower MAC and PHY layers and meet the established research goals.

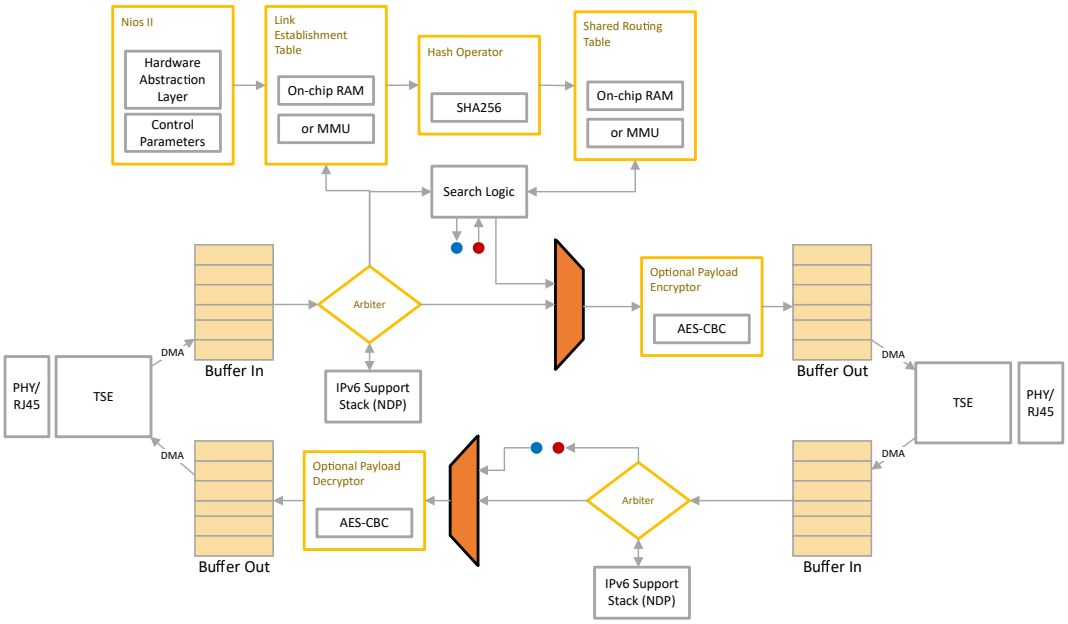
2.1 Embedded MT6D

The work in this chapter seeks to design, build, and evaluate an embedded version of MT6D on an FPGA as a stepping stone to fully understand the mechanisms necessary to produce an RTL design. An FPGA is an integrated circuit that can be configured after manufacturing using a Hardware Description Language (HDL). Verilog and SystemVerilog are the HDL languages used.

The processor chosen was the Nios II soft processor, a RISC processor that can be instantiated on an FPGA [6]. Efficiency and speed was not the goal, but rather, learning subsystem dependencies. Direct programming of the Nios II Hardware Abstraction Layer HAL was done to drive hardware and move and manipulate data. This process reveals first hand the software and hardware dependencies deployed in previous implementations. The SHA256 implementation was provided by Brad Conte [11].



(a) Embedded MT6D physical architecture.



(b) Embedded MT6D logical architecture.

Figure 2.1: Embedded MT6D design architecture. In order to understand the software implementation of MT6D, an embedded version was first created. (a) The system consisted of a Nios II processor, on-chip memory, four Scatter-Gather Direct Memory Access (SG-DMA)s, two triple speed Ethernet cores, two Double Data Rate I/O (DDIO), and a Phase-Locked Loop (PLL) signaling clock. All components were connected on a common memory-mapped bus. (b) The resulting architecture was buffer-based and extremely slow due to the memory transaction, control, and library overhead all on a common memory-mapped bus.

The system designed is represented in Fig. 2.1a. Fig. 2.1a shows a common data bus with multiple peripherals attached in a Master-Slave relationship. The Nios II soft processor is positioned as the master, with Joint Test Action Group (JTAG) Universal Asynchronous Receiver/Transmitter (UART), Phase-Locked Loop (PLL), On-Chip memory, and an Ethernet MAC subsystem attached as slave peripherals. The Ethernet MAC subsystem includes two complete sets of the following: Altera's Triple Speed Ethernet (TSE) megacore IP, transmit and receive Scatter-Gather Direct Memory Access (SG-DMA) controller, Double Data Rate I/O (DDIO), and external Ethernet PHY. Fig. 2.1b shows a logical representation of the implementation, with packets being buffered into main memory, then processed in user space.

Feasibility Study 1 did not complete full MT6D encapsulation and decapsulation, but it did maintain a SRT, pass packets, and replace addresses with their hashed values. Nonetheless, embedded MT6D served its purpose in making apparent the subsystems necessary to bridge software-based MT6D into hardware, especially since software MT6D relied heavily on Linux kernel functions and commercial off the shelf computing equipment. Table 2.1 outlines these subsystems. Of note, the execution was rather slow. The network stack was nonexistent and was ported from the Linux kernel. Many actions contended for processor resources: calling functions every second to check if rotations needed to be recalculated, moving data in and out of memory by managing SG-DMA descriptors, maintaining the data structures of each protected node, forming IPv6 packet, calculating SHA256 hashes, and maintaining the SRT data structure. The threads used in [24] would have had to be built to support multiple

processes, and process management itself would have to be built. On top of only having a clock rate of 100 MHz, it was quickly apparent that close integration of parallel hardware modules that account for all these subsystems would be required. Further investigation of the characteristics of IEEE 802.3 Ethernet would be necessary. Analysis is provided in Section 2.2.

Table 2.1: Enumeration of the functional dependencies required to run MT6D on top of the Linux kernel. Embedded execution relied on the Nios II Hardware Abstraction Layer (HAL). Hardware will need to address all these functions in Register Transfer Level (RTL).

Element	OS-based Execution	Embedded Execution
Timers, Time Management	Kernel Library	HAL
Process Scheduling	Kernel Library	User Space
Event management	Kernel Library, threads	HAL
Memory Management	Kernel Library, data structures	HAL, SG-DMA Drivers
Event Calls	`libev`	HAL
Hash Function	`libsodium`	User Space
MT6D Integration	User Space	User Space
Network Stack	`libnet1`, `libmnl`, `netfilter_queue` socket programming	HAL, TSE

Console output is shown in Listing 2.1 for the hash computations and rotation management of one node. Compute times are shown to be long and unpredictable. Here, a new hash is calculated from as little as 0.086 seconds to as long as 0.24 seconds.

```
Opened scatter-gather device '/dev/sgdma_tx_h'
Opened scatter-gather device '/dev/sgdma_tx_n'
Opened scatter-gather device '/dev/sgdma_rx_h'
Opened scatter-gather device '/dev/sgdma_rx_n'
LET[0] Key: |1011121314151617|18191a1b1c1d1e1f
Rotation interval est to: 2
LET[0] set to active=1
LET[0] Source Base Address: |0000111122223333|4444555566667777
```

```
LET[0] Dest Base Address: |88889999aaaabbbb|ccccddddeeeeffff
Overhead of timestamp timer is 164 cycles. Which is also 0.000002 sec

Tick tps is at 1
Current time is 1
READY
Waiting for packet

Rotation #0
Source hash: 0x|a3d66f8f42bd7eb2|94b9ca6507d8c4bb|a5d4338c1a4e1566|
    cc045977ee53ab6c
Destin hash: 0x|70aafcadaea62bdc|4ffffda92cf5591c|a1f544f41b805295|51370
    b9aa231c888
Cycle time is 24173011. Which is also 0.241730 sec

Rotation #1
Source hash: 0x|60574ce9fd3c4941|58fa520461a7f2f0|5fe8a232ec8facf5|3
    d88c9fbeb213f56
Destin hash: 0x|cb675f72f83086f9|ab230bfa807a0650|ac92143d0c5108b9|150
    a427a1d970438
Cycle time is 8642193. Which is also 0.086422 sec

Rotation #2
Source hash: 0x|77cd3eafafb3b4d8|4db8e4735fa7b0f3|6fc88b16a9058443|8
    bc339885d8da42b
Destin hash: 0x|5bfc739587c8e2db|22f2d59e9146acc3|8d35f181805c1a03|
    b894ebc8fd36102d
Cycle time is 13054314. Which is also 0.130543 sec
```

Listing 2.1: Embedded MT6D initialization and rotation maintenance.

2.2 Lower Layer Integration: Physical and Data Link Layers

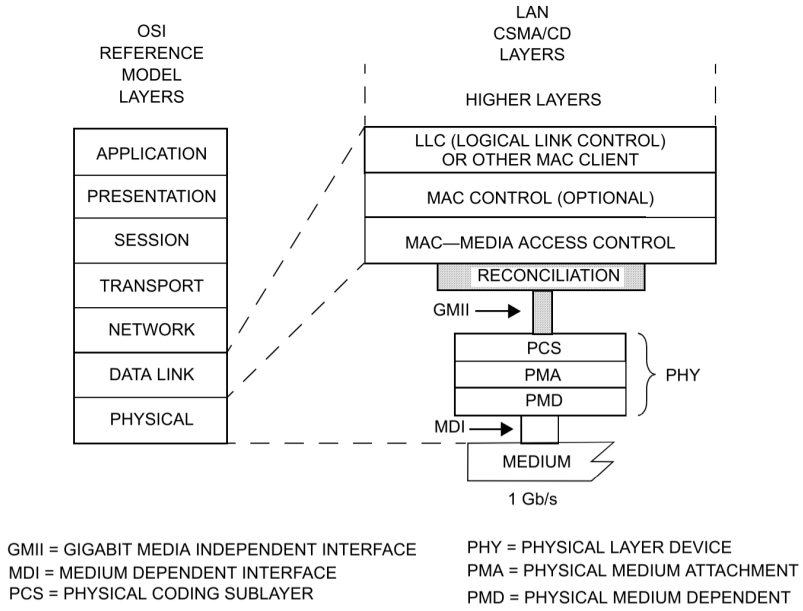
Section 2.2 discusses where in the communications stack to insert hardware based MT6D, describes performance characterization of IEEE 802.3 Ethernet, and discusses some design decisions for the lower layers below IPv6 communications.

2.2.1 Placement of Hardware-based MT6D

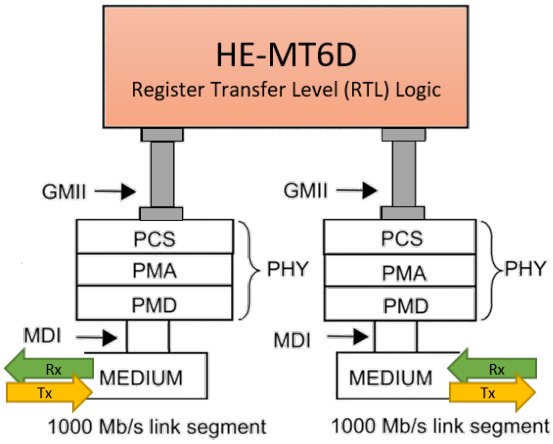
It is a challenge to decide at what layer of abstraction in the network stack HE-MT6D should be placed in order to provide maximum effectiveness, simplicity, and industry adoption. Placing HE-MT6D too close to the programming user space entangles it with software and library dependencies, which must be avoided as it does not support the research hypothesis. Going too low (such as managing the electrical charges a the wire level is both unnecessary and non-scalable due to the wide number of physical mediums available (unshielded twisted pair (UTP), coaxial cable, fiber optics, etc.) The ideal is to insert HE-MT6D at a universal protocol between the physical and logical interfaces.

The IEEE 802.3 working group produces the IEEE 802.3 standard for Ethernet, which defines international standards for 1 Mbps to 100 Gbps physical layer transmission[3]. The IEEE 802.3 specification operates with the Physical and Data Link layers of the seven layer OSI model. The Data Link Layer (DLL) provides logical tie in to the physical medium by providing MAC. The MAC protocol within the DLL provides CSMA/CD to arbitrate access to the PHY [3].

IEEE 802.3 Ethernet MAC is usually implemented in either software or hardware, and CSMA/CD and PHY implemented in hardware. After MAC, there exists the Reconciliation Sublayer (RS) seen in Fig. 2.2a that provides interface before data departure into any physical medium dependent device which would in turn transceive signals over a physical medium.



(a)



(b)

Figure 2.2: Placement of Network Security Processor and Hardware Engine for MT6D (HE-MT6D) in the Reconciliation Sublayer (RS) between the Data Link and Physical layers of the IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Local Area Network (LAN) model. (a) This diagram shows the Gigabit Media-Independent Interface (GMII) relationship to the Open Systems Interconnection (OSI) reference model and the IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Local Area Network (LAN) model [3]. (b) HE-MT6D is designed to interface with the Media Independent Interface interface and bilaterally replaces the communications stack above the Reconciliation Sublayer (RS), similar to a baseband repeater.

The RS provides the most logical demarcation point for which to insert any silicon that may manipulate traffic in an arbitrary manner, as it incorporates a Media Independent Interface (MII). The MII has several variants, to include Reduced Media-Independent Interface (RMII), Gigabit Media-Independent Interface (GMII), Reduced Gigabit Media-Independent Interface (RGMII), 10-Gigabit Media-Independent Interface (XGMII) and Serial Gigabit Media-Independent Interface (SGMII) [3]. These variants target different speeds applications for IEEE 802.3 Ethernet. They provide brokering between upper-layer logic and lower level PHY. The three sublayers below the RS—the Physical Coding Sublayer (PCS), Physical Medium Attachment (PMA), and Physical Medium Dependent—comprise the formal PHY sublayer and generally depend upon the attached physical media whether they be coaxial, twisted pair, or fiber optic cables [3]. Examples of such PMDs are seen in common Gigabit Interface Converter (GBIC) or Small Form-factor Pluggables (SFP) that are used fiber optic switches. An example for future ASIC deployment would be a man-in-the-middle deployment withing the RS as seen in Fig. 2.3. The design presented in HE-MT6D is geared toward intercepting and modifying communications through the RS. The hierarchical structure of HE-MT6D deployment as it relates to the OSI reference model is depicted in Fig. 2.2b and is similar to baseband repeaters.

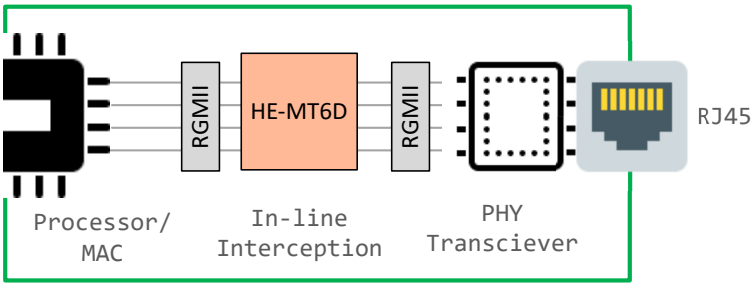


Figure 2.3: Illustration of HE-MT6D implemented in-line with the Reduced Gigabit Media-Independent Interface (RGMII). As designed, the targeted location of HE-MT6D intercepts, decodes, then resends information on the RGMII of the Reconciliation Sublayer (RS). The RS is located after data would generally leave a processor and before it is placed onto the physical medium transceiver.

An important note for future work is that although HE-MT6D is designed to intercept packets at the RS, the IP is modular and can be packaged as dedicated hardware sub-engine within a larger ASIC design. The modularity of HE-MT6D is later seen in Chapter 5.

2.2.2 Understanding and Maximizing Throughput

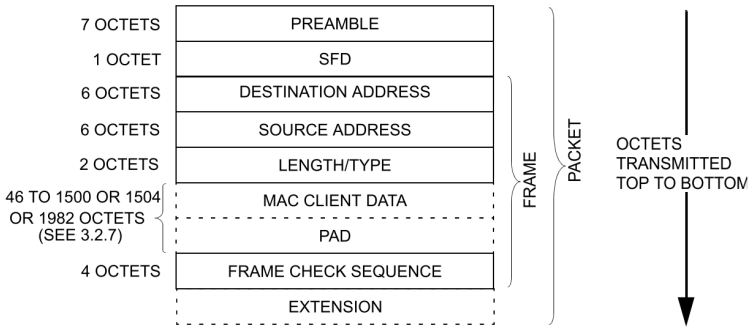


Figure 2.4: Size of and fields within a basic Ethernet Frame. [3].

The standard Ethernet frame is shown in Fig. 2.4. It begins with the Preamble, which is an alternating stream of 1's and 0's (1010101...101010), followed by a Start of Frame Delimiter (SFD) (10101011). Afterwards follows a set source and destination MAC addresses, the

EtherType, the client payload itself, then a frame check sequence and any optional extensions. Following every frame is a 9.6 ms inter-frame transmission gap that temporally occupies 12 octets. In total a transmitted frame will comprise between 7+1+6+6+2+46+4+12=84 octets and 7+1+6+6+2+1500+4+12 = 1538 octets [3].

IEEE 802.3 1 Gbps Ethernet clocks at 125 MHz, with 8 bits per symbol [3]. As such, a standard packet with sizes 84 to 1538 octets requires a period between 672 ns and 12,000 ns to transmit. For packet throughput, smaller packets can be produced at a rate of $\frac{10^9}{84*8} \approx 1,488,095$ packets per second (pps), and larger at $\frac{10^9}{1538*8} \approx 81,274$ pps, as shown in Table 2.2.

Table 2.2: Basic Ethernet Frame Measurements

	Octets	Transmission Time	PPS
Smallest	84	672 ns	1,488,095 pps
Largest	1538	12,000 ns	81,274 pps

In conclusion, to prevent packet loss at full line saturation, the HE-MT6D processor must process a packet in less than 672 ns. The Cyclone IV FPGA used for development of HE-MT6D runs at 100 MHz, which means that packets must be processed and delivered in 67 cycles to prevent packet loss under maximum strain. In the real world, stochastic processes govern packet arrival rates and sizes, which greatly relaxes this condition, but the stricter 67 cycles will be retained as a measurement of success in this thesis.

2.2.3 Adjustments

HE-MT6D is designed to surreptitiously observe traffic and inject changes to packets only when necessary. It also expands packets that may exceed standard Maximum Transmission Unit (MTU) sizes. To support both surreptitious operation and packet expansion, several options must be set at both the DLL and PHY layers.

Promiscuous Operation Normally, a Layer 2 devices (such as a network switch) will crosschecks destination MAC addresses and filter out inbound frames that do not match its own MAC address. Additionally, when Layer 3 devices (such as routers and host systems) send a frame out any Ethernet port, they tend place their outbound MAC address into the source MAC address field.

However, to accept and analyze all traffic, the MAC function must be place into promiscuous mode. Transmission tagging should also be disabled, so that the original source MAC address is unaltered and retains all the original MAC information. Retaining the original MAC information helps to identify the node sending traffic during this research. Transmission tagging can enabled to protect host MAC address if need be for whatever administrative reason and is done by manipulating MAC function control registers.

Unsupported Frames Not all types of Ethernet frames and IP packets are supported. More specifically, 801.Q Virtual LAN (VLAN) tagging and jumbograms are not supported.

PHY Control Settings The Marvell 88EE1111 Ethernet transceiver chip used in the development platform limits frame transmissions to 1518 octets (not counting the preamble, start of frame delimiter, and frame check sequence) by default. To accommodate for expanded frame sizes, the PHY transceiver must be set to accept 16 more octets on its payload, otherwise the transceiver will drop MT6D frames as they pass through.

The Ethernet transmission speed must be set to 1000 Mbps with auto-negotiation enabled. Since there are no drivers to make these changes, they must be done manually.

2.3 Conclusion

Feasibility Study 1, unearths what subsystems are used in MT6D as a user space program in a Linux kernel and general purpose computing environment, and what may be required to enable MT6D in full RTL logic. It also exposes where to design HE-MT6D within the IPv6 communications stack—between the DLL and PHY sublayers in the Reconciliation Sublayer (RS), using the Media Independent Interface (MII) protocol. This placement supports future ASIC development and integration. Feasibility Study 1 also determines the specifications of IEEE 802.3 that allow HE-MT6D to reach full line rate speed.

The next chapter is Feasibility Study 2, which looks into the IPv6 communications stack and seeks to answer Research Question 3.

Chapter 3

Feasibility Study 2: IPV6 Communications

This chapter sets out to answer the next research question with a feasibility study:

3. Previous implementations relied on network libraries to handle the IPv6 network stack. What protocols are taken care of by the network stack? What packets need to be generated to initiate and maintain communications?

Feasibility Study 2 looks at the IPv6 communications protocol and the required packet exchanges necessary to establish and maintain IPv6 Communications. Section 3.1 reviews IPv6 basic concepts, the addressing model, and packet structures. Section 3.2 is critical to building the preprocessing and packet assembly stages of HE-MT6D at the network level. This section describes the 13 packet case types that have been identified for MT6D to work

and how they each need to be handled. This is usually not necessary in kernel-based designs as a communications library is usually available. Since HE-MT6D does not have one, it must bind directly with the IPv6 protocol.

3.1 Internet Protocol version 6 (IPv6)

The need for IPv6 addresses stems from the limited availability of IPv4 address space. In the evolution from IPv4 to IPv6, some features are maintained, while most have been improved upon. There are three types of addresses in IPv6: unicast, anycast, and multicast. A unicast address identifies a single interface. An anycast address identifies a set of interfaces that usually belonging to different nodes; a packet addressed to an anycast address is sent to the nearest interface identified by that address [28]. A multicast address identifies a set of interfaces also usually belonging to different nodes; a packet addressed to a multicast address, is sent to all interfaces identified by that address instead of the nearest as determined by the routing protocol [28].

For HE-MT6D, both unicast and multicast addresses are of particular interest. Multicast addresses are extremely important in beginning any communication session, as they are used by nodes to discover unknown parameters in a process called the Neighbor Discovery Protocol (NDP). During NDP, a node sends out special solicitation messages to different multicast address groups to discover if there are any routers on the network, what subnet to use during IPv6 address generation, what Domain Name Server (DNS) to use, and what a neighbor's

MAC address is so an Ethernet frame can be properly addressed. Unicast addresses are used generally to then continue point-to-point conversations.

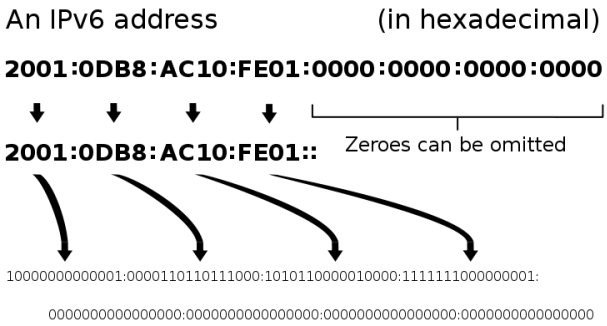
Through multicast addresses, a node can query for a class of services without knowing the exact address to begin with, since all similar-class servers will "join" or listen to any packet sent to their multicast address group. To name a few, `ff02::1` is the multicast address used to reach all nodes on the local segment, `ff02::2` for all routers on the local segment, `ff02::1:2` for all DHCP, `ff0x::fb` for Multicast DNS, and `ff0x::101` for NTPv4.

3.1.1 Addressing model

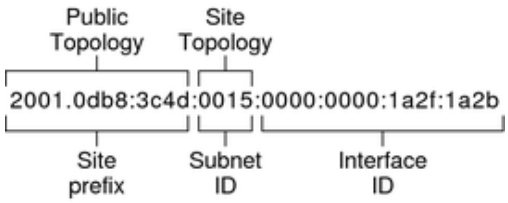
IPv6 addresses are assigned to interfaces rather than to hosts themselves [28]. In that case, a host can be identified by any of its interfaces or a subset thereof. All interfaces must have at least one directly addressable unicast address, but may also have additional unicast, anycast, or multicast addresses [28]. In the case of load sharing, it is possible to have multiple physical interfaces to be presented to and addressed by the network layer as a single address. Subnets are still used in IPv6, whereby a subnet prefix refers to a link; however, with IPv6 multiple subnet prefixes can be assigned to the same link [28].

IPv6 addresses themselves are 128-bits in length, with the IPv6 that identifies a particular interface. IIDs are the last n bits on a 128-bit IPv6 address. The first $128 - n$ bits are the subnet prefix and can be used to identify an aggregation of addresses in a similar way CIDR was used for IPv4. The interface identifier can be the same for a particular node across

multiple interfaces as long as the subnet link address is different. Subnet prefix is used for routing into regional, Internet Service Provider (ISP), site, and local network subnets. IPv6 address formating is seen in Fig. 3.1 with an example using 64-bit subnets and 64-bit IIDs.



(a)



(b)

Figure 3.1: IPv6 address format. (a) IPv6 addresses are 128-bit addresses represented in hexadecimal notation [1]. (b) This figure shows an example configuration of an IPv6 address. This one in particular is the globally routable unicast address [43]. The first 48 bits is the Global Routing Prefix or site prefix, the next 16 bits is the subnet identifier, and the last 48 bits is the unique Interface Identifier (IID).

Unlike IPv4, the generous number of IPv6 addresses available allows nodes to self-declare their own IPv6 address with low chance of conflict with an existing node. This process is known as Stateless Address Auto-Configuration (SLAAC). All the messages exchanged during SLAAC belong to the ICMPv6 protocol [48]. To execute SLAAC, a node will solicit the router multicast group for any router using an Router Solicitation (RS) message, with the purpose of discovering the local subnet prefix. A router would respond with a Router

Advertisement (RA) containing the local subnet prefix along with other key information. Alternatively, a node can simply listen for periodic unsolicited RAs. After learning what subnet prefix to use, the node will then choose its own IID. Before permanently bonding to its newly autogenerated address, the node will solicit the network for any other interface that might be using the intended address by sending a Neighbor Solicitation (NS) Message. If it receives no reply within a short timeout period, the node assumes ownership of the address.

3.1.2 Resolving MAC Addresses

IPv6 address are logical and can span any geographical location across the globe and are used to route IPv6 packets. However, IPv6 packets are transported inside of Ethernet frames, which are created between each physical link. So, to move IPv6 packets between two physical devices, such as a host and its network gateway router, MAC addresses are used. This is why a MAC address is also known as the "physical" address of a device.

NDP is used to resolve the next-hop MAC addresses of neighboring devices and replaces Address Resolution Protocol (ARP) from IPv4 [39]. The protocol works by sending a NS packet to the solicited-node multicast address of the intended recipient. This special multicast address is defined as `ff02::1:ffXX:XXXX`, where `XX:XXXX` represents the last 24 bits of the solicited node [39, 27]. All IPv6 devices interfaces subscribe to and listen to a solicit-node multicast address that is derived from their IPv6 address. If a node listening to solicited-node multicast address group sees the NS, it will respond with a Neighbor Advertisement

(NA), which among other things, contains its MAC address, which is used for Ethernet frame delivery.

3.1.3 IPv6 Packet Structure

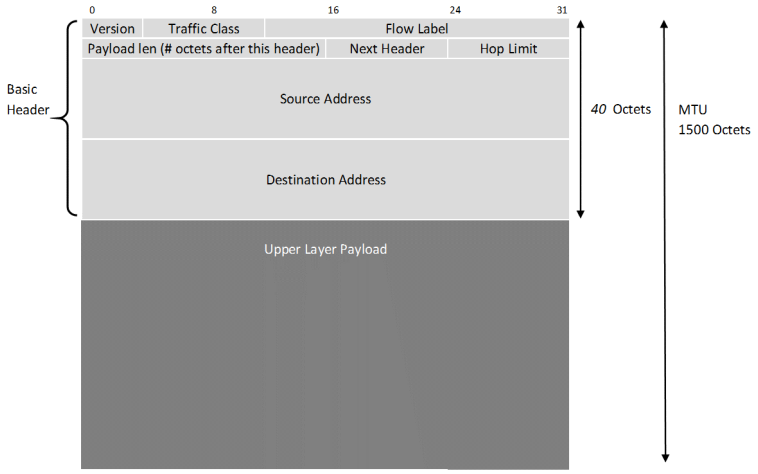


Figure 3.2: Standard IPv6 packet format. All IPv6 packets begin with the IPv6 basic header, which is then followed by upper layer payload data.

IPv6 packets begin with the IPv6 basic header, followed by the upper layer payload, as depicted in Fig. 3.2. Relative to this figure, bits are transmitted left to right, starting at the top row, down to the bottom. The IPv6 basic header comprises 40 octets, which includes the version number (4 bits), traffic class (4 bits), flow label (20 bits), next header selector (8 bits), hop limit (8 bits), source address (128 bits), and destination address (128 bits). The upper layer payload follows, containing any extension headers (such as UDP, Transmission Control Protocol (TCP), and ICMPv6 among numerous others) as well as any data. Altogether, the IPv6 basic header and upper layer payload are limited in size by the MTU length. RFC 2460 [13] recommends transmission links accommodate MTU sizes of 1500 octets or greater

(greater in order to allow for tunneling protocols, such as MT6D or IPsec). IPv6 packets are encapsulated within an Ethernet frame and are the considered the MAC Client Data, as presented in Fig. 2.4.

HE-MT6D expands packets by 16 octets, so setting the transmission network architecture to support a larger MTU link size of 1516 is recommend. However, HE-MT6D also allows for the use of ICMPv6 "Packet Too Big" Type 2 error feedback to let clients know to reduce their MTU size so that their packets fit within the 1500 octet limit after including the additional encapsulation overhead.

3.1.4 User Datagram Protocol (UDP)

The upper layer payload section may contain any combination of upper layer protocols and their respective data payloads. These extra headers are called Extension Header (EH). Of particular interested are UDP and ICMPv6.

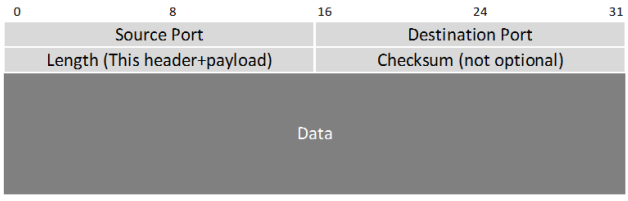


Figure 3.3: The User Datagram Protocol (UDP) Extension Header (EH) format. When in use, the UDP EH comes after the IPv6 basic header.

Fig. 3.3 depicts the UDP EH format. The UDP EH includes source port (2 octets), destination port (2 octets), UDP packet length (2 octets), and checksum (2 octets); for a total of 8 octets. RFC 2460 [13] requires all clients to reject and log IPv6 packets that do not contain

a correctly computed UDP checksum, but RFC 6935 [18] allows the relaxing of the checksum requirement for packets tunneled over UDP. As such, HE-MT6D does have a checksum validation mechanism deployed, but it is not enabled for simplicity.

MT6D adopts the UDP header as a subset of it's encapsulation process as described earlier in Fig. 1.2 and is used to form the MT6D EH.

3.1.5 Internet Control Message Protocol for IPv6 (ICMPv6) Type 1-4 Errors

All IPv6 packets contain the basic header source and destination addresses, which are referred to as a_0 and a_1. However, sometimes two or more additional IPv6 addresses are located within the upper layer payload that must also be handled properly. ICMPv6 Type 1-4 Errors are especially challenging, as they contain copies of their invoking packet as payload, which must be examined and decapsulated or translated.

Since they contain a copy of the offending packet that invoked the error, a typical ICMPv6 Type 1-4 Error Packet has generally four IPv6 addresses that require examination. These are denoted as a_0, a_1, a_2, and a_3 in Fig. 3.4. Generally, a_1=a_2 since the packet is usually destined for the offending source, which means only three address lookup requests are generally required per inbound packet (a_0, a_1, and a_3) to aide in determining packet case type. These request are known in HE-MT6D as Translation Lookup Request (TLRQ)s. Packet case types are addressed in Section 3.2.

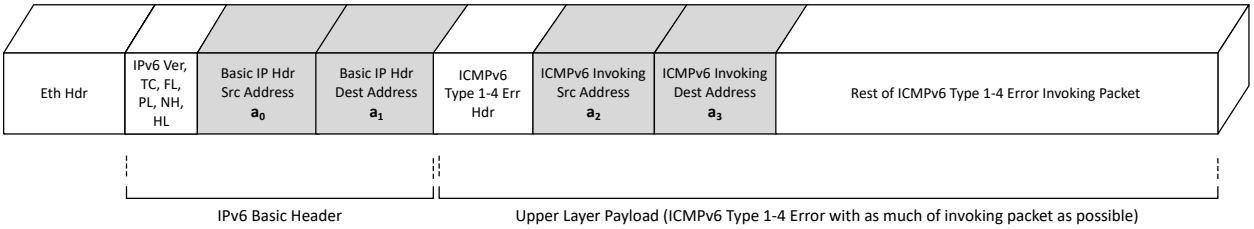


Figure 3.4: A typical Internet Control Message Protocol for IPv6 (ICMPv6) Type 1-4 Error Packet. Typical ICMPv6 Type 1-4 Error packets generally have four IPv6 addresses denoted as a_0, a_1, a_2, and a_3. These addresses must all be examined.

In order to provide transparent operation, properly handling ICMPv6 Type 1-4 Error messages is necessary. These nontrivial errors notify a client that the destination is somehow unreachable (Type 1), the packet is too big (Type 2), the hop limit has been exceeded and packet lost (Type 3), or there is an error in one of the basic header fields that needs to be corrected (Type 4) [10]. The ICMPv6 Type 1-4 Error EH fields are shown in in Fig. 3.5. An IPv6 Basic Header with the next header value of 58 (`0x3A`) would precede the ICMPv6 EH. The ICMPv6 EH itself contains the type code (8 bits), option code (8 bits), checksum (16 bits), a reserved slot which is used for MTU adjustment for the Packet Too Big Type 2 Error (32 bits), and then as much of the invoking packet as possible.

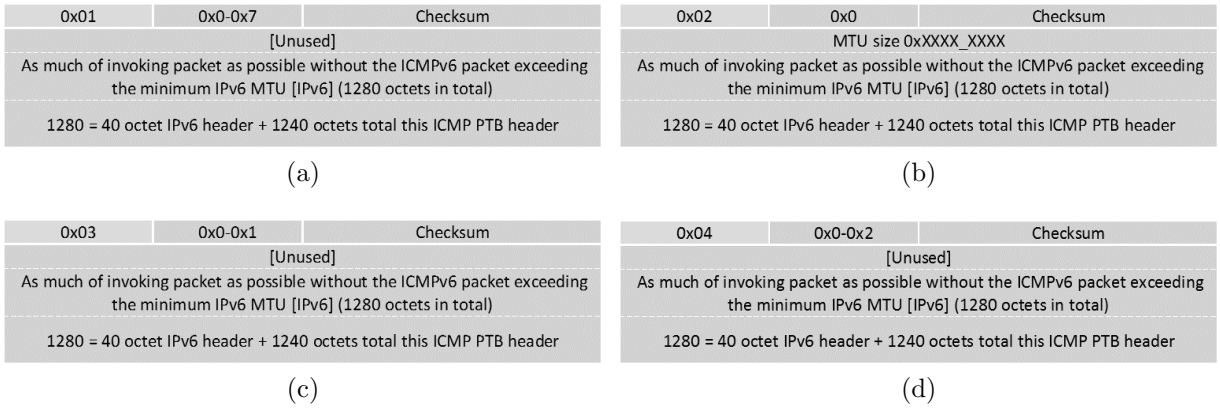


Figure 3.5: ICMPv6 Error types. There are four ICMPv6 Error types that must be handled carefully: (a) Type 1 Error: Destination Unreachable. (b) Type 2 Error: Packet too big, adjust Maximum Transmission Unit (MTU) size. (c) Type 3 Error: Time exceeded (d) Type 4 Error: Parameter problem. The second 32-bit word is used to adjust MTU size for Type 2 Errors.

3.2 Packet Case Types

Each HE-MT6D core has preprocessing components, namely the Field Extractor, memory search structures and transaction brokers, and a reconciliation stage within the Packet Assembler. These components are further discussed later in Section 5.6. The entire goal of these components is to analyze certain relative field patterns of each packet that passes through, then extract metadata and all IP address sets in order to positively identify the structure of the packet, which fields to replace, and whether either the entire packet or ICMPv6 invoking payload (or both) need to be encapsulated, decapsulated, or simply translated. Determining packet case types is one of the primary goals of packet preprocessing. It is only after a packet has been classified will the Packet Assembler know what actions to take upon each individual packet.

3.2.1 Unmodified Communication Across a Router

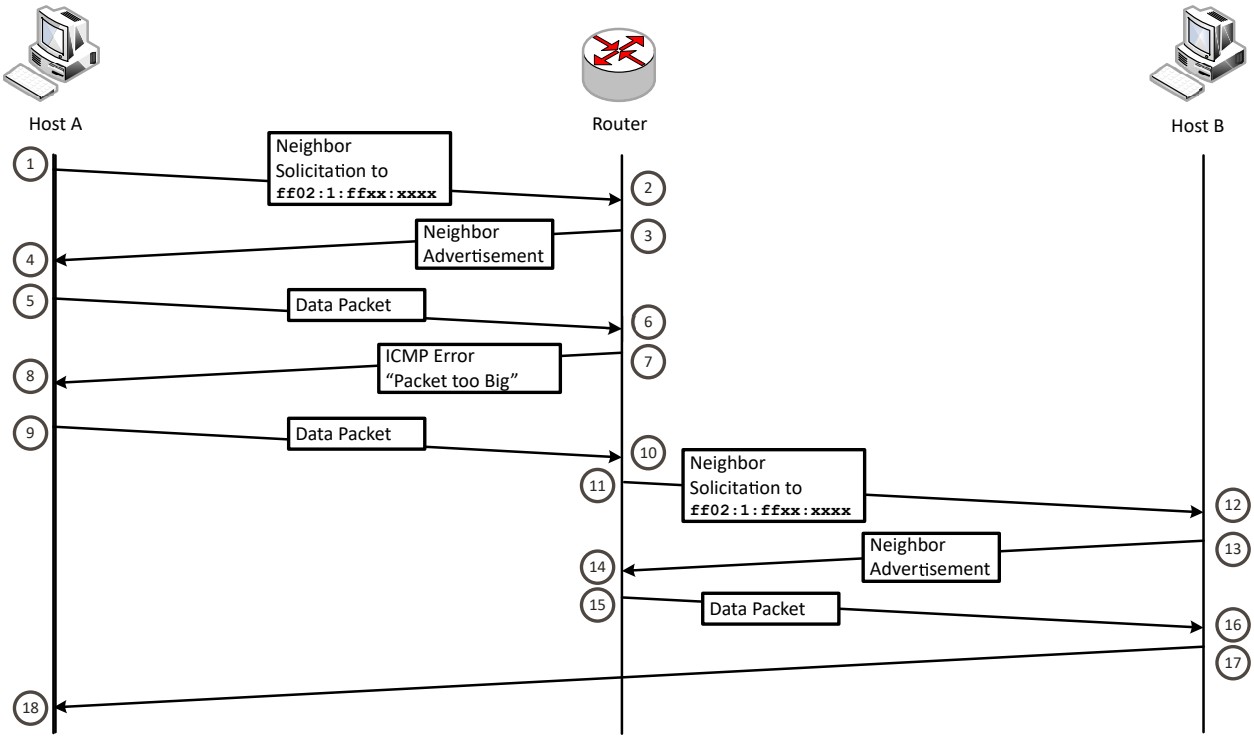


Figure 3.6: Packet exchanges used to start and continue communication over IPv6.

HE-MT6D does not rely on an third party IPv6 network communications stack to handle IPv6 packets, but rather directly manipulates portions of the IPv6 communications protocol. As such, it is requisite to examine the IPv6 protocol to see what packets are used to initiate and maintain communications. Fig. 3.6 shows an example IPv6 communications session and illustrates the different packet types that must be handled. For this example, the two nodes that are attached to a single router on two different interfaces, are statically addressed, and have not yet communicated. The example involves the NDP process, ICMPv6 error handling and adjustment, and data transmission over TCP. The example in Fig. 3.6 is described below:

Host A wants to communicate with Host B. In order to determine the next-hop physical device to send its data payload to, Host A must first determine the MAC address of any intermediate neighboring device that might be able to reach Host B. To do so, Node 1 sends an ICMPv6 NS packet to the solicited node multicast group address that Host B would be a member of (1). The router receives the NS (2). It knows that it can reach Host B, so it replies with an NA response (3), which Host A receives (4). Host A now knows the next-hop physical device to send the data packet for Host B to and sends the data packet (5). However, Host A in this example was configured with the wrong MTU size, and the sent packet was too large. The router responds with an ICMPv6 Type 2 Error message "Packet Too Big" in order to instruct Host A to reduce its MTU size (7). Host A receives the ICMPv6 Type 2 Error message (8) and complies by adjusting its MTU size and resends the data packet (9). The Router receives and accepts the data packet (10). It sees that it must forward the packet to Host B but does not know Host B's physical address. It sends an NS message to the solicited-node multicast address and waits for a response (11). Host B sees the solicitation message (12). Since it is a member of the multicast group, it responds with its information using an NA message (13). The router receives the NA message (14) and forwards the packet it had received from Host A onto Host B (15). Host B receives the packet (16). It then decides to respond and sends a packet back to Host A (17), which Host A eventually receives (18).

3.2.2 Selective Modification of Multiple IPv6 Communication Protocols

Normally, one may think that HE-MT6D would simply need to provide encapsulation and decapsulation services. However, that is not the case, and HE-MT6D must provide encapsulation, decapsulation, translation, partial translation, as well as nested variants of these same functions. It must process NDP, ICMPv6 error control messages, and data payloads over TCP or UDP in different manners. First, NS messages are sent to solicited-node multicast addresses. These multicast addresses do not contain the full address of any particular host; rather, they contain just the last 24 bits of the solicited node. HE-MT6D must qualify packets and partially translate the multicast addresses of packets that meet certain exact conditions. Second, the NA message received by Host A comes from an intermediate router, which might not be a registered host on the MT6D Access List, as is seen in the example previously provided by Fig. 3.6. HE-MT6D would need to allow the NA messages from an untrusted router to pass through but still offer protection to Host A. To do so, HE-MT6D would not encapsulate the packet but merely provide IPv6 bilateral address translation services in order to hide Host A's base address. Third, an ICMPv6 Type 1-4 error message packet contains as much of the invoking error packet as possible within its payload. This means that the payload must be examined for a nested packet that possibly needs to be additionally translated or decapsulated. the

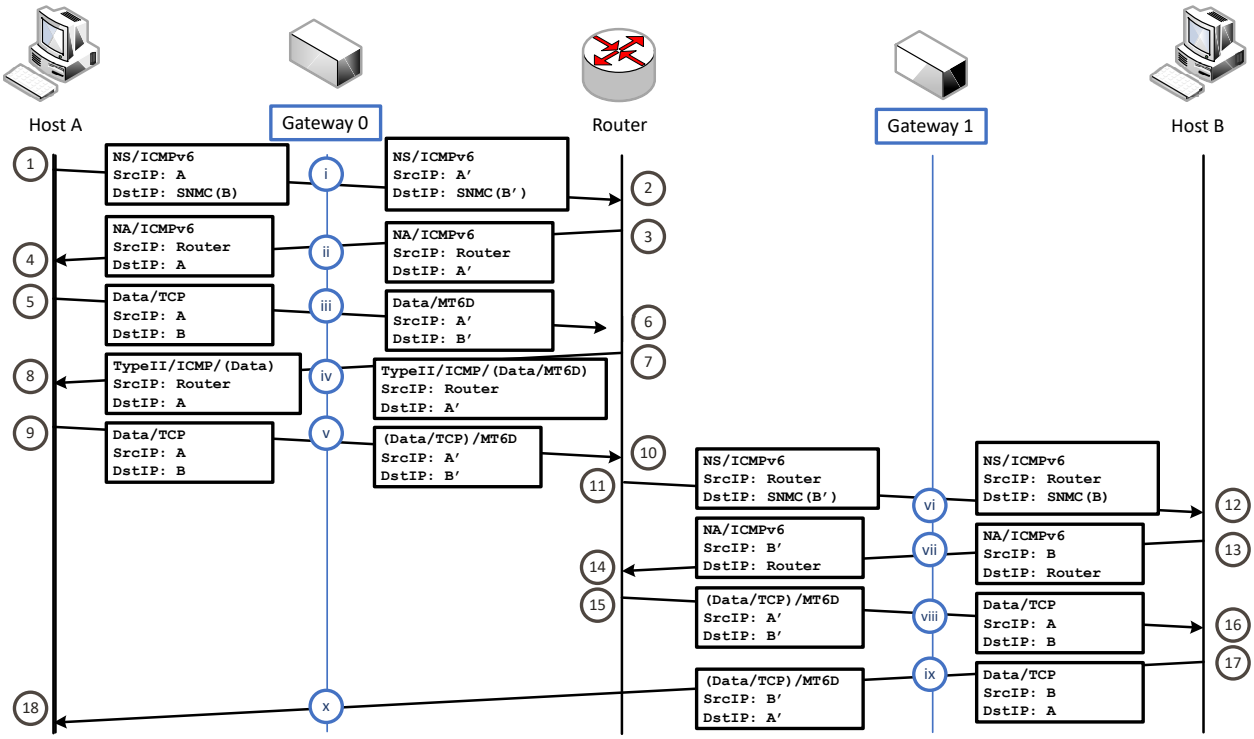


Figure 3.7: IPv6 packet transformations with HE-MT6D gateways inserted. Each packet type and direction combination is handled in a particular manner in order to perform MT6D in a surreptitious manner.

Fig. 3.7 details the individual packet handling done by HE-MT6D for the previous packet conversation in Fig. 3.6. The details are discussed below with the packet case type identified in bold. It is important to note that **TX_A(snmc)** and **RX_B(snmc)** packet case types were discovered after experimentation and evaluation of HE-MT6D as detailed in Chapter 6. They are described and listed here but are not part of the current implementation; rather, they are to be incorporated as future work.

i **TX_A(snmc)**. The packet handled as an outbound packet to a solicited-node multicast address. The packet case type is identified by matching the source IPv6 address and the last 24 bits of the solicited-node multicast address to the Access List as well as matching the IPv6 basic header `next header` field and several ICMPv6 upper layer codes. For action taken on the packet, translation is provided for the source interface address as well as the destination solicited-node multicast address. The packet is not encapsulated, but the source address for Host A is obfuscated with its MT6D translation. The last 24-bits of the solicited-node multicast address also is also translated. As later discovered later during evaluation in Chapter 6, this packet case type was formerly labeled as **TX_B** (any outgoing packet from a protected node to unprotected node). The research presented in this thesis preserves the **TX_B** classification implementation; transition into **TX_A(snmc)** implementation is left as future work.

ii **RX_B**. The packet handled as an inbound packet from an untrusted router to protected node. The packet case type is identified by only having the destination IPv6 address match the Access List. For action taken, translation is provided for only the destination

address. The packet is not decapsulated, but the obfuscated destination address for Host A is translated back to its original value so that Host A can receive it.

iii **TX_A**. The packet handled in this example is a data packet over the connection-oriented TCP protocol from a protected node to another protected node. The packet case type is identified by matching both source and destination IPv6 addresses to the Access List. For action taken, full MT6D encapsulation is provided for the packet. The source and destination IPv6 addresses are translated, and an MT6D extension header generated to create a tunnel. The original data packet header information is embedded into the payload, and the MTU field is incremented by 16.

iv **RX_C(er)**. The packet handled is an inbound ICMPv6 Type 2 error message from an unprotected node. The packet case type is identified by only the destination IPv6 address matching the Access List as well as matching the IPv6 basic header `next header` field and several ICMPv6 upper layer codes. For action taken, since the MTU field is larger than originally allowed, the router generates an ICMPv6 Type 2 error control message to reduce the error size. The MTU adjustment field in this packet must be reduced by an additional 16 octets. The destination IPv6 address must be translated. Also, all ICMPv6 Type 1-4 error messages contain a copy of the invoking error packet. This means that the nested ICMPv6 payload packet must be examined and be fully MT6D decapsulated.

v **TX_A**. Same as (iii).

vi **RX_B(snmc)**. The packet is handled as an inbound packet to a solicited-node multicast

address from an unprotected node. The packet case type is different from **TX_A(snmc)** in not only that it is an inbound rather than outbound packet, but also that the packet originates from an unprotected rather than a protected node. The packet is identified by only having the last 24 bits of the destination solicited-node multicast address match against the Access List as well as matching the IPv6 basic header `next header` field and several ICMPv6 upper layer codes. For action taken on the packet, translation is provided for only the destination solicited-node multicast address. The packet is not encapsulated, but the source address for Host A is obfuscated with its MT6D translation. The last 24-bits of the solicited-node multicast address also is also translated.

vii **TX_B**. The packet is handled as an outbound packet from a protected node to an unprotected node. The packet case type is identified by only the source destination IPv6 address matching against the Access List. For action taken, the packet is not encapsulated, but translation is provided for only the source address to obfuscate it.

viii **RX_A**. The packet is handled as an MT6D tunneled packet from protected node to protected node. The packet case type is identified by both source and destination IPv6 address matching against the Access List. Also, specific IPv6 basic header fields and MT6D extension header fields, to include port numbers, must be correctly qualified. For action taken, full MT6D decapsulation is provided for the packet. The source and destination IPv6 addresses are translated, the MT6D extension header is removed, and the original data packet header and payload are restored.

ix **TX_A**. Same as (iii).

x **RX_A**. Same as (viii).

3.2.3 Accounting for 13 Different Packet Case Types

The specific example shown in sections 3.2.1 and 3.2.3 comprise only 7 different packet case types. However, 13 different packet types were identified so far that facilitate HE-MT6D operation over IPv6. The 6 additional packet case types provide for the handling of the following: transmission of data between two unprotected nodes (**PASS**), ICMPv6 error control messages between protected and unprotected nodes (**TX_A(er)**, **TX_B(er)**, **RX_A(er)**, **RX_B(er)**), and direct communications from protected nodes to other protected nodes without a router (**RX_A(snmc)**). The 13 different packet case types are are summarized in Table 3.1. Notation for case types are $\{TX/RX\}\_\{A|B|C([(er/snmc)]\}$, where TX or RX represent Outbound packets or Inbound packets, respectively. A represents an external trusted node, B represents an external untrusted node, and C represents an intermediate router. The *(er)* represents an ICMPv6 Type 1-4 Error packet, and the *(snmc)* represents communication with a solicited-node multicast address group.

Table 3.1: Thirteen currently identified packet case types. Packet case types are determined during reconciliation before packet assembly. Sources and destinations may be a protected node (P), unprotected node (U), and intermediate router (I). ICMPv6 Type 1-4 Error packets may happen at the beginning of new MT6D transmissions since MT6D adds 16 B to the packet length. If not previously adjust for, the expansion may violate the default MTU of a transmission link and invoke an ICMPv6 Type 2 Error (Packet Too Big) in response. Packets maybe ICMPv6 error packets (er) or directed to solicited-node multicast address groups (snmc).

Classifier	Direction	Src Dst	ICMPv6 Err	Action to perform
TX_A	Outb	P → P		Full Encapsulation
TX_B	Outb	P → U		Only translate protected IPv6 addresses
TX_A(er)	Outb	P → P	X	Encapsulate packet and translate ICMPv6 payload headers
TX_B(er)	Outb	P → U	X	Translate protected IP as well as ICMPv6 payload headers
TX_A(snmc)	Outb	P → MC		Translate protected IP as well as last 24 bits of solicited-node multicast address
RX_A	Inb	P → P		Full decapsulation
RX_A(snmc)	Inb	P→ MC		ICMPv6 Neighbor Solicitation to solicited-node multicast address on local link. Translate source IPv6 address only.
RX_B(snmc)	Inb	U→ MC		Translate protected IP as well as last 24 bits of solicited-node multicast address
RX_B	Inb	U → P		Only translate protected IPv6
RX_A(er)	Inb	P → P	X	Full decapsulation as well as translate ICMP payload headers
RX_B(er)	Inb	U → P	X	Translate protected IPv6 address as well as ICMPv6 payload headers
RX_C(er)	Inb	I → P	X	Translate protected IPv6 address as well as decapsulated ICMPv6 payload
PASS	All others			Pass packet without modification

3.3 Conclusion

Feasibility Study 2 explored some of the basic concepts of IPv6 as well as the enabling protocol exchanges necessary to develop HE-MT6D. The study uncovers 13 packet case types that will be used to fingerprint each packet during preprocessing, further discussed in Section 5.6. These packet case types drive creation of nonlinear single-stage CISC-based packet processor as its corresponding packet assembly language, as further discussed in Section

5.6.3. It is also noted that two of the packet case types, **TX_A(snmc)** and **RX_B(snmc)** were not implemented in the scope of this thesis work. They were discovered and identified as future work during evaluation, as will be seen later in Chapter 6. These two were not necessary for evaluation, as the experimental setup to demonstrate HE-MT6D evaluates performance over a direct connection between two nodes.

The next chapter is System Design, which looks into Research Question 4 and designs an architecture to support HE-MT6D.

Chapter 4

Design: Overview

This chapter sets out to answer the final research question:

5. The HE-MT6D implementation is in essence an NSP. How can an architecture be fully designed using RTL logic? What challenges might be faced during the design process? How can the design be optimized to reasonably maximize data and packet throughput as well as allow for a scalable number of client nodes?

This chapter integrates the lessons learned from Feasibility Studies 1 and 2. With an understanding of the software subsystems, IPv6 communications, and hardware and firmware support needed, this chapter moves forward with design. An overview of the architecture and processes that are implemented in HE-MT6D are presented.

The overall system is designed to accomplish sub-goal 1, in that it should perform functionally. As will be seen later in this chapter, HE-MT6D is as a system of systems, with

many asynchronous and independent sub-modules. On top of each module functioning as it should, maintaining robust control flow was one of the more challenging design aspects. To accomplish sub-goal 2, the architecture was designed to handle incompatible traffic. To accomplish sub-goals 3 and 4 of achieving as close to line rate speeds as possible while using low resources, a novel non-mapped stream-oriented architecture is chosen over traditional mapped memory. The data stream never leaves first in, first out (FIFO) buffers but always remains on the datapath bus from end to end. To accomplish sub-goal 5, a scalable number of nodes supported is provided by a configurable Hybrid tag-based Content Addressable Memory (CAM) subsystem and parameterizable Rotation Driver. It is hybrid, because it provides both the fast initial lookup times of traditional CAM memories but then also completes the lookup process with larger and less resource intensive on-chip RAM.

4.1 Overall Architectural Model

A system topology model is provided by Fig. 4.1. As noted, there are two directions, relative to the protected side. Outbound packets travel from the internally protected side of the engine to the external and untrusted network. Inbound packets traverse in the opposite manner, from the external untrusted network into the protected side of the engine.

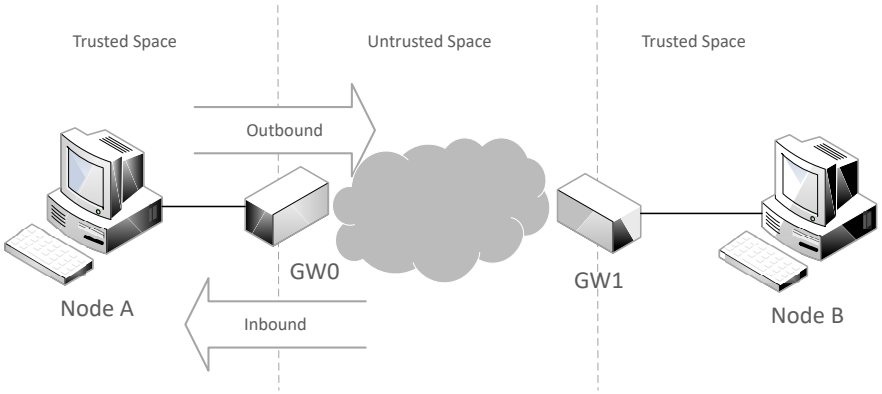


Figure 4.1: The basic topology of HE-MT6D. Each gateway protects trusted space from untrusted space. The direction of movement from trusted to untrusted is considered outbound, and from untrusted into trusted inbound. Outbound traffic is generally encapsulated, while inbound traffic is generally decapsulated.

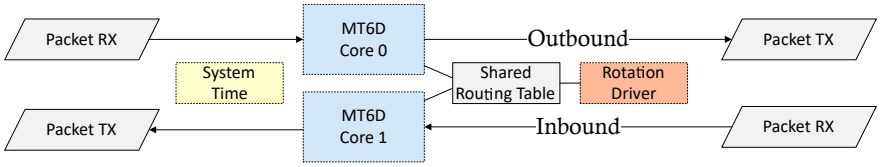


Figure 4.2: Overall system architecture with separation of data and control planes. The data plane comprises the packet forwarding path between receive (RX) and transmit (TX) interfaces. The control plane comprises all other supporting subsystems. Two MT6D processing cores interface the two planes.

HE-MT6D departs from traditional memory-based packet buffering and aims to modify packets in flight. An overall system model is presented in Fig. 4.2. A packet is received in either direction by the RX interface, is sent through an MT6D core for selective processing, and then is finally transmitted by the TX interface. If the packet is not IPv6, it is not qualified for examination and will simply pass through to the final transmission buffer. If there is an error in the packet due to transmission errors or malformed content (such as invalid field values, incorrect packet length, or invalid checksum), the packet will be discarded. If the packet is qualified for examination, one of the two processing cores will conduct deep packet inspection as well as cross check address fields against an Access List (a list of protected nods) for outbound traffic, and a Rotation List (a list of their rotations) for inbound traffic. Both the Access List and Rotation List are stored in the Shared Routing Table (SRT). If a match occurs, the core will classify the packet then encapsulate, decapsulate, or translate it as necessary. Depending upon the determined packet case type during classification, nested encapsulation, decapsulation, or translation may be also necessary. Nested actions happen with ICMPv6 Type 1-4 errors, which include a copy of invoking packets within their payload data. Packet case determinations are discussed in Section 3.2.

The SRT is continuously maintained by the Rotation Driver co-processor module, which constantly checks the Rotation Table for expired nodes and refreshes them with new values.

Initialization is done by writing node profiles $\mathcal{P}_i$ through the Rotation Driver, which then propagate through the system. A profile includes the protected node's IPv6 data $\{\sigma_i, \beta_i\}$,

session key k_i, rotation interval δ_i, and hash select h_i:

$$\mathcal{P}_i = \{\{\sigma_i, \beta_i\}, k_i, \delta_i, h_i\} \tag{4.1}$$

The Access List may contain Node profiles with a heterogeneous mix of rotation periods. The lowest rotation period allowed in HE-MT6D is smallest Unix time unit: one second.

4.2 Control and Data Plane Separation

HE-MT6D provides control and data plane separation. This paradigm is used standard routing and switching architectures [21, 44, 47]. The control plane provides logic for controlling packet forwarding and manipulation behavior. The data plane forwards and moves data according to guidance from the control plane. This methodology allows independent evolution and development of the systems contained in either plane. In the case of HE-MT6D, the data plane is represented by the forwarding path between packet RX and TX interfaces, and the control plane is represented by all supporting modules that are used to make intelligent packet handling decisions. Both planes run independently and concurrently to provide maximum throughput. The two MT6D processing cores as shown in Fig. 4.2 are the interface between the two planes .

This paradigm also helps with debugging and development, as statistics can be pulled from modules without any affect on forwarding and processing rates.

4.3 Stream-based Packet Buffering

Common buffering methods are to offload packets into Static RAM (SRAM), special purpose Dynamic RAM (DRAM), multiple DRAMs in parallel, or a hierarchy of SRAM and DRAM [30, 26, 25, 32, 51, 33]. However, for HE-MT6D, we chose an in-flight stream-based processing design and used only FPGA on-chip memory to keep resource utilization low as well as to facilitate nonlinear packet modification with dynamic packet expansion and contraction. It is possible to expand and contract packets using traditional SG-DMA controllers in tandem with statically mapped memories, but doing so is much more complex and lends itself more towards embedded processor control rather than a full RTL environment.

Evaluation from Chapter 6, Fig. 6.3, shows that the stream-based architecture does well and drops approximately 0.03% of packets during line rate UDP throughput stress testing, the same as performance of a straight through Ethernet cable, although it does overflow during encapsulation due to packet expansion. Intermediate stream-based buffers throughout the system generally saw less than 3 KB of buffer utilization. A final overflow buffer is incorporated to contend with overflows and prevent data corruption.

The general idea of in-stream packet processing is illustrated in Fig. 4.3. Packets that are received are immediately sent right back out for transmission. However, an arbiter substitutes, modifies, inserts, or removes 32-bit words of the data stream from prepared injection data based upon feedback from a pre-processor. Adding or removing words from the data stream is accomplished by selectively manipulating flow control.

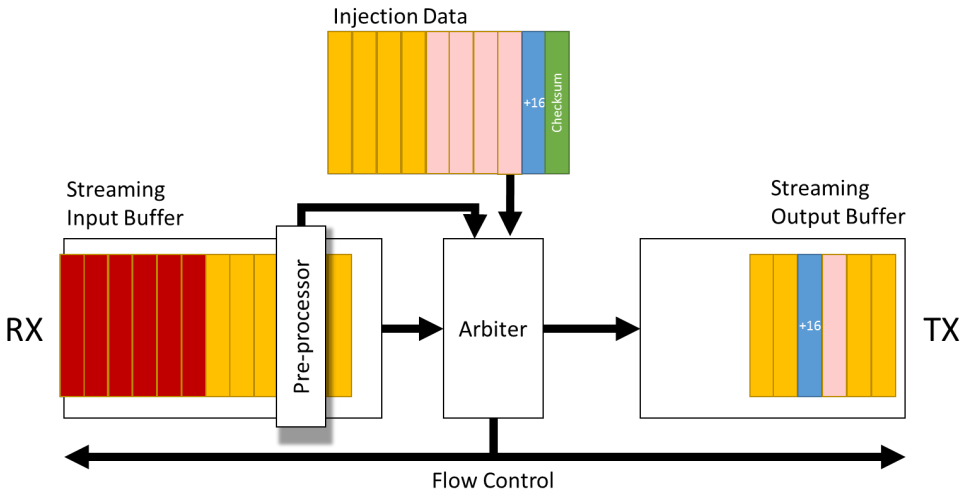


Figure 4.3: General concept of in-stream processing. Packets are sliced into 32-bit words and are transmitted using the Altera Avalon Streaming interface (Avalon-ST) interface. Every clock cycle, words progress across the datapath directly from the receive interface to the transmit interface. As data moves through the system, it is analyzed and manipulated. If storage is needed, the datastream is never deposited into addressable memory. Rather the data words are streamed into and out of a series of first in, first out (FIFO) queues. This architecture allow for nonlinear in-flight packet manipulation without the need for building complex SG-DMA descriptors requisite in traditional static-memory-based packet buffering.

4.4 Process Flows

There are four main processes in HE-MT6D: time synchronization, system initialization, system maintenance, and the processing datapath. Each process is asynchronous and independent. Time synchronization, system initialization, and system maintenance comprise the control plane. Processing datapath comprises the data plane. The control plane and data plane are separate. The following sections are organized by process and logically describe the signal flows used in HE-MT6D as well as how they generally relate to each other at a high level.

4.4.1 Time Synchronization

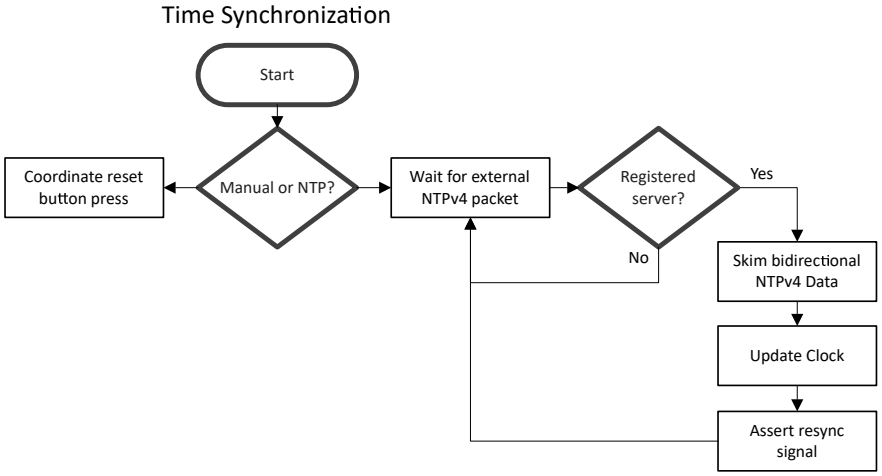


Figure 4.4: Time synchronization process flow. Local device time may be set by manual synchronization or by observing trusted Network Time Protocol v4 (NTPv4) traffic.

Time synchronization is the first and simplest process and is illustrated in Fig 4.4. There are two methods, manual or NTPv4.

Manual synchronization requires reseting all system synchronously by physically pressing the reset button on all systems. Deviation tolerance is the lowest rotation interval for a loaded node, which is generous at ±1 second or longer.

NTPv4 synchronization is more complicated and involves an external agent, as HE-MT6D does not generate NTPv4 packets but simply listens for server-client transactions. If HE-MT6D detects an NTPv4 client request on the wire, it checks to see if the time server is registered as trusted. If so a custom NTPv4 module will skim the time fields off the packet. When the server response comes back through, the module will skim the server's response time fields off the packet, perform the requisite time calculations, and then adjust the system clock. Time is maintained in Network Time Protocol (NTP) time, which uses an epoch of January 1st, 1900 [37]. However, MT6D uses Unix time, which uses an epoch of January 1st, 1970—70 years ahead of NTP time—so the time is adjusted during hash requests. Time server registration is done during system initialization.

4.4.2 System Initialization

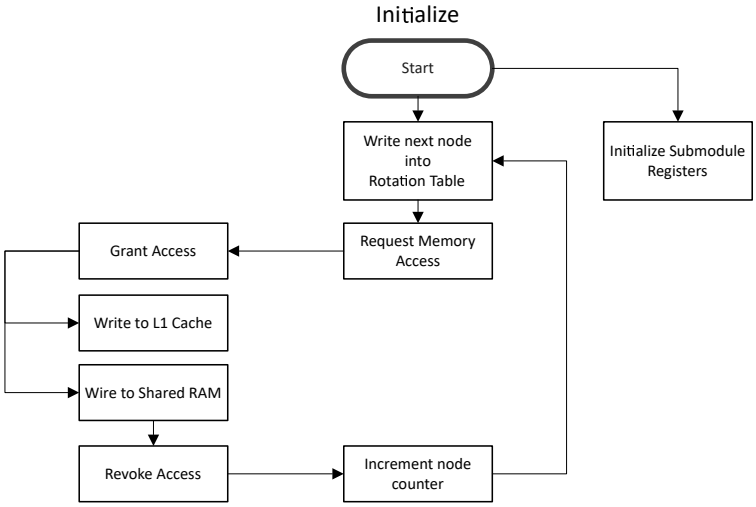


Figure 4.5: System initialization process flow. Initialization writes Access List entries into memory as well as initializes system modules.

The system initialization process is shown in Fig. 4.5 and involves initializing system modules as well as establishing what is called the Access List for HE-MT6D. The Access List contains a list of protected nodes along with each node's specific configuration data. Initialization is done via Nios II coprocessor during system boot. Once the system is initialization, the Nios II no longer contributes to HE-MT6D operation but does collect statistics.

Initialization of system modules is done by writing to a sequence of control registers. It includes registering an NTPv4 time server, configuring and resetting the PHY, and configuring and resetting the MAC function. Once complete, the Nios II goes on to initializing MT6D protected nodes.

Initialization of protected of nodes is done by writing into the Rotation Driver co-processor. The final destination of these records is the SRT in memory shared between the MT6D cores. Since the MT6D processing cores maintain direct control and access over the SRT,

the Rotation Driver must request memory access, which is done by requesting a memory access token. Once in possession of the token, the Rotation Driver writes through the L1 HCAM cache into the SRT. The L1 HCAM cache stores a tag of each record to accelerate later lookups. When done writing, possession of the token is revoked. The initialization process continues until all nodes are written.

The SRT holds both the Access List and Rotation List. The Access List is the record of initialization and original IPv6 data for each protected node. The Rotation List is collocated with the Access List and contains the record of hashed IID pairs for each node.

4.4.3 System Maintenance

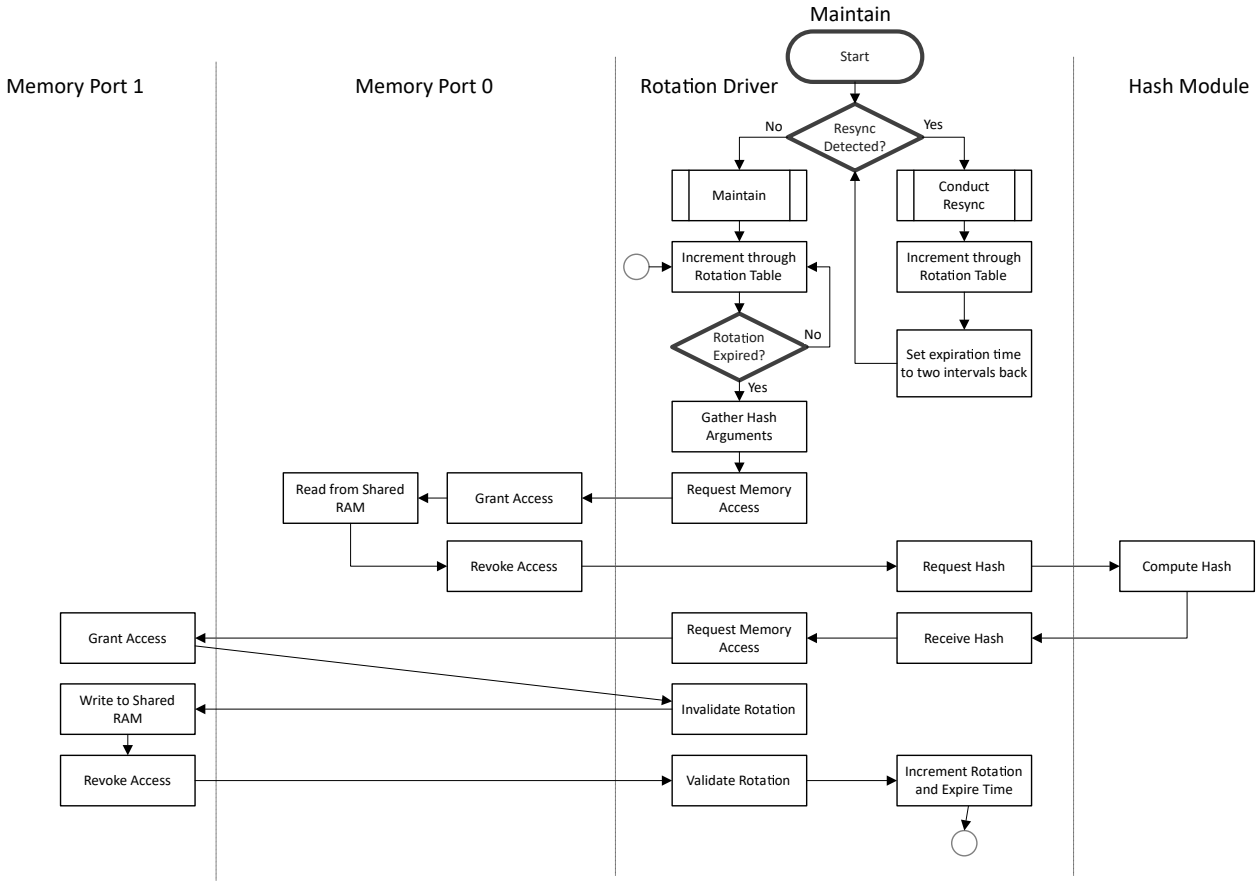


Figure 4.6: System maintenance process flow. System maintenance ensures all HE-MT6D devices stay in global synchronization by locally maintaining and updating Shared Routing Table (SRT) records and user-defined policies.

System maintenance is depicted in Fig. 4.6. There are two parts: resynchronization and regular maintenance. Both parts rely on an internal Rotation Table, shown in Table 5.1, which maintains a record of the expiration time, current rotation pointer, and rotation valid array for each node.

Resynchronization is performed each time a new set of nodes is written into the system, it is also used if the time is adjusted by the NTPv4 module. Resynchronization updates expiration timers and forces a refresh of the Rotation List for all nodes in the system.

Regular maintenance is otherwise performed. During regular maintenance, the Rotation Driver increments through its Rotation Table looking for expired entries. When found, the Rotation Driver gathers hashing arguments from memory, which again requires requesting a memory access from an MT6D processor core. Arguments are packaged and sent to the Hash Engine for a new hash digest, which is then written to the Rotation List. After writing to the Rotation List, the Rotation Driver updates its records in the Rotation Table and resumes looking for expired nodes.

4.4.4 Processing Datapath

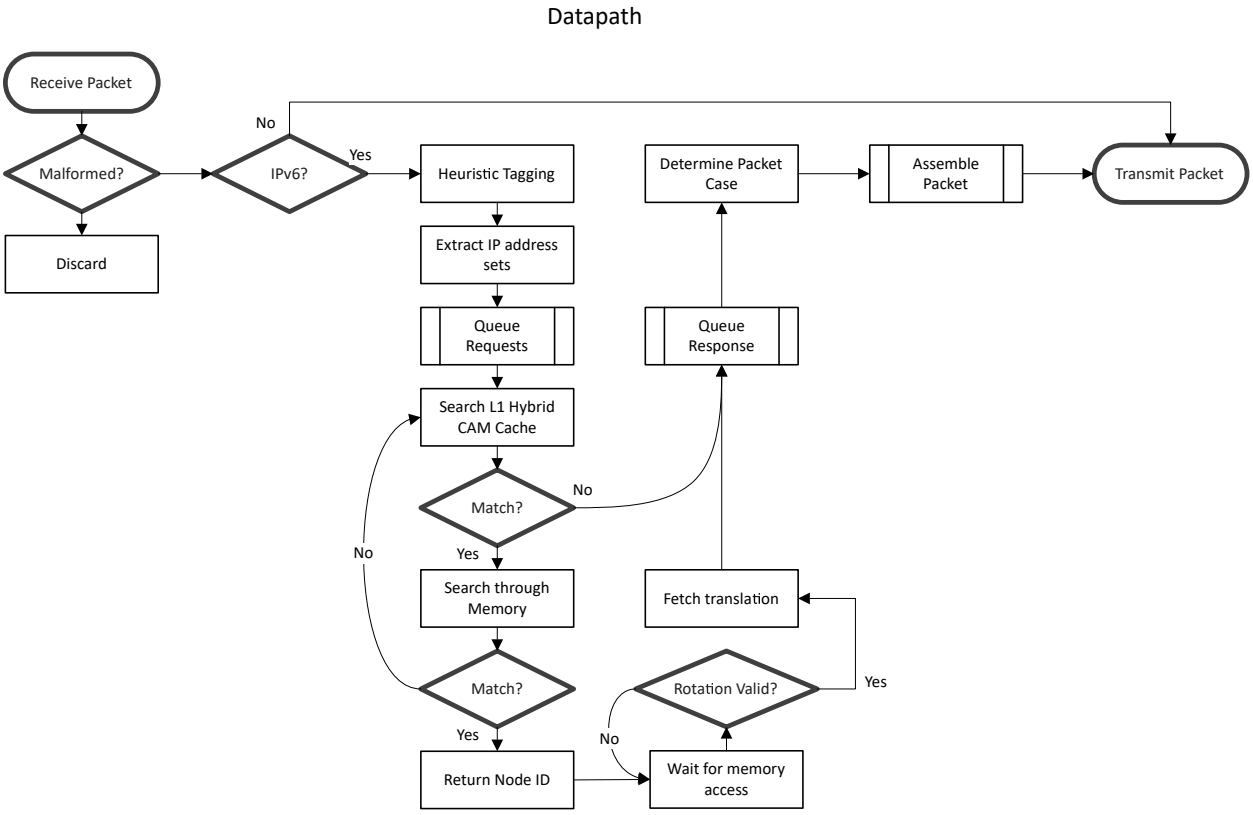


Figure 4.7: Datapath process flow. The Datapath dynamically analyzes each packet, then uses extracted metadata and information from the Shared Routing Table (SRT) to discriminate each packet. Qualified packets are then reconstructed according to user-defined policies.

The processing datapath is depicted in Fig. 4.7 and is the path each packet takes between ingress into the system and egress back out. The datapath first checks for packet errors and discards malformed packets. It then splits the data stream into IPv6 and non-IPv6 traffic. Every IPv6 packet is then fingerprinted with metadata describing internal fields, and IPv6 addresses are extracted into individual lookup requests (there are up to three IPv6 addresses sent per packet). The Broker Module then cross checks each request against either the Access List or Rotation List to retrieve a corresponding Node Identifier (Node ID) for each address. The Broker uses the Node ID if returned to retrieve the corresponding translation response. Translation responses and extracted metadata are finally used to determine the packet case and how to handle the packet. The Packet Assembler then manipulates the packet as necessary and places the packet onto the output buffer for transmission.

4.5 Conclusion

This chapter goes over the overall stream-based architectural model and the four process flows involved: time synchronization, system initialization, system maintenance, and the processing datapath. Each part of the system is designed to help accomplish a research objective design sub goal as well as the primary objective of moving MT6D to hardware in such a manner that supports the ultimate design of an ASIC MT6D gateway device.

The next chapter goes into detail about the systems used in these four processes, and their corresponding submodules.

Chapter 5

Implementation: a System of Systems

This chapter is the implementation of Chapters 2, 3, and 4, and continues the design process but in more detail. HE-MT6D is a system of systems. The full system design was inspired by the Nios II UDP Offload Example [2] and Wang et al.'s design A Gbps IPsec SSL Security Processor Design and Implementation in an Prototyping Platform [50]. Both made heavy use of streaming FPGA architectures as well as used channelization to split the data stream into multiple processing paths.

The main architecture is shown in Fig. 5.1 and has seven subsystems with various internal modules that perform specialize functions. Each one of these subsystems will be discussed in the sections following.

1. Statistics and initialization

2. Time

3. Rotation Coprocessor

4. Hash Engine

5. Memory Subsystem (HCAM and Shared Memory)

6. MT6D Cores. With the following submodules:

   Field Extractor (extracts metadata and IPv6 addresses)

   Broker

   Packet Assembler (performs reconciliation and packet assembly)

7. Datapath

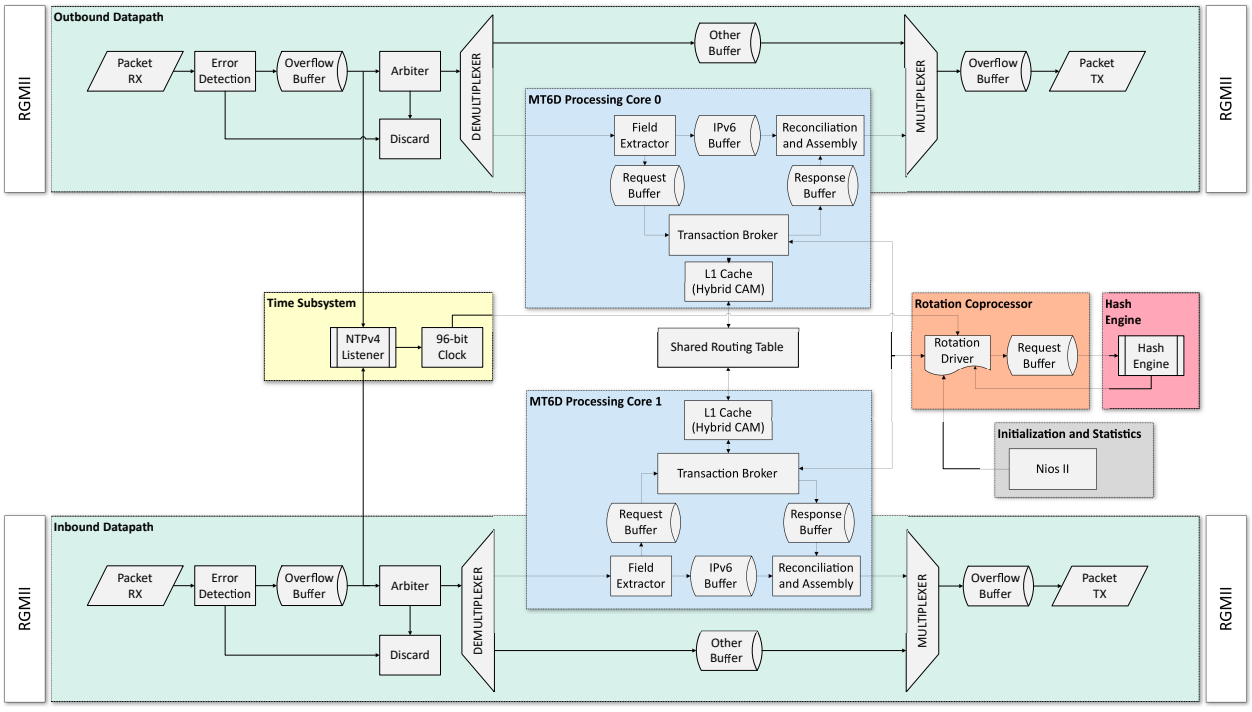


Figure 5.1: Expanded system architecture. The general system architecture of Network Security Processor and Hardware Engine for MT6D (HE-MT6D), with each subsystem and its comprising modules is shown.

5.1 Statistics and initialization Subsystem

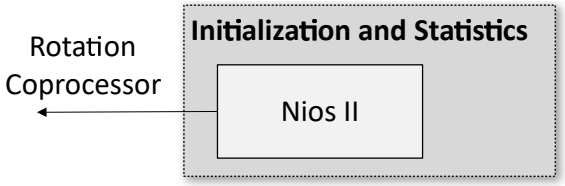


Figure 5.2: Initialization and Statistics subsystem.

The statistics and initialization subsystem contains only the Nios II soft processor and is depicted in Fig. 5.2. The Nios II soft processor is used to initialize PHY, MAC, register a trusted NTPv4 server, write nodes into the Access List. Once HE-MT6D is initialized, the only function of the Nios II is to read statistics counters throughout the whole system for data collection and debugging. The Nios II otherwise provides no operating function or support for HE-MT6D. To populate the Access List, the function `load_node()` is called on an array containing node initialization parameters. The Access List would be populated by all deployed HE-MT6D gateways with the same data.

5.2 Time Subsystem.

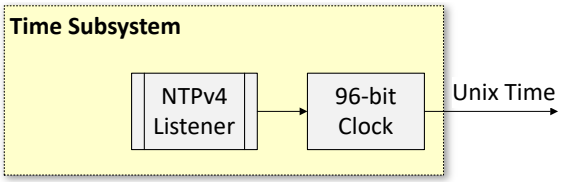
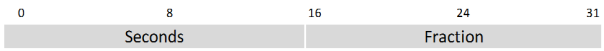


Figure 5.3: The Time subsystem.

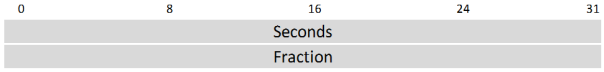
The time subsystem comprises two modules: Clock96 and the NTPv4 Module. Clock96 is a

tunable 96-bit system clock based on DE2-115 50 MHz time crystal and seeded by NTPv4 via the NTPv4 module, as illustrated in Fig. 5.3.

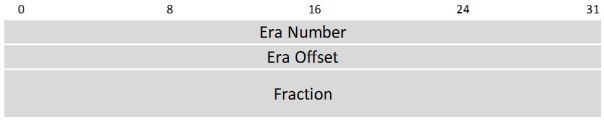
Clock96. Structure of the Clock96 module follows and abbreviated form of NTP data format as shown in Fig. 5.4c, except Era Number is truncated and assumed as NTP Era 0, beginning January 1, 1990. It is helpful to note that the MT6D protocol relies on Unix time, which begins at January 1, 1970. Therefore, 70 years is added to the clock for time read by any HE-MT6D module.



(a) Short format



(b) Timestamp format



(c) Date format

Figure 5.4: NTP time formats. The 64-bit timestamp format is transmitted within NTPv4 packet time value fields. The date format is a 128-bit record that is used when space is sufficient [37]. The tunable clock used in HE-MT6D is a 96-bit variant of the date format, with the epoch field removed always calculated as NTP Era 0 (zero time begins January 1, 1900). NTP belongs to the application layer of the OSI model.

NTPv4 Module. The NTPv4 module is a simplified implementation of RFC 5905 and observes both inbound and outbound datapaths as seen in Fig. 5.5 [37]. The module does not initiate any requests but instead relies on listening for clients NTPv4 requests to a trusted server. The module is pass-through and incurs no cycle penalties against the datapath. The module qualifies all packets that pass through in either direction and latches on to any NTPv4 client request traveling to a previously registered server. When observed, it records the time fields as depicted in Fig. 5.6, and begins a timeout counter. When the module hears the corresponding server response, it will extract the time fields and perform a T1 sanity check according to RFC 5905 to protect against replay attacks [37]. If the sanity check passes, the module then computes clock offset and round trip delay using equations 5.1 and 5.2. After either accepting the server response or timing out, the module resets its state.

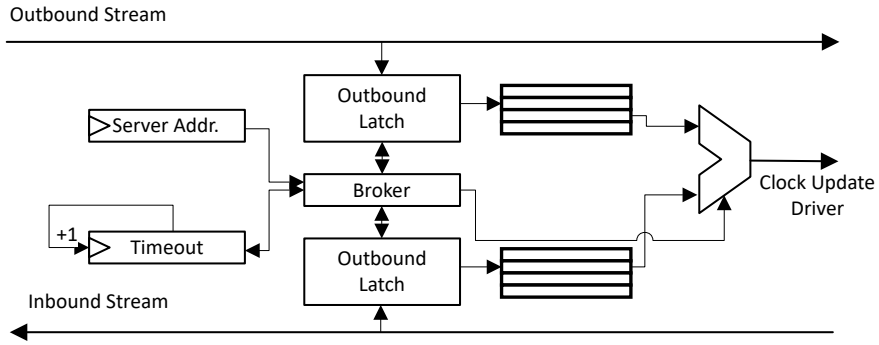


Figure 5.5: Network Time Protocol v4 (NTPv4) Module design. The NTPv4 Module is a pass-through design and does not incur cycle penalties against the datapath. A trusted time server is first registered with the module during initialization. The module then listens for NTPv4 requests and responses and uses information gleaned from packets to calculate clock offset and round trip delay. Sanity checks according to RFC 5905 are performed to prevent replay attacks.

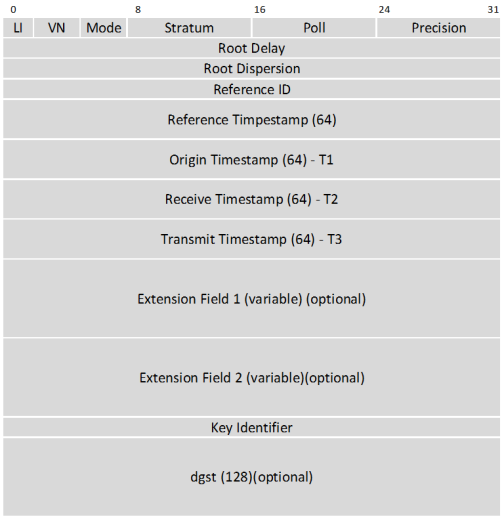


Figure 5.6: NTPv4 packet format. The NTPv4 packet format shows 64-bit time fields T1, T2, and T3. T4 is the received local time. NTPv4 is an application layer protocol is a UDP datagram [37]. It belongs to the application layer of the OSI model.

$$\text{clock offset} = ((T_2 - T_1) + (T_3 - T_4))/2 \tag{5.1}$$

$$\text{round trip delay} = (T_4 - T_1) - (T_3 - T_2) \tag{5.2}$$

5.3 Rotation Coprocessor

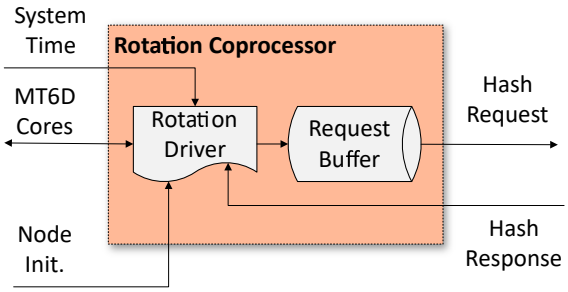


Figure 5.7: Rotation Coprocessor Subsystem. The Rotation Coprocessor is a single module that writes initial node profiles to the SRT, provides system initialization, node synchronization, and provides continuous system maintenance.

The Rotation Coprocessor is a single module that writes initial node profiles to the SRT, provides system initialization, node synchronization, and provides continuous system maintenance. It is depicted in Fig. 5.7. For reading from or writing to the SRT, the Rotation Driver must first request memory access from one of the two MT6D processing cores.

Initialization. For initialization, node profiles $\mathcal{P}_i$ of node n_i are writing into the Rotation Driver Module by the Nios II coprocessor. The format of $\mathcal{P}_i$ is shown in Table 5.2. In order to write $\mathcal{P}_i$ into memory, the Rotation driver must first request control of the memory access token from the outbound MT6D core, as this specific core is the entry point into writing to the Access List. In a likewise manner, the inbound MT6D core controls write access to the Rotation List, as will be discussed in the maintenance process.

When access is granted, the Rotation Driver has direct write-through access into dual-port on-chip RAM via the L1 HCAM write-through cache. The cycle immediately after the Rotation Driver deasserts `read` or `write`, the memory access token is revoked, and the Rotation Driver must wait again to write the next profile $\mathcal{P}_{i+1}$ into the SRT. In specific,

each MT6D Core has an internal module called the Broker Module that manages its token and only gives up the access token when either between packet analysis periods or while the Broker Module is waiting for the current hash to be update.

When writing to the Access List in memory, the Rotation Driver makes an internal copy of the current rotation time $\epsilon_i(U)$, current rotation j_{curr}, and valid state of each rotation in memory **val** for each node n_i in the Rotation Table, as seen in Table 5.1. The Rotation Table is actively checked to ensure the rotations stored in the Shared Routing Table are current and valid. The table is maintained local to the Rotation Driver module to allow the MT6D cores as much unfettered access to memory as possible and retain priority to Translation Lookup Requests on the datapath.

Table 5.1: The Rotation Table. The Rotation Table is indexed by Node ID and maintains the next expiration period, current rotation pointer, and valid array.

Index	Current Expiration Time	Current Rotation	Valid Array
n_i	$\epsilon_i(U)$	$j_{curr} = j(U)$	**val[2:0]**

Table 5.2: Shared Routing Table (SRT) data structure entries. Each node profile is written into shared memory. Node profiles are stored and maintained as 80 B states in the Shared Routing Table (SRT).

Offset	Entry
0	Base IID (β_i)
1	Base Subnet (σ_i)
2	Session key low ($k_{i63\rightarrow 0}$)
3	Session Key high ($k_{i127\rightarrow 64}$)
4	Rotation 0 IID ($\phi_{i,j_0 79\rightarrow 16}$)
5	{`hash_sel`, Period (δ_i), 16'bx, MT6D Port ($\phi_{i,j_0 15\rightarrow 0}$)}
6	Rotation 1 IID ($\phi_{i,j_1 79\rightarrow 16}$)
7	{ 48'bx, ($\phi_{i,j_1 15\rightarrow 0}$)}
8	Rotation 2 IID ($\phi_{i,j_2 79\rightarrow 16}$)
9	{ 48'bx, ($\phi_{i,j_2 15\rightarrow 0}$)}

Resynchronization. Resynchronization forces quick convergence of all nodes to the proper expiration times. Resynchronization is necessary, as otherwise, expiration timers for each n_i first initialize to `0`, and it may take an inordinate amount of time to step up to the current time, requesting a new hash in between each step.

During initialization, the Rotation Driver increments through the rotation table and sets the expiration time of each node to two intervals prior to current time:

$$\epsilon_i = U - (U \mod \delta_i) - \delta_i \text{ for all } n_i \in \mathcal{N} \tag{5.3}$$

Setting the expiration time in two intervals in the past forces two updates for each node.

Maintenance During normal operation, unless another resynchronization is called, the Rotation Driver will maintain its Rotation Table by incrementing through and looking for any expired nodes. If an expired node is found, the Rotation Driver will request a new hash for that node, then increment the expiration time and current rotation pointer:

$$\begin{aligned} &\text{For each } n_i \in \mathcal{N}, \text{ if } U \geq \epsilon_i : \\ &\qquad \phi_{i,j(U+\delta_t)} = H(\beta_i || k_i || \epsilon_i (U + \delta_t) \\ &\qquad t_{exp}(i) = t_{exp}(i) + \delta_t \\ &\qquad j_{curr} = j_{curr} + 1 \mod j_{max} \end{aligned} \tag{5.4}$$

The expiration time and current rotation pointer will be stored in the Rotation Table, while the newly generated IID pair $\phi_{i,j(U+\delta_t)}$ will be sent to memory. To update the Rotation List, the Rotation Driver will request memory access from the Inbound MT6D processor core. While writing, the Rotation Driver deasserts the `valid` bit associated with the record being written. When complete, the `valid` bit is reasserted.

5.4 Hash Engine

The hash engine is modular in design, which allows any hash algorithm to be inserted with an appropriate hardware wrapper. There are eight hash slots available. Three `hash_select` bits in a node's profile selects the algorithm to use. In HE-MT6D, only SHA256 is used, with the IP of the implementation available from OpenCores.org, written by `marsgod` [35].

Due to system timing constraints, the hash engine clock is half the system clock frequency at 50 MHz. Still, the hash engine is able to fulfill hash requests in less than 2 μs, which is over 500k fulfillments a second, more than sufficient for initializing and maintaining far fewer nodes.

The hash engine is presented in Fig. 5.8 with eight modular channels. Each channel can be used for future testing of hash algorithms other than SHA256.

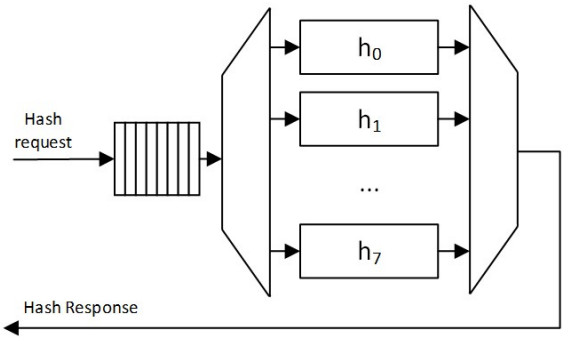


Figure 5.8: Modular Hash Engine design. The modular Hash Engine design allows for future protocol experimentation using different hashing or encryption protocols for address translation. The three `hash_sel` bits of a node's profile data are used to select which crypto submodule $h_{\texttt{hash\_sel}}$ to use.

5.5 Memory Subsystem

The system has two independent memory lanes, one per packet stream processor, that reads and writes into a single dual-port shared on-chip memory. Coherence is maintained by use of a memory access token for each lane that is managed by the Broker Module. Priority access is given to datapath Translation Request lookups. Secondary access is given to external I/O (the Rotation Driver) during system initialization or rotation updates.

Address translation request are aided by a write-through L1 cache that is designed as a hardware configurable hybrid CAM, which is designated as the HCAM. Each processor has its own dedicated HCAM, which enables two-cycle lookups for a scalable number of tags derived from each node's stored address or rotation IID pair. Each HCAM contains a search engine, with a configurable number of sub-engines, each 16 wide. For instance, the current implementation has one sub-engine for outbound HCAM traffic, and three for inbound (since each node has up to three concurrent IID pairs). With this configuration, in just two cycles 16 tags are simultaneously searched on the outbound path, and 48 on the inbound. Tags are the last μ_d of a stored IPv6 address record.

The SRT is implemented in Dual-Port shared memory, with each core having independent direct access. Memory coherence is maintained by use of a Broker Module that maintains a memory access. During writes to shared memory, both the inbound and outbound Broker Modules have access to their own dedicated tag-based L1 HCAM write-through cache.

5.5.1 Reconfigurable Hybrid Content Addressable Memory (CAM) (HCAM)

There have been vast studies in efficient IPv6 address lookups. Most can be categorized into into trie-based systems [31, 41, 46] or hashed-based solutions [15, 19].

Trie bases solutions are a tree-type lookup method that use a branch-leaf system that follows symbol match strings until the last leaf is reached. The entire matched chain then corre-

sponds to a complete subset of IP addresses. Trie-based fast memory lookup system may be beneficial for looking up base address nodes on the Access List, but since rotation IID pairs are hash-based and psuedorandom, tries would not be an effective means of lookups.

Hash-based search solutions would be more applicable. The design developed is inspired by [19], which uses multiple lookup engines in a reconfigurable search tree. [15]'s use of Bloom filters is most appealing and could further accelerate the matching process, as Bloom filters are probabilistic data structures that help eliminate false negatives and can tell whether an item is possibly or definitely not in a set. They can perhaps be utilized in future designs. However, since a low number of nodes are supported, the design presented more than suffices for its objective and can support a scalable number of nodes.

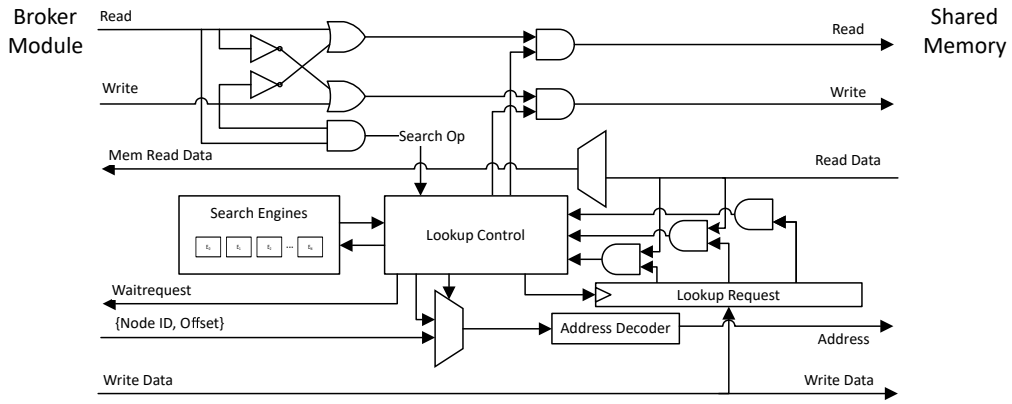


Figure 5.9: The Hybrid Content Addressable Memory (HCAM) Module Design. The HCAM Module Resides between each Moving Target IPv6 Defense (MT6D) Core's Broker Module and the dual port shared on-chip RAM. Each HCAM (inbound and outbound) has its own dedicated port connection into RAM.

A diagram of the HCAM Module is provided in Fig. 5.9. For an overview of operation, the Broker Module first writes a target string onto the `writedata` bus and simultaneously asserts `read` and `write` signals. The HCAM receives the 144-bit search request (comprising

a 128-bit IPv6 address and 16-bit UDP port) and extracts the tag η_{TARGET}, sending the tag to its internal search engine to find a match. The search engine then looks for stored potential tags to match the search request. If a match is found, the search engine returns an `f_hit` flag and the Node ID of the potential match to its internal control sequence. The HCAM Modules then reads from memory the data corresponding with matched Node ID 64-bits at a time (since the shared RAM is 64-bits wide) to see if the rest of the data in memory matches the full `writedata` bus. If either all 144-bits or a partial 128-bits match, the HCAM returns the Node ID n_i to the Broker Module on the `readdata` bus. Once, n_i is returned, it is then up to the Broker Module to fetch the appropriate translation from memory through a normal read operation. During the entire search operation, the HCAM Module asserts the `waitrequest` signal until its search is complete.

Enabling the HCAM is its search engine, a configurable array of cascaded sub-engines. Fig. 5.10 shows the overall search engine structure, and Fig. 5.11 shows its more detailed structure.

As seen in Fig. 5.11, each search engine features a configurable number of sub-engines that each allow parallel search 16 tags per row. During operation, a row value is addressed to all sub-engines. A search operation requires two cycles per round. For a search operation, the target value η_{TARGET} is also fed into the system. In the first cycle of a round, the address of lowest-address matching tag is fed to a registered aggregator. Of these lowest-address matched tags per engine, the aggregator choses the lowest position and decodes the value.

Each tag is μ_d bits wide, which determines the collision resistance of the module. During

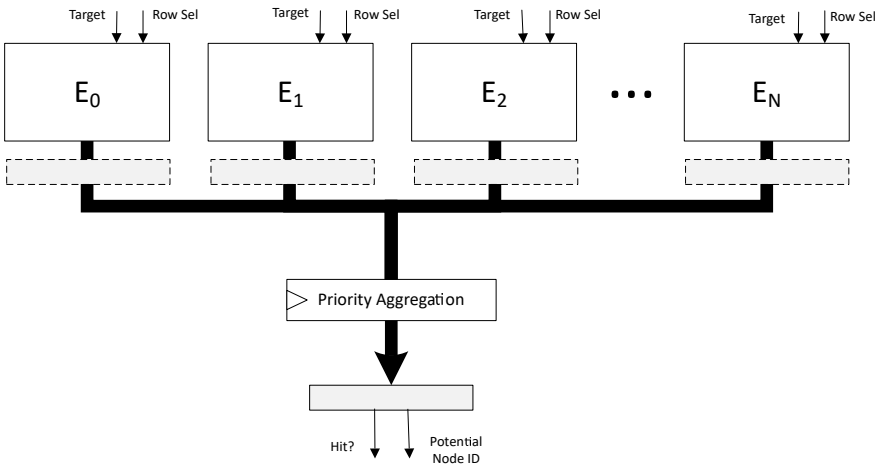


Figure 5.10: Overview of an HCAM internal search engine. The intermediate results of each engine is aggregated and if there is a hit, only the lowest matching Node ID is returned.

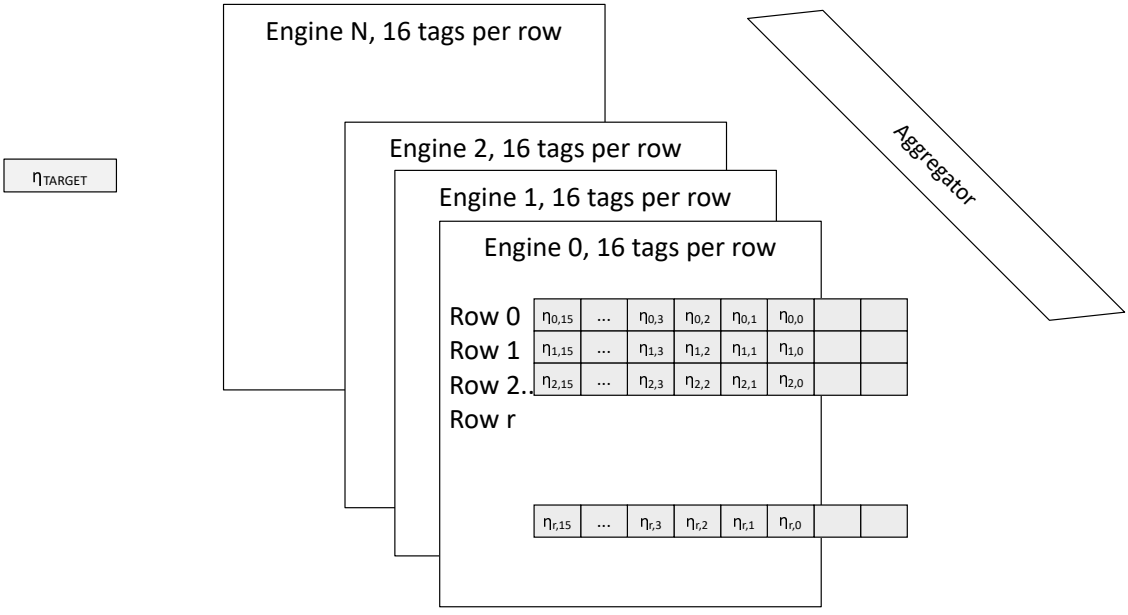


Figure 5.11: HCAM internal search engine in detail. Within each HCAM is a series of engines running in parallel. Each engine stores a multiple of 16 tags per row (16, 32, 48, etc.), with the number of rows depending upon the total specified number of nodes supported.

a search operation, a row is selected. The all columns of the row are then searched in parallel. For three sub-engines, for example, 48 tags are searched in parallel. Any matches are compared to a skip array that flags previously checked values. If there are any matches that are not flagged by the skip array, the node with the lowest Node ID number is returned and hit flag raised for the prospective match.

The location of the returned tag is decoded into the Node ID n_i. For the Outbound HCAM, tag $\eta_{O,r,c}$ for node n_i is located at row i/w_n and column $i \mod w_n$. During a write to memory, the value stored at tag $\eta_{d,r,c}$ is the lowest μ_d bits of the base IID being written:

$$\eta_{O,r,c} = \beta_{\mu_O \to 0} \tag{5.5}$$

For the Inbound HCAM, tag $\eta_{I,r,c}$ for node n_i is located at row i/w_n and column $(i \mod w_n)+ j_{curr} \cdot w_n$. During a write to memory, the value stored at tag $\eta_{d,r,c}$ is the lowest μ_d bits of the hashed IID pair being written:

$$\eta_{O,r,c} = \phi_{i,j(t)_{\mu_I \to 0}} \tag{5.6}$$

The number of collision bits μ_d for each tag η_d specified by each tag is configurable.

During rotation record writes, the address $\phi_{i,j(t)}$ is sent to is computed as:

$$address = 10 \cdot i + 4 + j \cdot 2 \tag{5.7}$$

5.5.2 Shared Memory

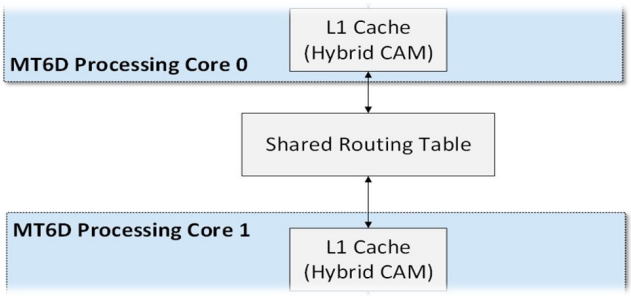


Figure 5.12: On-chip shared memory. The on-chip shared memory containing the Shared Routing Table (SRT) resides between the two MT6D cores, directly attached to each L1 HCAM wire-through cache module. The SRT contains the profiles of each node n_i.

Fig. 5.12 shows the positioning of the dual-port on-chip shared memory between the two MT6D processing cores. The SRT is stored in this shared memory. There is no common memory bus to this memory module. Rather, each of the MT6D processing cores is directly attached to its own port, allowing each core simultaneous and independent read and write access to memory. To allow the Rotation Driver access to memory to maintain the Rotation List, each core is equipped with an external I/O interface and uses a memory token to divert access when needed.

To maintain memory coherence of both the SRT and each L1 HCAM cache, only the outbound core can write to the Access List, and only the inbound core can write to the Rotation List.

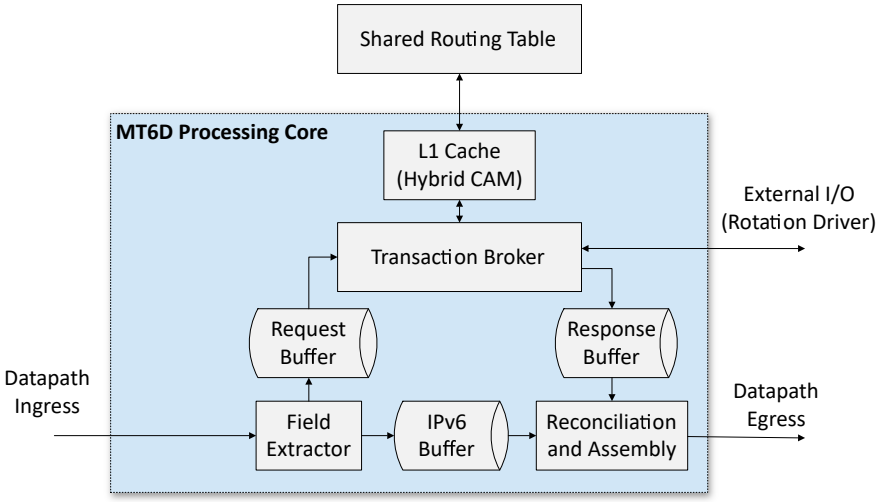


Figure 5.13: MT6D packet processing core design. There is one core each per datapath (outbound and inbound). Each core interfaces the control and data planes. Extracted information about each packet is sent into the control plane, which returns to the Packet Assembler information needed to manipulate the data stream as necessary as packet pass through in-flight. An intermediate IPv6 FIFO buffer temporarily queues packet fragments (in 32-bit slices) as they arrive while the control plane performs translation lookups.

5.6 MT6D Processing Cores

HE-MT6D is a dual-core Network Security Processor. One core is dedicated to processing the inbound datapath, and the other core for the outbound. Fig. 5.13 features a diagram of one of these cores. Each core has two packet pre-processors, the Field Extractor Module and the Broker Module. The Field Extractor extracts IPv6 address sets from each packet and sends them to the Broker to be translated if applicable. The Field Extractor simultaneously extracts metadata fingerprints from each packet the help the Packet Assembler later determined what sort of packet is being processed and how to operate on it. The Broker looks at information it receives from the Field Extractor and uses the L1 HCAM cache as well as the SRT to search for the proper address translation, which it then sends to the Packet

Assembler. Metadata from the Field Extractor and translation information from the Broker is collected and reconciled at the Packet Assembler Module, which then determines each packet's case type. The Packet Assembler then uses packet assembly instructions associated with that case type in order to manipulate packets as they pass through.

The subsections below describe the function of each of these Modules in more detail.

5.6.1 Field Extractor Module

The Field Extractor Module analyzes packets as they stream into the MT6D core an onto an IPv6 datapath buffer. As the packets stream by, the Field Extractor extracts IPv6 address sets $\{a_0, p_0, \{a_1\},p_1\}$ and $\{a_3,p_3\}$, as applicable and sends them to the Broker Module as a Translation Lookup Requests. Each address set contains a full IPv6 address and potential MT6D EH port address (same as the UDP port address for encapsulated packets). At the same time, the Field Extractor extracts and encodes metadata from each packet. This metadata contains travel direction, IPv6 Basic Header next header, next header option, and nested next header values. The metadata is placed within the channel data of each respective packet, which would later be used by the Packet Assembler along with data returned from the Broker Module, to determine what each packet's case type is.

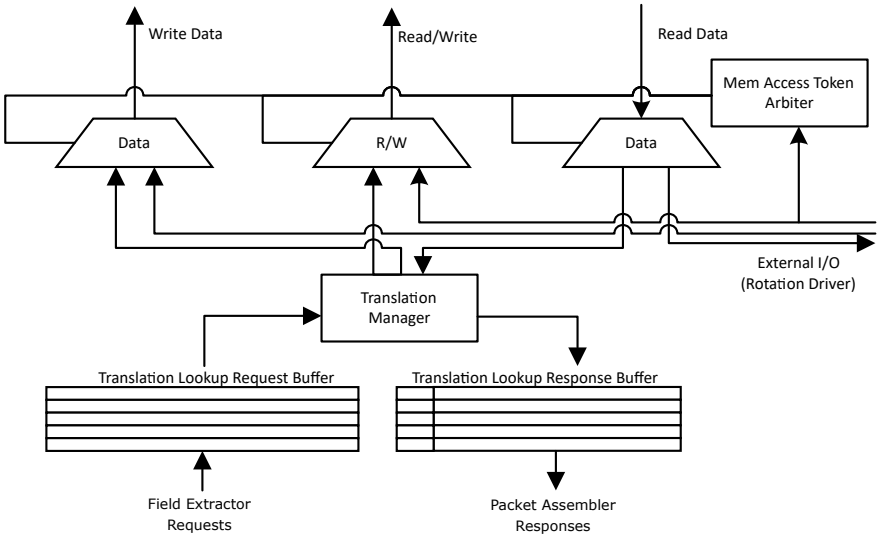


Figure 5.14: The Broker Module. The Broker Module brokers both packet translation lookup requests from the Field Extractor as well as memory access requests from the Rotation Driver co-processor as required.

5.6.2 Broker Module

The Broker Module brokers both packet translation lookup requests from the Field Extractor as well as memory access requests from the Rotation Driver co-processor as required. Fig. 5.14 features a diagram of the Broker Module.

The first function of the Broker Module is to broker translation lookup requests. The translation lookup process is a `value-key`, `key-value` operation. This means an initial `value` is first received (the IPv6 address pair), to return a `key` (Node ID). This `key` (Node ID) is then used to retrieve the final `value` (the translated IPv6 address pair). The initial request value is known as the Translation Lookup Request (TLRQ). The final returned response value is known as the Translation Lookup Response (TLRS). The Broker Module processes TLRQs by sending them to the HCAM module for memory search operations, which would

return the key value of the Node ID if found. The Broker Module then uses the Node ID to read from memory the appropriate address translation to form the TLRS response value and places it onto the response buffer.

The second function of the Broker Module is to broker memory access for the Rotation Driver co-processor. When the Rotation Driver needs to access records in memory, the Broker module uses a memory access token to grant it access. The Broker Module revokes the token immediately after the Rotation Driver is done with its read or write operation. During TLRQ processing, the Broker Module will lock the memory access token and send the IPv6 address set needing to be examined to L1 HCAM cache to perform the requires search operations.

Standard Outbound Packets. On an outbound translation requests, the Broker will examine each address set a_s of packet $\mathcal{P}_O$, where a_0 represents the IPv6 packet's source address, and a_1 represents the IPv6 packet's destination address, in order to determine whether either corresponds with a protected node on the Access List:

$$n_i = Q_O(a_s) \text{ for } s \in \{0, 1\} \tag{5.8}$$

If identified, the Broker then searches for the corresponding IID pair $\phi_{i,j(t)}$ based upon the current rotation for that node, then returns a response to the Packet Assembler along with

asserting a "hit". If the Broker fails to find the entry, it will return a "miss" and all zeros:

$$R_O(n_i, t) = \begin{cases} \{1, \phi_{i,j(t)}\} & \text{for hit} \\ \{0, 0\} & \text{for miss} \end{cases} \tag{5.9}$$

Completed responses are then sent to the Packet Assembler. During packet assembly, the original address pair $\{\alpha_i,\beta_i\}$ will be replaced by the translated $\{\alpha_i, \phi_{i,j(t)}\}$ during the translation process.

Standard Inbound Packets. For inbound packets, a similar process occurs. the Broker will examine each address set a_s of packet $\mathcal{P}_I$, where a_0 represents the IPv6 packet's source address, and a_1 represents the IPv6 packet's destination address, in order to determine whether either corresponds with a protected node on the IID List.

$$n_i = Q_O(a_s) \text{ for } s \in \{0, 1\} \tag{5.10}$$

If identified, the Broker then reads the corresponding original base IID pair β from memory and sends the pair to the Packet Assembler as a response. If not, it returns a miss:

$$R_I(n_i) = \begin{cases} \{1, \beta_i\} & \text{for hit} \\ \{0, 0\} & \text{for miss} \end{cases} \tag{5.11}$$

Special Cases. Unless the protected client's MTU is adjusted, one of the first outbound MT6D packets $\mathcal{P}'_O$ packet should return as an ICMPv6 Type 2 Error "Packet too big". In the case of MT6D, outbound packets $\mathcal{P}_O$ are expanded by 16 octets in order to accommodate for the MT6D headers. If the original packet is at full 1500 B MTU size, adding 16 octets will cause the outgoing packet to violate the standard 1500 B MTU size. In that case, either the next switch or router will send the packet back as ICMPv6 Type 2 error in order to have the client adjust its MTU size.

In the special case ICMPv6 Type 1-4 errors, the source address a_0 may be an unprotected interface, and there may exist a condition where not just the packet itself must be examined, but also the error-invoking payload. The invoking payload source address set is represented as a_2 and destination set as a_3. Since offending packets are returned to the original sender, it can be surmised that $a_2 = a_1$. That is, the invoking packet source address set represents packet $\mathcal{P}_I$'s destination address, and it is not necessary to examine a_2. In short, only three address sets must be examined — $\mathcal{P}_I$'s source address a_0, destination address a_1, and invoking source address a_3— in order to make a determination on if the ICMPv6 payload must also be decapsulated:

$$n_i = Q_O(a_s) \text{ for } s \in \{0, 1, 3\} \tag{5.12}$$

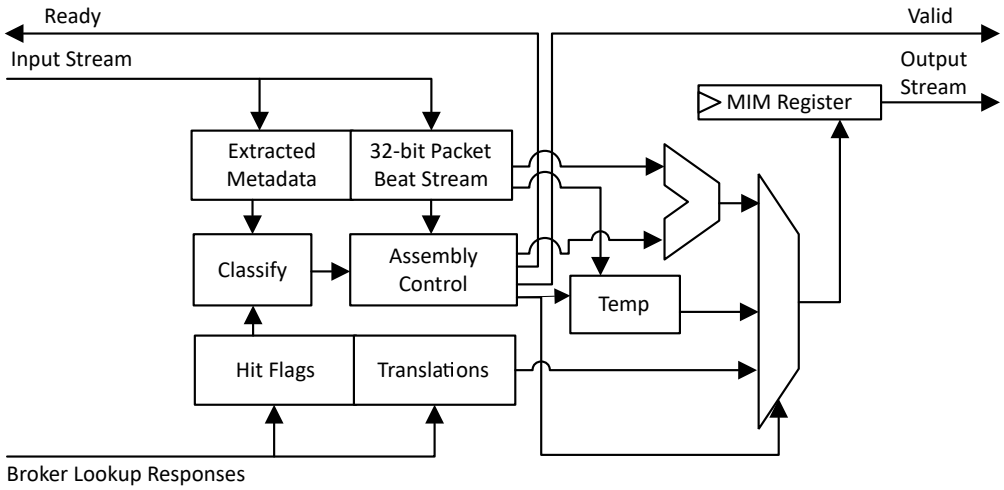


Figure 5.15: Single-stage pipeline nonlinear Packet Assembler Module. The Packet Assembler is a CISC based packet processor and the heart of the HE-MT6D packet encapsulation, decapsulation, translation, and modification. It uses extracted metadata as well as Translation Lookup Response data to classify packets, then manages control flow while injecting new values as applicable into the datapath.

5.6.3 Packet Assembler Module

The Packet Assembler Module is a nonlinear single-stage CISC Packet processor and is seen in Fig 5.15. The module includes a reconciliation method to determine what type of packet is received using a combination of the metadata extracted from the Field Extractor as well as the collection of Translation Lookup Responses received from the Broker Module.

The Packet Assembler is a CISC based packet processor and the heart of the HE-MT6D packet encapsulation, decapsulation, translation, and modification. To handle different packet cases, a trajectory, or string of opcodes is used. Once a packet case has been identified according to the criteria of Table 3.1, an associated assembly trajectory shown in Table 5.6 is used. All these determinations are done during the reconciliation process of the Packet Assembler. In the reconciliation process, the Packet Assembler determines a packet's case

type using information from the previously collected metadata located in a packet's channel data as well as from the Translation Lookup Responses from the Broker Module.

The Packet Assembler is nonlinear since it can dynamically delete, add, replace, insert 32-bit word beats without regard to tracking relative position of each word to the head of the frame. This nonlinear strategy enables operation of multi-layer encapsulations and decapsulations. Further, the stream-based architecture used is much more suited for this type of work, as compared to static SG-DMA buffer designs, which would require creating complex scatter-gather descriptors to reassemble a packet from different portions of memory. The Packet Assembler is single-stage, as it requires only a one-cycle delay to make all packet modifications.

Reconciliation. Reconciliation is completed through packet qualification according to Table 5.3. The packet Case Type is determined depending upon certain qualification met by each packet, which include the travel direction, IPv6 Basic Header next header, next header option, nested next header values, and the combination of returned hits from the Broker Module. A snippet from the Reconciliation cycle is shown in Listing 5.6.3.

Table 5.3: Reconciliation Decision Matrix. The Packet Assembler uses metadata from the Field Extractor and confirms the packet case type using the collection of hit responses from the MT6D Broker Module.

Packet Type Case	Metadata Qualifiers						Broker Hit Resp
	Direction	NH	ICMP Code	ICMP .NH	MT6D .NH	MT6D .icmp.cd	$a_0 \vert a_1 \vert a_3$
TX_A	Outb	X					1 1 X
TX_B	Outb	X					1 0 X
TX_A(er)	Outb	0x3a	0x1-4		0x3a		1 1 1
TX_B(er)	Outb	0x3a	0x1-4				1 1 X
RX_A	Inb	0x11					1 1 X
RX_A(snmc)	Inb	0x3a	0x87				1 0 X
RX_B	Inb	X					0 1 X
RX_A(er)	Inb	0x11			0x3a	0x1-4	1 1 1
RX_B(er)	Inb	0x3a	0x1-4				0 1 1
RX_C(er)	Inb	0x3a	0x1-4	0x11			0 1 0
PASS	-	Everything else					

```
STATE_SRT_DETERMINE_PACKET_TYPE:
begin
// This used to be a wire with assignments but was causing too much
   combinational delay
// f_hits[0] source
// f_hits[1] dest
// f_hits[2] invoking dest
// f_hits[x][1] At least IP match
// f_hits[x][0] Both Port and IP match
// A - protected node
// B - unprotected node
// 0 - Non ICMP Type 1-4 Error
// 1 - ICMP Type 1-4 Error
  packet_type <=
    f_hits[0][1]& f_hits[1][1]& outb &        !f_icmperr          ? CASE_TX_0A :
    f_hits[0][1]&!f_hits[1][1]& outb &        !f_icmperr          ? CASE_TX_0B :
    f_hits[0][1]& f_hits[1][1]& outb &         f_icmperr          ? CASE_TX_1A :
    f_hits[0][1]&!f_hits[1][1]& outb &         f_icmperr          ? CASE_TX_1B :
    f_hits[0][0]& f_hits[1][0]&!outb & f_udp &!f_icmperr &
                        !f_icmperr_over_mt6d                      ? CASE_RX_0A :
    f_hits[0][0]& f_hits[1][0]&!outb & f_udp &!f_icmperr &
                        !f_udp_over_icmp & f_icmperr_over_mt6d    ? CASE_RX_1A :
    f_hits[0][1]&!f_hits[1][1]&!outb &         f_mcndp      ? CASE_RX_0A_MCNDP:
   !f_hits[0][1]& f_hits[1][1]&!outb &        !f_icmperr          ? CASE_RX_0B :
   !f_hits[0][1]& f_hits[1][1]&!f_hits[2][1] &!outb & f_icmperr ? CASE_RX_1B :
   !f_hits[0][1]& f_hits[1][1]& f_hits[2][0] &!outb & f_icmperr
                       & f_udp_over_icmp & !f_icmperr_over_mt6d ? CASE_RX_1C :
```

```
                                CASE_PASS;
27    state_srt_in      <= STATE_SRT_RES_FULLY_ABSORBED;
28 end
```

Listing 5.1: Reconciliation Cycle

Packet Assembly Primitives. The packet assembly language developed is built upon four basic primitives, as shown in Table 5.4. These primitives drive control flow manipulation of the Packet Assembler Module's `ready` and `valid` signals as well as direct manipulations for adjusting 32-bit word slices of packets as they stream by.

Table 5.4: Fundamental Reduced Instruction Set Computer (RISC) instructions. These instructions are the building blocks for the Complex Instruction Set Computer (CISC) packet action opcodes. These RISC primitive actions are accomplished by pausing or passing both output and input streams independently to achieve the desired effect, as well as replacing data frames at dictated moments.

Action	Sink	Source	Description
Pass	`ready`	`valid`	Opens both input and output streams; all data is passed forward.
Drop	`ready`	`!valid`	Pauses output stream; newly received input is deleted.
Pause	`!ready`	`valid`	Pauses input stream; allows injection of new words into the datapath.
Replace	`ready`	`valid`	Maintains both input and output streams open, but substitutes the steam with data from the man-in-the middle attack register

CISC Instruction Set Architecture (ISA). Building upon these primitives are the CISC instructions outlined in Table 5.5. Each of these CISC instructions performs a specific packet manipulation and control flow task.

Table 5.5: CISC instructions for hardware execution. The basic Pass, Drop, Pause, and Replace Primitives are used to create Complex Instruction Set Computer (CISC) instructions for hardware execution. Note that checksums were calculated but not used during experiments covered in this research.

OPCODE	Hardware Action
`PASS IMM#`	Passes IMM# 32-bit words without modification.
`ENC0`	Performs the first part of MT6D encapsulation. This instruction records the first two words of the IPv6 basic header (`ver`, `tc`, `fl`, `pl`, `nh`, `hl`) and replaces them with dummy. values.
`ENC0`	Inserts the MT6D extension UDP header. Pauses the input stream while inserting MT6D ports, adjusted packet length, and adjusted upper layer checksum.
`RPLC REG0`	Replaces the current IPv6 address seen with the translation response indexed by REG0. Acceptable values of REG0 are (`0x0-0x2`).
`MTU`	Adjusts the MTU by adding 16 octets (outbound) or subtracting 16 octets (inbound).
`DCAP REG0, REG1`	Performs decapsulation using the address translation indexed by REG0 and REG1. Acceptable values for both are 0x0-0x2). This instruction pauses the output stream to drop 12 words while recording both subnets of the IPv6 basic header. It then pauses the input stream to insert two complete address translations.
`PSUB`	Subtracts 16 from the payload length field.
`REST`	Passes the rest of the packet.
`CKSM`	Passes the rest of the packet, but also identifies the location of the upper layer checksum field based on the packet fingerprint and replaces it.

For example, `PASS, 0x4` is generally performed as the first instruction for all packets. This instruction passes along the first four 32-bit words of the datapath, which contain the Ethernet frame header (source and destination MAC addresses and the EtherType). Next may be `ENC0`, which is the first part of the MT6D encapsulation process. This instruction is a two-cycle instruction. During the first cycle, the instruction asks the processor to perform a replacement of the first word of the IPv6 basic header—`version, traffic class, flow label`—and record the original values for later injection. The replaced value is the generic `0x60000000`. During the second cycle, the instruction asks the processor to do the same for the second word (`payload length, next header, hop limit`) but increases the payload

length by 16 and changes `next header` to `0x11` for UDP traffic: `0x[+0x10]11ff`. Further CISC instructions are strung together to complete manipulation of the packet.

Packet Assembly Trajectories. Each packet case type is mapped to a corresponding trajectory of CISC instructions. When the Packet Assembler Module receives the metadata from the Field Extractor as well as the Translation Lookup Responses from the Broker Module, the Packet Assembler reconciles the two together to determine the packet case type. From this determined packet case type, the Packet Assembler uses the assigned packet assembly trajectory listed in Table 5.6 to assemble the packet accordingly. These trajectories are handling instructions for each packet case type. With this packet assembly language, if new packet types are discovered, handling instructions can be built by creating new packet assembly trajectories without rebuilding too much of the Packet Assembler module.

Table 5.6: Packet Assembly Trajectories. Depending on the packet type determined, the Packet Assembler will operate on packets according to set assembly instructions.

Classifier	CISC Packet Assembly Trajectory
TX_A	`PASS, 0004, ENC0, RPLC, 0000, RPLC, 0001, ENC1, REST`
TX_B	`PASS, 0006, RPLC, 0000, PASS, 0004, REST`
TX_A(er)	`PASS, 0004, ENC0, RPLC, 0000, RPLC, 0001, ENC1, MTU, PASS, 0002, RPLC, 0001,`
	`RPLC, 0000, REST`
TX_B(er)	`PASS, 0006, RPLC, 0000, PASS, 0004, MTU, PASS, 0006, RPLC, 0000, REST`
RX_A	`PASS, 0004, DCAP, 0000, 0001, REST`
RX_A(ns)	`PASS, 0006, RPLC, 0004, REST`
RX_B	`PASS, 0010, RPLC, 0001, REST`
RX_A(er)	`PASS, 0004, DCAP, 0000, 0001, MTU, PASS, 0002, RPLC, 0001, RPLC, 0000, REST`
RX_B(er)	`PASS, 0010, RPLC, 0001, MTU, PASS, 0002, RPLC, 0001, REST`
RX_C(er)	`PASS, 0005, PSUB, PASS, 0004, RPLC, 0001, MTU, DCAP, 0001, 0002, REST`
PASS	`REST`

5.6.4 Processing an Example Packet

Let us give and example that involves decapsulation of an inbound MT6D-encapsulated packet from a protected node send to another protected node and containing an ICMPv6 Type 1-4 Error (perhaps an MTU Packet Too Big error).

In this example, a packet enters the inbound MT6D processing core through the Field Extractor. The Field extractor would extract the IPv6 addresses a_0, a_1, a_3, and send them to the Broker Module as Translation Lookup Requests. The Field Extractor would then extract packet metadata as {`inbound`, `nh = 0x11`, `mt6d.nh = 0x3a`, `mt6d.icmp.code=0x2`} and place an encoded version of the metadata into the packet's channel data. The packet and its channel data would pass into a datapath buffer to wait. At the same time, the Packet Assembler would wait until the Broker Module returns all of the required Translation Lookup Responses before processing the queued packet. As it waits, the Packet Assembler receives the hit array $\{a_0|a_1|a_2\} = \{11X\}$. When all Translation Lookup Responses are received, the Packet Assembler performs reconciliation according to Table 5.3 and resolves the received packet as case type `RX_A(er)`, which corresponds with the following packet assembly trajectory:

```
PASS    0x4
DCAP    0x0, 0x1
MTU
PASS    0x2
RPLC    0x1
PRLC    0x0
REST
```

Once the packet assembly trajectory is determined, the Packet assembler streams the packet

off the datapath buffer and manipulates the packet accordingly. This specific trajectory passes the first four 32-bit words, performs decapsulation using the translation responses `0` and `1` returned by the Broker Module (in that specific order), makes an MTU adjustment, passes another two words, translates the IPv6 address with translation response `1`, then again translates another IPv6 address with translation response `0`, then finally sinks the rest of the packet.

5.7 Datapath

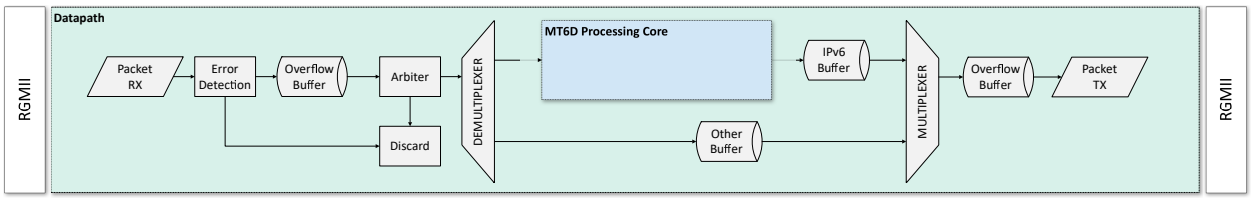


Figure 5.16: The Datapath. The Datapath forms the data plane and is completely separate from the control plane. Data packets are stored in FIFO streaming buffers.

The Datapath forms the data plane and fully contains the data packets as they stream through. During HE-MT6D evaluation, the system clock is set at 100 MHz, and the primary datapath transmission bus is a 32-bit Avalon Streaming interface (Avalon-ST), providing 3.2Gbps max theoretical throughput. Avalon-ST supports packetization to mark the beginning and end of packets and up to 128 bits of channel data for side-band communication to tag each packet as needed [5].

A Nios II co-processor is used to populate the access list for initial configuration, register a trusted NTPv4 server, initialize the off-chip Marvell 88EE1111 1Gbps transceiver over

the Management Data Input/Output (MDIO) interface, and read statistics counters. After initialization, it does nothing else to impact the datapath.

The Datapath includes a components necessary to transcieve packets from the RGMII interface, qualify, and channelize them and is shown in Fig. 5.16. Packetization is done by use of `startofpacket` and `endofpacket` flags, which mark the start and end of packetized data. These flags are stored with each 32-bit word of data as they progress through the system. Modules send data through a source interface, and receive them through a sink interface. Control flow is provided by a `valid/ready` handshake method, where data is moved forward only when the sink module asserts `ready`, and data is only accepted from a previous module when the source module asserts `valid`. When the next module deasserts `ready`, it is said to be applying backpressure. Overflow buffers are used at strategic points to prevent data corruption when backpressure is applied for an extended time. These buffers are store-and-forward, which means that the packets are fully stored before they are transmitted.

The datapath is a bus 32-bits wide, with separate data and channel streaming buffers. Data FIFO buffers store packet data, while channel FIFO buffers store associated information about each packet (sequence numbers,metadata, etc). The data buffer is written in 32-bit increments, while the channel buffer varies but is only written to once per packet.

Modules include Altera's TSE Megacore, Error Packet Discard (EPD), Overflow Buffer (OVF), a Channelizer and Checksum arbiter (CC), demultiplexer, channel processors, multiplexer, and a Final Overflow Buffer (FOVF). The TSE provides the MAC function and interface with RGMII, which passes on packets received by the PHY. The EPD discards

packets with transmission errors. The OVF buffers packets but discards them when an overflow condition arises; use of OVF in this manner prevents data corruption if the system applies too much backpressure. The CC arbiter tags packets with what channel they will be on depending upon their contents. The channels available are IPv6 (which all go through MT6D processing), Other (all other types of packets), Discard (for malformed packets), and Pass All (a debugging channel to simply pass traffic and provide no MT6D processing). Packets are split apart into separate channels by the demultiplexer, then processed or buffered accordingly. Packets are then recombined into a single stream by the multiplexer and buffered for transmission at the FOVF. Another TSE transmits packets back onto the PHY through the RGMII interface.

5.8 Conclusion

HE-MT6D is a system of systems. The Time, Rotation Coprocessor, Hash Engine, Memory (HCAM and Shared Memory), MT6D Processing Core subsystem comprise the control plane, and the Datapath subsystem comprises the data plane. Both form a full NSP architecture.

Of note, although HE-MT6D is designed to target development of an ASIC, the modular nature of its Intellectual Property (IP) also allows it to be packaged as dedicated hardware sub-engine within a larger ASIC design. This opens design options beyond simple ASIC design.

The next chapter evaluates HE-MT6D in accordance to how well it meets the research goals

and ultimately proves or disproves the research hypothesis.

Chapter 6

Evaluation of Hardware Engine for HE-MT6D)

This chapter evaluates the performance of HE-MT6D as it relates to all research objectives.

6.1 Evaluation Platform

The development and evaluation platform for this research is the Terasic DE2-115 Cyclone IV development board with 115k Logic Elements, 486 KB of embedded memory arranged in 432 M9K memory blocks, 50 MHz base oscillator, 4 PLLs with maximum frequency of 472.5 MHz, and two 10/100/1000 Mbps TSE Ethernet ports [29]. Ethernet interface `E0` is used as the internal interface that faces the trusted space, and `E1` the external that faces the untrusted space. Outbound connections traverse from `E0` to `E1` and inbound connections

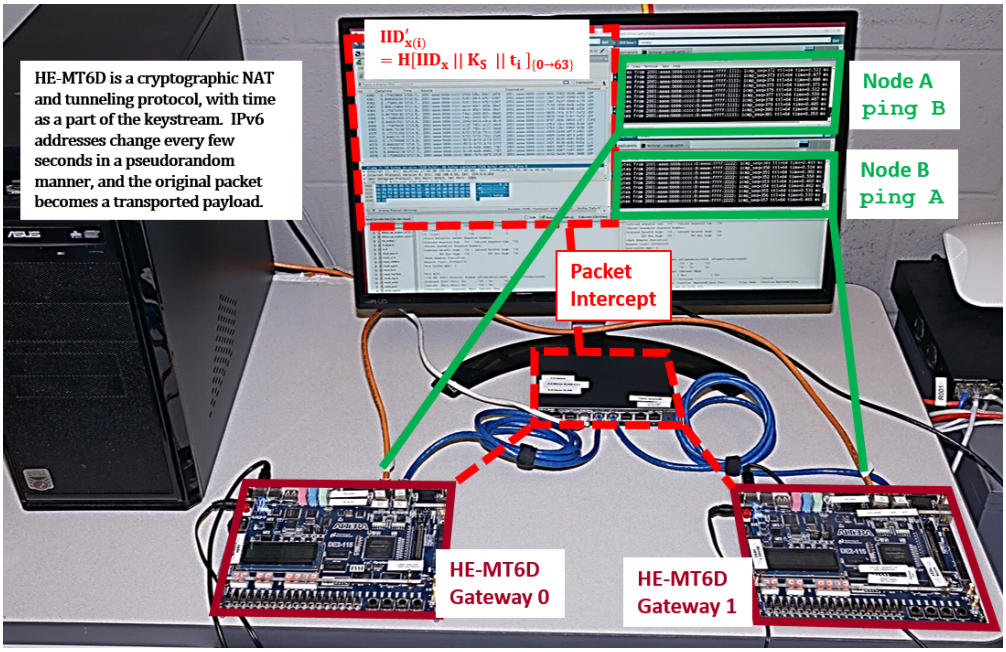


Figure 6.1: Evaluation platform. The evaluation platform comprised a desktop running XenServer 7.0 with multiple virtual machines, an Intel quad-port 1 Gbps Ethernet NIC, two FPGAs loaded with HE-MT6D IP configured as gateways, and a 1 Gbps switch with port mirroring to observe and verify packet structures.

vice versa.

Packet sniffing and performance testing was done with multiple Kali Light 2.0 virtual machines. Each virtual machine was assigned 2 GB of RAM and two virtual cores. The host hypervisor was XenServer 7.0 running on a quad core 3.30 GHz Intel i5-2500K processor with 32 GB of DDR3 RAM and an Intel PRO/1000 PT Quad-Port Gigabit Ethernet NIC PCI-E Adapter (HM9JY). In-line packet inspection was aided via a TP-Link 8-Port Gigabit Ethernet Easy Smart Switch (TL-SG108E) configured for man-in-the-middle port mirroring observation. The TL-SG108E was only used at low speeds, as the port mirroring on this device was limited to approximately 500 Mbps. For high speed tests, statistics were reported directly from the FPGA over JTAG/UART from statistics counters as well as observed by software run on the virtual machines. Benchmarking software used was `ping`, `iperf3`, and `wireshark`. The setup is seen in Fig. 6.1

6.2 Network Communications Performance

To benchmark HE-MT6D, only peer-to-peer connections were examined at this time. Routers introduce their own processing delays that are avoided in this specific battery of HE-MT6D testing. For all testing below, a direct wire connection from Host A to Host B serves as a control, shown in Fig. 6.2b. All other tests have HE-MT6D gateways between the connection of the two machines as shown in Fig. 6.2c. Before tests commence, correct MT6D operation is confirmed with packet captures from a third machine using Wireshark; this is done by using

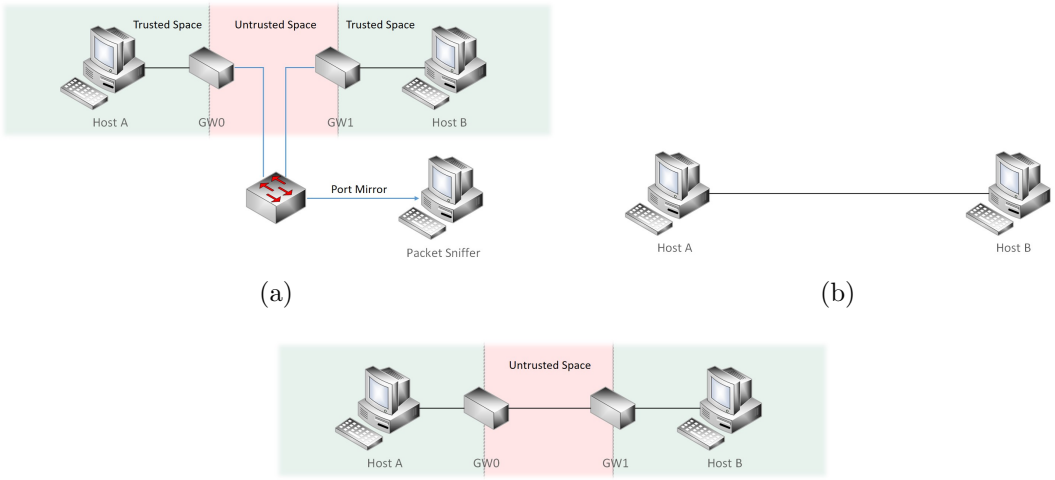


Figure 6.2: Experimental setup diagrams. (a) Correct MT6D operation is verified with port mirroring and packet capture. (b) Baseline control testing is completed with a bare wire connection. (c) For HE-MT6D testing, the same hosts were connected on the internal protected space of two HE-MT6D gateways

a switch with port mirroring enabled, which allows observation of traffic without interfering with the data stream, as seen in Fig. 6.2a. The setup in Fig. 6.2a is only for temporary verification as the port mirrored switch throttles traffic at 500 Mbps.

UDP Performance. UDP tests were conducted to test connectionless throughput, packet loss, and jitter. Tests were conducted with `iperf3` and set for a duration of 600 seconds at a target line rate speed of 1 Gbps. Two nodes were loaded into the Access List, and rotation interval for all set to 2 seconds. Tests were conducted for bare wire (control), HE-MT6D "no hits," HE-MT6D full encapsulation/decapsulation. All These tests were compared to the the performance of previous C and Python implementations and are consolidated in Fig. 6.3. The HE-MT6D "no hits" mode is used to show HE-MT6D's ability to process all packets but not perform MT6D encapsulation or decapsulation. In this mode, the Access List is

populated with dummy address, so no hits are found and all packets are classified as Case Type `PASS`. HE-MT6D full encapsulation/decapsulation has both nodes registered on the Access List and performs full MT6D encapsulation and decapsulation.

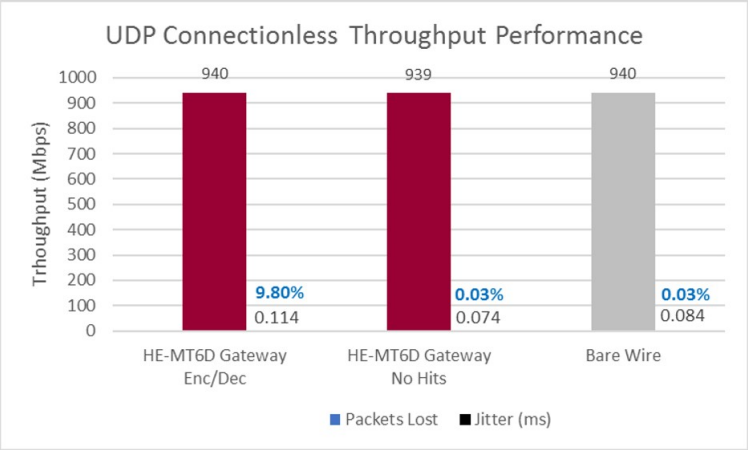


Figure 6.3: User Datagram Protocol (UDP) connectionless performance results. UDP throughput shows HE-MT6D is capable of receiving packets at high throughput. However, suffers 9.8% packet loss due to output buffer overflow.

These results, comparing HE-MT6D Gateway in "not hits" mode to bare wire performance, show that the HE-MT6D is capable of receiving packets at high throughput, and performance is exactly comparable to a bare coper Ethernet cable. Packet loss is seen at 0.03%, and jitter at 0.074 ms. However, during encapsulation and decapsulation, MT6D suffers 9.8% packet loss due to an overflow in its output buffer. This may be due to the way the final output buffer with overflow is design. The overflow of this output buffer is address in the **Buffer Overflows** section.

TCP Performance. TCP tests were conducted to test connection oriented throughput. Tests were conducted with `iperf3` and set for a duration of 6,000 seconds at a target line

rate speed of 1 Gbps, which represents a data transfer between 600-650 GB (depending upon throughput accomplished). The first five seconds were ignored to skip the TCP slowstart period. Two nodes were loaded into the Access List, and rotation interval for each set to 2 seconds. Tests were conducted for bare wire (control), HE-MT6D "not hits" mode, and HE-MT6D full encapsulation. These tests were compared to the the performance of previous C and Python implementations and are consolidated in Fig. 6.4. The command executed is `iperf3 -6 -t 6000 -i 15M -b 1G -c 2001:aaaa:bbbb:cccc::eeee:ffff:2222 -O 5 -V` .

HE-MT6D in "not hits" mode shows a transmission speed of 928 Mbps, which was exactly the same as the direct Ethernet cable connection at 928 Mbps as well. For data throughput, full encapsulation and decapsulation performed at 863 Mbps, a 1,025% improvement of connection-oriented throughput as compared to the 84.2 Mbps of C-MT6D, and 7.1% less than direct Ethernet cable connection.

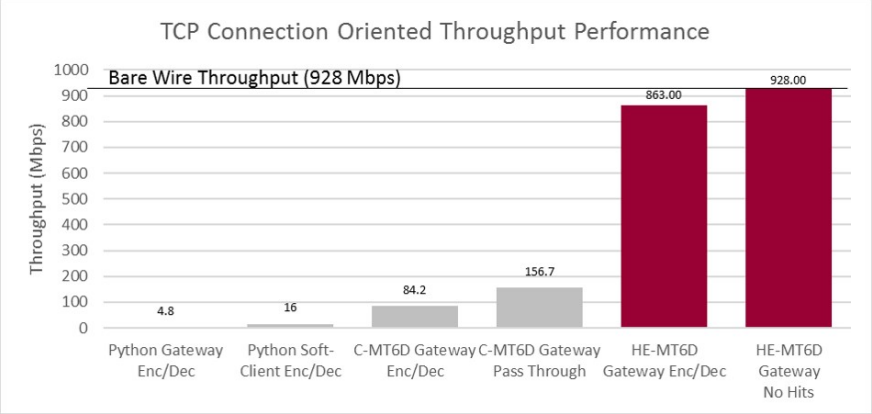


Figure 6.4: Transmission Control Protocol (TCP) connection-oriented performance results. TCP throughput tests show 1,025% improvement of connection-oriented throughput as compared to C-MT6D, 7.0% less than connection over a direct connection with bare Cat5e unshielded twisted pair (UTP) Ethernet cable.

Buffer Overflows During HE-MT6D encapsulation at 1 Gbps, packets are dropped. This is expected. Since HE-MT6D adds 16 octets to each packet, and HE-MT6D is capable of accepting packets at near line rate, expansion pushes the transmission line beyond saturation. The effect of this is seen by examining the output buffer, as shown in Fig. 6.5. This figure shows results after running UDP tests inbound from Host B to Host A, then outbound from Host A to Host B. Each test targeted 1 Gbps and is run in both outbound and inbound directions in order to observe both decapsulation and encapsulation effects at high speeds through the same gateway device. Only 60 seconds of testing is completed. This number is low but adequately shows the effects of encapsulation on the final output buffer. The command executed is `iperf3 -6 -t 60 -i 15M -b 1G -c 2001:aaaa:bbbb:cccc::eeee:ffff:2222 -V -u`.

```
Poll #221
>TSE MAC RGMII Ethernet Frames (aFramesReceivedOK, aFramesTransmittedOK)
Outbound [Eth0->Eth1] Rx:   5062574 | Tx:  4986480
Inbound  [Eth1->Eth0] Rx:   5056169 | Tx:  5056169
>Throughput:    Outbound: Mbps| Inbound: Mbps
                            0 Pps|              0 Pps
>Overflow Buffers (OVF)---------Outbound----------------------|-----------------Inbound----------------------
                  Total Seen  Overflow  MaxUsedW[Data Chan]|       Total Seen   Overflow MaxUsedW[Data
Chan]         Note: Currently set to [1024 128]
CH0 [Discard ]:           0          0                     | CH0:           0          0
CH1 [IPv6 DP ]:     5062574          0          [ 382   3] | CH1:     5056169          0         [ 382   3]
CH2 [OTHER   ]:           0          0          [   0   0] | CH2:           0          0         [   0   0]
CH3 [PASS ALL]:           0          0          [   0   0] | CH3:           0          0         [   0   0]
Final    [MUX]:     5062574      76094    |     [8191  24]  Final:   5056169          0         [ 382   3]
```

Figure 6.5: Overflow buffer statistics taken after bi-directional UDP throughput testing. Results show packet expansion from the MT6D protocol naturally causes packet loss at the Final Overflow Buffer (FOVF).

From top of the figure, the same number of Ethernet frames are received interface (5.06 million). However, the Outbound TX interface only sees 4.99 million transmitted frames. In the bottom portion of the figure, the overflow buffer statistics are shown. The final FOVF buffer as highlighted on the Outbound path shows saturation at 8191 32-bit words

(32/32 KB) comprising 24 packets; this buffer has 76,094 packets lost due to overflow. The Inbound path shows 382 words (1.5/32 KB) comprising 3 packets, with no packets lost to overflow. When the Outbound output buffer reaches saturation any other packets that are received are simply dropped in order to prevent data corruption. This saturation leads to the approximately 9.8% packet losses at high throughput as seen in the UDP throughput testing. However, 9.8% is much more than $\frac{16}{1538} \approx 1.04\%$ expansion that is taking place.

The exhibited high packet loss behavior seems to only happen on the outbound path, but some of the most cycle-intensive processes are on the inbound path. This is seen with the inbound path final buffer in Fig. 6.5 being at only $\frac{563}{8192} \approx 6.9\%$ capacity, which corroborates a hypothesis that the design of the overflow mechanism of the output buffer may be the culprit.

Further experimentation can be done to reduce lost packets. The overflow buffers used were designed to accept all data frames but invalidate packets that do not completely write onto the buffer before saturation. Simply invalidating incomplete frames prevents data corruption but wastes buffer space as the invalid packet fragment remains on the FIFO buffer until the fragments are read out. These invalid fragments occupy valuable space on the buffer and cause more packets to drop than necessary. One method remedy this flaw and conserve space is to compare the size of a packet to available space before placing it on the buffer. Doing so should greatly reduce data packet loss. Another method may be to offloading overflow into on-chip memory, but doing so seems to ultimately provide no benefit over simply increasing the overflow buffer size. The former method is preferred.

Routed packets Switched network and direct peer-to-peer connections with static addressing seems to work well. However, routed connections do not and may be due to the current handling of solicit-node multicast address (`ff02::1:ffXX:XXXX`) used for NDP with IPv6 addresses destinations outside the local subnetwork. The current packet assembly trajectory prescribed for packet case `RX_A(sn)` is perhaps simply not enough; the trajectory currently allows NS messages to pass through to a protected node, but the protected node would not respond since it does not belong to the multicast group `ff02::1ff`$\phi_{i,j(U)_{23\to 0}}$ derived from the rotation IID $\phi_{i,j(U)}$, but rather maintains membership with the multicast group `ff02::1ff` $\beta_{i_{23\to 0}}$ derived from its original IID β_i. Since the protected node ignores all of these NS solicitations, the router never receives the requisite NA acknowledgement response carrying the MAC address of the protected node. This disconnect in communication may be corrected by changing the action taken with identified `RX_A(sn)` packets. They are currently simply let `ff02::1ff`$\phi_{i,j(U)_{23\to 0}}$ through unmodified, but what really needs to happen is translation of those last 24 bits. Doing so will allow the protected node to receive and respond to the router and complete the NDP process.

Incompatible traffic It was observed that Internet websites and services (web sites, video streaming, email, etc.) were passed through without problem.

Checksum Failures During design, it seemed that the checksum process is different depending upon the IPv6 basic header next header value. So, instead of computing checksums for various types upper layer datagrams, RFC 6935 was invoked to relax the requirement for

checksums [18]. However, this relaxation only applies to UDP datagrams. Since ICMPv6 requires proper checksums, the NDP process for communication with unprotected nodes remains currently broken. Potential remedy for this problem is to reinstate checksums for all upper layer datagrams except for UDP. A checksum engine already exists but must be selectively enabled on a per-packet basis. The reason why they were decided originally not to be used was due to the complexity of providing a new checksum for nested translations that come about with ICMPv6 Type 1-4 Error packets.

6.3 Memory Search

Translations. Memory search with the HCAM memory system shows to happen at a minimum of 22 cycles per IPv6 address pair lookup. The average packet will require two address lookups, a_0 and a_1, which means that packets will usually require 44 cycles for translation lookup request. A snapshot of the Broker Module and HCAM is given in Fig. 6.6, along with a detailed explanation of the process.

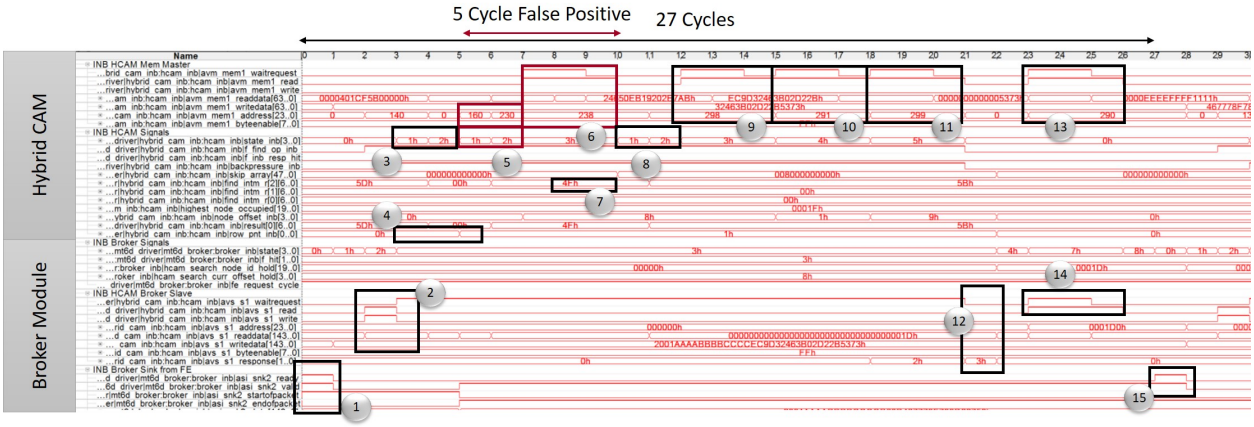


Figure 6.6: Walking through a Translation Lookup Request (TLRQ). This figure walks through the processing of a TLRQ for an inbound MT6D packet with 32 nodes loaded into the HE-MT6D gateway's Access List. Signals for the inbound Broker Module and inbound HCAM are shown. The process begins with a TLRQ received from the Field Extractor request buffer (1). The Broker waits one cycle to ensure that the the External I/O interface (Rotation Driver) is not currently requesting a memory access token. When no external I/O is pending, the Broker initiates a find operation towards the HCAM by asserting both `read` and `write` signals and writing `0x2001_AAAA_BBBB_CCCC_EC9D_3246_3B02_D22B_5373` into the HCAM Module. The Hybrid Content Addressable Memory (HCAM) responds with a back-pressure `waitrequest` signal until the search operation is complete. The HCAM begins searching its tag tables row parallel (3) (16 node tags in this example). None are found, so the internal search engine transitions to the next row (4). There is a hit seen for Node ID `29` (5). The HCAM performs a memory lookup and determines the tag-based hit to be a false positive (6). The miss is recorded in the skip array (7), and the internal search engine continues searching (8). Another hit is found, and the HCAM reads from memory to find a good match for $\phi_{2979 \to 16}$ (9). Note that the HCAM retrieves the values from memory at addresses `0x298`, `0x291`, and `0x299`. These addresses correspond with rotation $j(t) = 2$ according to node profile data structure in Table 5.2. The HCAM continues reading from memory and matches $\phi_{29143 \to 80}$(10) and finally $\phi_{2915 \to 0}$(11). The search is successful for both the IID and port, so the HCAM returns a hit response of `11` for $n_i = 0x1D$ (Node ID 29) to the Broker, and `waitrequest` is deasserted. The Broker Module in turn performs a memory lookup to find the original base IID β_i (13, 14). Finally, the Broker Module writes the original base IID onto the Translation Lookup Response (TLRS) buffer and asserts a `ready` on the Field Extractor request buffer to pop the buffer and process the next translation request.

The additional case of ICMPv6 Type 1-4 Errors would require 66 cycles due to a third address lookup for a_3. However, in these cases, the packet should be much longer than the minimum packet size of 84 octets. The Field Extractor has been designed to send the translation request for a_0 at the end of the IPv6 basic header in order to also include potential MT6D EH UDP port information for inbound decapsulation. This design decision improves efficiency and enables memory lookups to happen concurrently as the remainder of the packet sinks into the MT6D Datapath Buffer. If a packet is sufficiently long enough, the Broker Module will have completed all TLRS before the packet finishes buffering before the Packet Assembler Module.

6.3.0.0.1 Translation Lookup Buffering. There are three buffers involved during Translation Lookups. The Translation Lookup Request (TLRQ) buffer holds pending lookup requests; the TLRS buffer holds lookup responses; and the Datapath buffer holds packets while lookups are being processed. Fig. 6.7 shows the status of Datapath and Translation Lookup buffers during the bidirectional UDP test of Section 6.2. The Datapath Buffer shows 381 32-bit words (1.5 KB) representing up to 3 packets being buffered at any one time, or about 74.4% utilization. The HE-MT6D translation lookup buffers never go past 4 32-bit words (16 B), or about 6.3% utilization. None of the MT6D core buffers reach saturation. In the end, the HCAM memory system proves to conduct lookups fast enough to provide throughput at line rate speeds. Every request frame is read into the Broker Module for processing.

```
                                Current   MaxUsedW  |  Current   MaxUsedW
Buffer     (Width[Depth ])    Dat  Ch [ Dat   Ch] |  Dat  Ch [ Dat   Ch]
FE Request (144b [64  32]):     0     [   1     ] |    0     [   1     ]
Datapath   ( 32b [512 32]):   365   0 [ 381    2] |    0   0 [ 381    3]
PA Response( 80b [64  32]):     0     [   2     ] |    0     [   4     ]
```

Figure 6.7: Datapath and the Translation Lookup Request (TLRQ) and Translation Lookup Response (TLRS) buffers. The Datapath and Translation Lookup buffers presented above are from the the bidirectional UDP test of Section 6.2. The Datapath Buffer shows 381 32-bit words (1.5 KB) representing up to 3 packets being buffered at any one time, or about 74.4% utilization. The HE-MT6D translation lookup buffers never go past 4 32-bit words (16 B), or about 6.3% utilization. None of the MT6D core buffers reach saturation.

6.4 Hash Engine Performance

The Hash Engine was evaluated for providing correct digests and the speed at which digests were produced.

During initial builds, a control flow issue caused incorrect padding required for the algorithm but has since been fixed. The engine used now correctly passes benchmark value tests and produces the same results as other reference implementations of SHA256.

To test hash digest generation speed, several nodes were initialized, but the time resync command was not issued. Not issuing a resynchronization command prevents quick convergence and causes continuous hash requests. To meet timing requirements for the overall HE-MT6D design, the hash engine clock operated at half frequency (50 MHz). Still, the engine produced SHA256 digests at more than 500k per second as seen in Fig. 6.8. This volume of hash responses is more than adequate, as the system services less than 1,000 nodes.

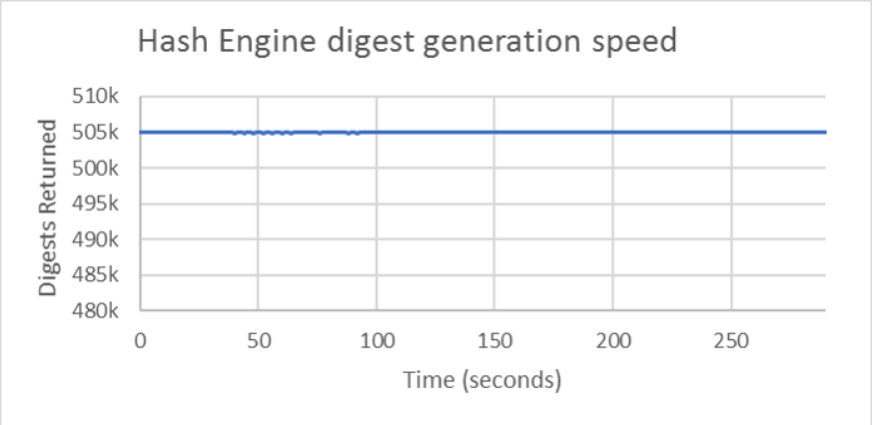


Figure 6.8: Hash engine performance. Hash responses are generated at over 500k digests per second.

6.5 FPGA Resources

Table 6.1: Field-Programmable Gate Array (FPGA) Resources consumed by HE-MT6D with both inbound and outbound HCAMs set to 7 bits of collision resistance.

Resource	2 Nodes	32 Nodes	64 Nodes	512 Nodes
Total Logic Elements	41.4k (36%)	44.2k (39 %)	48.5k (42%)	55.7k (49%)
Combinational Funcs	32.7k (29%)	35.0k (31%)	37.6k (33%)	43.0k (38%)
Dedicated Logic Regs	28.1k (25%)	29.9k (26%)	31.9k (28%)	36.1 (32%)
Total Registers	28.1k	29.9k	31.9k	36.1k
Total Memory bits	2.17 Mb (55%)	2.19 Mb (55%)	2.21 Mb (56%)	2.25 Mb (57%)
Total M9K Memory cells		286 (66%)	290 (67%)	298 (69%)

A summary of FPGA resources is presented in Table 6.1. When synthesizing varied number of supported nodes, for 512 nodes supported, the Rotation table occupied about 33k logic elements (about 1/4 the resources available on the FPGA) and just under 3k logic elements with 1 node. Resource utilization can be greatly reduced, as the Rotation Table occupies $37 * n_i$ register cells and uses the accompanied connection logic. Using register cells is not

necessary and can be implemented hard silicon on-chip memory cells. Currently, approximately 128 protected nodes can be supported with this hindrance. Rough estimation would allow support for about 300 nodes if this inefficiency were to be corrected.

6.6 MT6D Performance

Encapsulation and Decapsulation MT6D performs as expected withing a subnet. Fig. 6.9 shows addresses being properly translated, and encapsulation being performed with the original packet as the data payload. However, the same figure also shows illegal checksum rendered. Illegal checksums are not a problem according to RFC 6935 as referenced in Section 6.2. Future research may look into how other tunneling protocols handle the checksum relaxation and analyze if checksum handling is perhaps source of a side-channel attack.

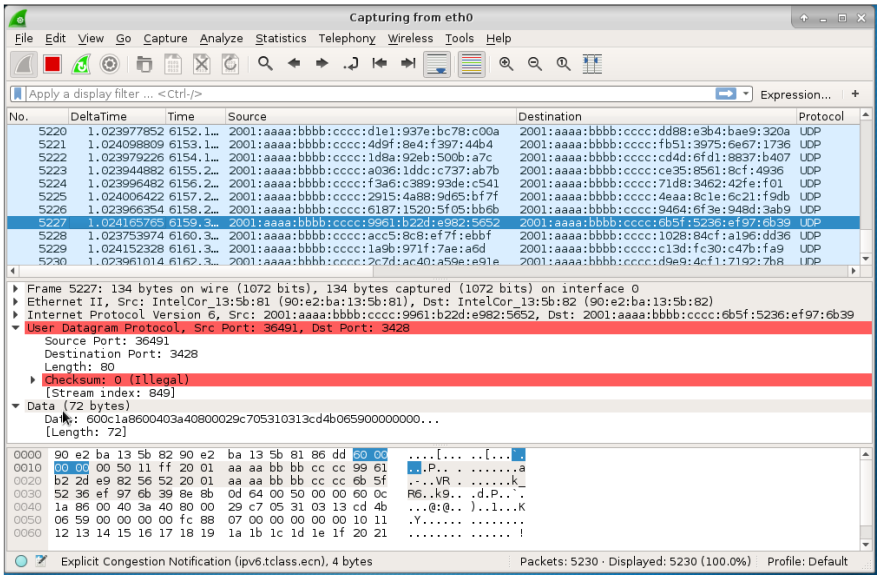


Figure 6.9: Wireshark capture of `ping` over MT6D. Packets are seen as User Datagram Protocol (UDP) tunnels.

Fig. 6.10 shows a data capture of MT6D translated IPv6 addresses intercepted via the TL-SG108E switch in port mirroring mode. The capture is from constant pings between two nodes over the course of six hours. The figure shows the approximately 21,000 rotations successfully taking place and is observed to be generally random.

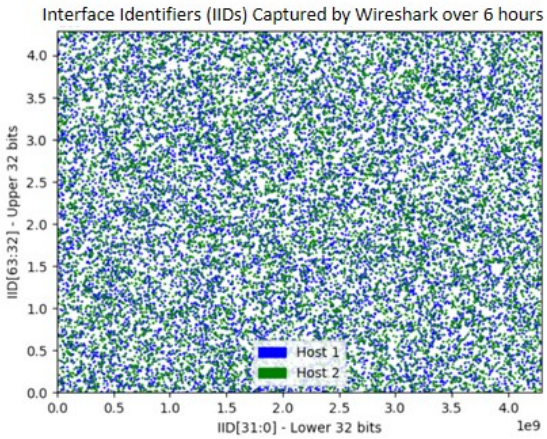


Figure 6.10: A visual representation of IPv6 addresses captured from constant pings between two nodes over HE-MT6D during a course of six hours. The x and y axes correspond to the $IID_{31\rightarrow 0}$ and $IID_{63\rightarrow 32}$ seen through the port mirrored switch setup as seen in Fig. 6.2a. The Media Access Control (MAC) address of each node was preserved in order to identify which addresses corresponded with the respective address.

Process Figures 6.11 and 6.12 show a packet being correctly identified and decapsulated. This packet is identified as a packet case type `RX_A` decapsulated accordingly.

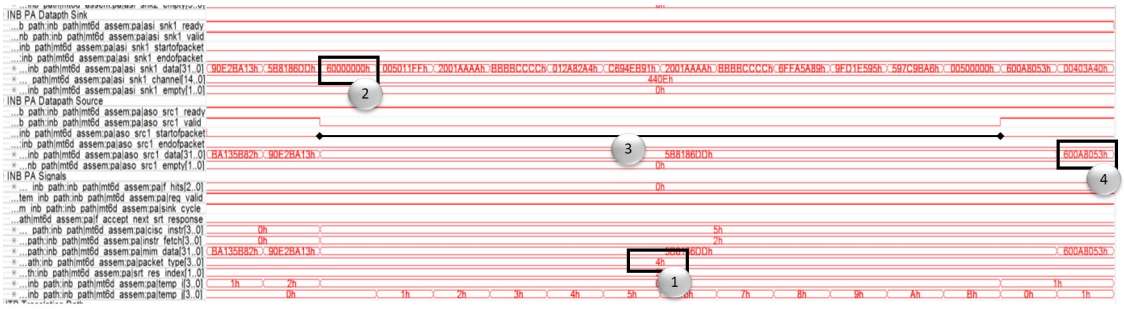


Figure 6.11: SignalTap capture of a packet being decapsulated, Part 1. (1) The Packet Assembler reconciles the packet with a case type of `0x4`, which is `RX_A`, an inbound MT6D encapsulated packet sent from a protected node to another protected node. The first IPv6 basic header word `0x60000000` is then shown at (2), which is dropped, along with the next 11 words (3). The original IPv6 basic header word `0x600A8053` is then placed on the line and transmitted instead.

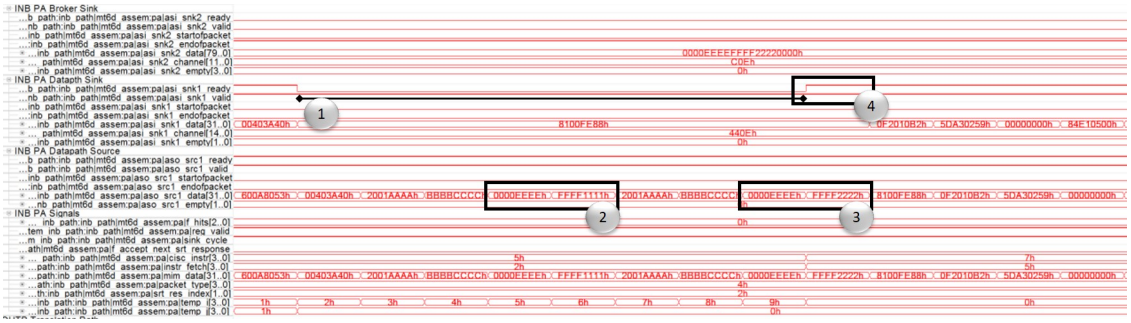


Figure 6.12: SignalTap capture of a packet being decapsulated, Part 2. The SignalTap Capture from Fig. 6.11 continues. Here, packet flow is paused (1), while transmitting the correct IPv6 addresses. In (2), the base Interface Identifier (IID) for a_0 is replaced from `0x012A82A4_C694EB91` to `0x0000EEEE_FFFF1111`. In (3), the base IID for a_1 is replaced from `0x6FFA5A89_9FD1E595` to `0x0000EEEE_FFFF2222`. The rest of the packet then passes through (4).

As an additional comment on MT6D performance, multiple Secure Shell (SSH) sessions were able to be maintained, and large files transfered. More evaluation on maintaining TCP connections should be done, but that is left to future work.

Multiple Nodes Although support for multiple nodes have been built in, performance for all nodes through the gateway has not been evaluated. Meeting the Slow 1200 mV 85 °C model timing requirements start to fail near 100 nodes supported. Timing failure for this single model does not mean the 100 nodes are not being fully maintained, but the system does not pass this specific extreme edge timing case. As mentioned, adjusting the structure of the Rotation Table within the Rotation driver may improve the number of nodes supported. A benchmark system to test all nodes also needs to be developed.

Link Local Addresses It is important to note that link local addresses have not been studied in detail. The global interface IPv6 address is what has been discussed in previous MT6D works for protecting a node. However, withHE-MT6D, it has also been observed that link local addresses will try to contact a distant end's global address at the same time and may leak that protected information. Packets generated with link local addresses do not leave the subnet, so link local address handling might not be of concern. However, it may be necessary to write in link local addresses into the Access List for protection if untrusted nodes exist on the untrusted side of HE-MT6D and are located within the same subnet.

One-second Rotation Intervals During some testing, it was noticed that one-second intervals did cause a drop in performance. Performance drops were not due at all to HE-MT6D packet processing speeds, but may be due to either the MT6D protocol itself or the output FOVF buffer overflow condition as described in Section 6.2. Future research should be done to examine the impact of maintaining one-second rotation intervals and what may be the cause of any performance drops at that resolution.

6.7 Summary and Future Work

The primary goal of HE-MT6D is to move MT6D into hardware in such a manner that supports the ultimate design of an ASIC MT6D gateway device at high throughput.

UDP performance testing reveals that the architecture and MT6D core processors perform at line rate speeds without dropping significantly more packets than bare wire speeds (0.03% packet loss for both). During encapsulation, however, the expanded packets over saturate the output buffer (which is to be expected), but more than the expected packets are dropped (9.8%). Packet losses in this case are probably due to the design of the output buffer. A possible culprit has been identified and recommended for correction in future work.

TCP performance for the MT6D processor cores show the same exact throughput as bare wire speeds (928 Mbps), but again show artifacts of buffer overflow to 863 Mbps for full encapsulation and decapsulation.

Routed packets unfortunately do not reach the router and are lost, which is most likely due

to the way solicited-node multicast addresses are handled for destinations beyond the router. The last 24 bits of this special multicast address need to be translated, and the `RX_A(sn)` packet assembly trajectory modified. Perhaps a new packet case type `TX_A(sn)` needs to be incorporated as well in future work. As purposefully intended, the CISC architecture of the HE-MT6D NSP makes this adjustment easier to incorporate as compared to having to completely rewrite the packet assembly engine.

Checksum failures do not cause failed communications, but they should be examined in future work as a possible avenue of side channel attacks, unless checksums are handled in an improved manner.

Memory performance is more than adequate, with the typical packet having two IPv6 address lookups that cost a total of 44 cycles. More research should be done to maximize the number of nodes supported, while adjusting the number of collision bits and sub-engines used in the L1 HCAM cache. Analysis on how the tag size affects collision resistance and total system performance should be conducted.

The hash engine performs more than adequately and supports over 500k hash digests a second.

For FPGA resources, moving the Rotation Table of the Rotation Driver from registers to hard silicon M9K memory cells should allow more logic cells to support a higher node count to be fit on the FPGA.

For MT6D Performance, HE-MT6D shows to perform according to design. Future work

should be done to further stress test the robustness of connections, especially with the impact of multiple nodes on the system. Although support for multiple nodes have been built in, performance for a complete system saturation stress test to support the maximum number of nodes has not been evaluated.

Other future research should include performance over routed topologies, and performance in packets per second.

6.8 Conclusion

In short summary of this chapter, in terms of meeting research objectives, most of the sub-goals are met. The system works and works well from the preliminary testing presented. HE-MT6D is indeed unobtrusive and lets non-IPv6 incompatible traffic pass through transparently. HE-MT6D performs at 1 Gbps line-rate speeds, although a problem develops when the full line is used for encapsulation and decapsulation, and performance is throttled at 863 Mbps. HE-MT6D does not use external memory to buffer packets but completely relies on a streaming architecture and native FPGA resources. HE-MT6D does support multiple nodes, but the exact number has not been extensively evaluated. And lastly, HE-MT6D is designed in full RTL.

The following chapter provides closing comments on HE-MT6D.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The work presented in this thesis contributes Network Security Processor and Hardware Engine for MT6D (HE-MT6D), the first hardware-oriented design of MT6D as a soft core Intellectual Property (IP) block developed in full Register Transfer Level (RTL) logic. The HE-MT6D IP block can be configured as an independent physical gateway device, built as embedded Application Specific Integrated Circuit (ASIC), or serve as a System on Chip (SoC) integrated submodule.

In its current state, HE-MT6D supports protection of multiple nodes and provides 1,025% throughput performance increase over earlier C-based MT6D at 863 Mbps full encapsulation and decapsulation on a direct connection among nodes, and matches bare wire throughput

performance for all other traffic. Notable contributions that enable HE-MT6D are separation of the data and control planes, the development of a nonlinear Complex Instruction Set Computer (CISC) Instruction Set Architecture (ISA) NSP for in-flight packet modification, a specialized Packet Assembly language, a configurable and a parallelized memory search through tag-based Hybrid Content Addressable Memory (HCAM) L1 write-through cache that supports a scalable number of nodes, full RTL Network Time Protocol v4 (NTPv4) hardware module to provide global time synchronization, and a modular crypto engine. Also, HE-MT6D does not rely on an third party IPv6 network communications stack, but rather binds directly with, manipulates, and adapts the IPv6 communications protocol.

With the general-purpose packet processor and modular crypto engines, HE-MT6D presents an MT6D platform for future development and maturation of the protocol, as the developed CISC ISA allows changes to the MT6D protocol specification without much further hardware modification and allows modular inter-operation with other hash functions besides the currently implemented SHA256.

7.2 Future Work

Future work for HE-MT6D involves incorporating a data encryption module, making further optimizations, and exploring new concepts that arise from its design. The data encryption module is relatively easy to insert into the stream based data path. For further optimizations, it is anticipated that by converting register based memory of the rotation driver module into

on-chip memory, more than double the nodes can be supported. Extraneous debugging circuits can also be removed. For exploring new concepts, the centralized nature of HE-MT6D introduces the concept of group rather than peer-to-peer MT6D. Group-based MT6D allows simplified management of distributed systems using a single key and profile for multiple nodes. This feature set would greatly simplifies the key management problem as presented in previous work [16, 24, 45] and can operate in tandem with the server server-client model developed in [38] but would need to be further analyzed for practical operation.

Bibliography

[1] IPv6 address. `https://commons.wikimedia.org/wiki/File:Ipv6_address.svg`, Oct. 2007.

[2] Nios II UDP Offload Example - Altera Wiki, July 2009.

[3] 802.3-2015 - IEEE Standard for Ethernet, Mar. 2016.

[4] M. Abomhara and G. M. Koien. Cyber Security and the Internet of Things: Vulnerabilities, Threats, Intruders and Attacks. *Journal of Cyber Security and Mobility*, 4(1):65–88, 2015.

[5] Altera. Avalon Interface Specifications, Dec. 2015.

[6] Altera. Nios II Gen2 Processor Reference Guide, Apr. 2015.

[7] J. Bai, W. Liju, N. Yun, Y. Liu, and Z. Zhang. A 10Gbps In-line Network Security Processor with a 32-bit Embedded CPU. pages 616–619, Chongqing, China, May 2013. IEEE.

[8] S. Bhardwaj and A. Kole. Review and study of internet of things: It's the future. In *Intelligent Control Power and Instrumentation (ICICPI), International Conference On*, pages 47–50. IEEE, 2016.

[9] I. Circuit Design. Radio Technology. `http://www.cdt21.com/resources/Modulation/modulation_SS.asp`.

[10] A. Conta and M. Gupta. RFC 4443: Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. 2006.

[11] B. Conte. Sha256, Sept. 2012.

[12] E. Davies, S. Krishnan, and P. Savola. RFC 4942: IPv6 Transition/Co-existence Security Considerations. Technical report, 2007.

[13] S. E. Deering. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification. 1998.

[14] Department of Homeland Security Science & Technology Directorate. Cyber Security Research and Development Broad Agency Announcement, Jan. 2011.

[15] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, Apr. 2006.

[16] M. Dunlop, S. Groat, W. Urbanski, R. Marchany, and J. Tront. Mt6d: A moving target ipv6 defense. In *Military Communications Conference, 2011-Milcom 2011*, pages 1321–1326, Baltimore, 2011. IEEE.

[17] M. W. Dunlop. *Achieving Security and Privacy in the Internet Protocol Version 6 Through the Use of Dynamically Obscured Addresses*. PhD thesis, Virginia Polytechnic Institute and State University, 2012.

[18] M. Eubanks, P. Chimento, and M. Westerlund. RFC 6935: IPv6 and UDP Checksums for Tunneled Packets. Technical report, 2013.

[19] H. Fadishei, M. S. Zamani, and M. Sabaei. A Novel Reconfigurable Hardware Architecture for IP Address Lookup. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, pages 81–90. ACM, 2005.

[20] C.-S. Ha, J. H. Lee, D. S. Leem, M.-S. Park, and B.-Y. Choi. ASIC design of IPSec Hardware Accelerator for Network Security. In *Advanced System Integrated Circuits 2004. Proceedings of 2004 IEEE Asia-Pacific Conference On*, pages 168–171. IEEE, 2004.

[21] R. Haas, L. Kencl, A. Kind, B. Metzler, R. Pletka, M. Waldvogel, L. Freléchoux, P. Droz, and C. Jeffries. Creating aAdvanced Functions on Network Processors: Experience and Perspectives. *IEEE network*, 17(4):46–54, 2003.

[22] W. Haixin, B. Guoqiang, and C. Hongyi. Zodiac: System Architecture Implementation for a High-performance Network Security Processor. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference On*, pages 91–96. IEEE, 2008.

[23] O. Hardman, S. Groat, R. Marchany, and J. Tront. Optimizing a network layer moving target defense for specific system architectures. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 117–118. IEEE Press, 2013.

[24] O. R. Hardman. *Optimizing a Network Layer Moving Target Defense by Translating Software from Python to C.* PhD thesis, Virginia Tech, Blacksburg, Dec. 2015.

[25] S. Hauger, T. Wild, A. Mutter, A. Kirstädter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. Packet Processing at 100 Gbps and Beyond-Challenges and Perspectives. In *Photonic Networks, 2009 ITG Symposium On*, pages 1–10. VDE, 2009.

[26] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse. Towards 100G packet processing: Challenges and technologies. *Bell Labs Technical Journal*, 14(2):57–79, Aug. 2009.

[27] R. M. Hinden and S. E. Deering. RFC 4291: IP Version 6 Addressing Architecture. 2006.

[28] R. M. Hinden, Nokia, S. E. Deering, and C. Systems. Internet Protocol Version 6 (IPv6) Addressing Architecture, Apr. 2003.

[29] T. T. Inc. DE2-115 User Manual, 2010.

[30] S. Iyer, R. Kompella, and N. McKeown. Designing Packet Buffers for Router Linecards. *IEEE/ACM Transactions on Networking*, 16(3):705–717, June 2008.

[31] W. Jiang and V. K. Prasanna. A Memory-Balanced Linear Pipeline Architecture for Trie-based IP Lookup. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium On*, pages 83–90. IEEE, 2007.

[32] M. Kabra, S. Saha, and B. Lin. Fast buffer memory with deterministic packet departures. In *High-Performance Interconnects, 14th IEEE Symposium On*, pages 67–72. IEEE, 2006.

[33] D. Lin, M. Hamdi, and J. Muppala. Distributed Packet Buffers for High-Bandwidth Switches and Routers. *IEEE Transactions on Parallel and Distributed Systems*, 23(7):1178–1192, July 2012.

[34] Y. Liu, L. Wu, Y. Niu, X. Zhang, and Z. Gao. A High-Speed SHA-1 IP Core for 10 Gbps Ethernet Security Processor. pages 237–241. IEEE, Nov. 2012.

[35] marsgod. Secure Hash Algorithm IP Core, May 2004.

[36] Marvell. 88F6281 Integrated Controller Hardware Specifications, Dec. 2008.

[37] D. Mills, U. Delaware, J. Martin, ISC, J. Burbank, W. Kasch, and JHU/APL. RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification, June 2010.

[38] C. Morrell, R. Moore, R. Marchany, and J. G. Tront. DHT Blind Rendezvous for Session Establishment in Network Layer Moving Target Defenses. In *Second ACM Workshop on Moving Target Defense*, pages 77–84. ACM Press, 2015.

[39] T. Narten, W. A. Simpson, E. Nordmark, and H. Soliman. RFC 4861: Neighbor Discovery for IP Version 6 (IPv6). 2007.

[40] National Science and Technology Council. Trustworthy Cyberspace: Strategic Plan for the Federal Cybersecurity Research and Devlopment Program, Dec. 2011.

[41] S. Nilsson and G. Karlsson. IP-address lookup using LC-tries. *IEEE Journal on selected Areas in Communications*, 17(6):1083–1092, 1999.

[42] L. M. Oliveira, J. J. Rodrigues, B. M. Mação, P. A. Nicolau, L. Wang, and L. Shu. End-to-End Connectivity IPv6 Over Wireless Sensor Networks. In *Ubiquitous and Future Networks (ICUFN), 2011 Third International Conference On*, pages 1–6. IEEE, 2011.

[43] Oracle. IPv6 Addressing Overview. `https://docs.oracle.com/cd/E18752_01/html/816-4554/ipv6-overview-10.html`, 2011.

[44] L. Polcak, L. Caldarola, A. Choukir, D. Cuda, M. Dondero, D. Ficara, B. Frankova, M. Holkovic, R. Muccifora, and A. Trifilo. High Level Policies in SDN. pages 39–57, Colmar, France, 2016. Springer International Publishing.

[45] M. G. Sherburne. *Micro-Moving Target IPv6 Defense for 6LoWPAN and the Internet of Things*. PhD thesis, Virginia Polytechnic Institute and State University, 2015.

[46] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Route Lookup. In *Network Protocols, 2005. ICNP 2005. 13th IEEE International Conference On*, pages 10–pp. IEEE, 2005.

[47] C. Systems. Cisco Router Router Architecture, 1999.

[48] S. Thomson, Cisco, T. Narten, IBM, T. Jinmei, and Toshiba. IPv6 Stateless Address Autoconfiguration, Sept. 2007.

[49] P. K. Verma, R. Verma, A. Prakash, A. Agrawal, K. Naik, R. Tripathi, M. Alsabaan, T. Khalifa, T. Abdelkader, and A. Abogharaf. Machine-to-Machine (M2M) Communications: A Survey. *Journal of Network and Computer Applications*, 66:83–105, May 2016.

[50] H. Wang, G. Bai, and H. Chen. A Gbps IPSec SSL Security Processor Design and Implementation in an FPGA Prototyping Platform. *Journal of Signal Processing Systems*, 58(3):311–324, Mar. 2010.

[51] H. Wang and B. Lin. Block-based packet buffer with deterministic packet departures. In *High Performance Switching and Routing (HPSR), 2010 International Conference On*, pages 38–43. IEEE, 2010.

Appendix A

Code Repository

Instructions for recreating this implementation, to include source code, comments, test benches, and project files are stored in the git repository of the Virginia Tech Information Technology Security Office and can be accessed at the following location:

https://git.cirt.vt.edu/he_mt6d.git

Familiarity is recommended in Verilog, SystemVerilog, Synopsys Design Constraints, ANSI C, and Tool Command Language (Tcl) Scripting.

Appendix B

Network Security Processor and Hardware Engine for MT6D (HE-MT6D) Architecture

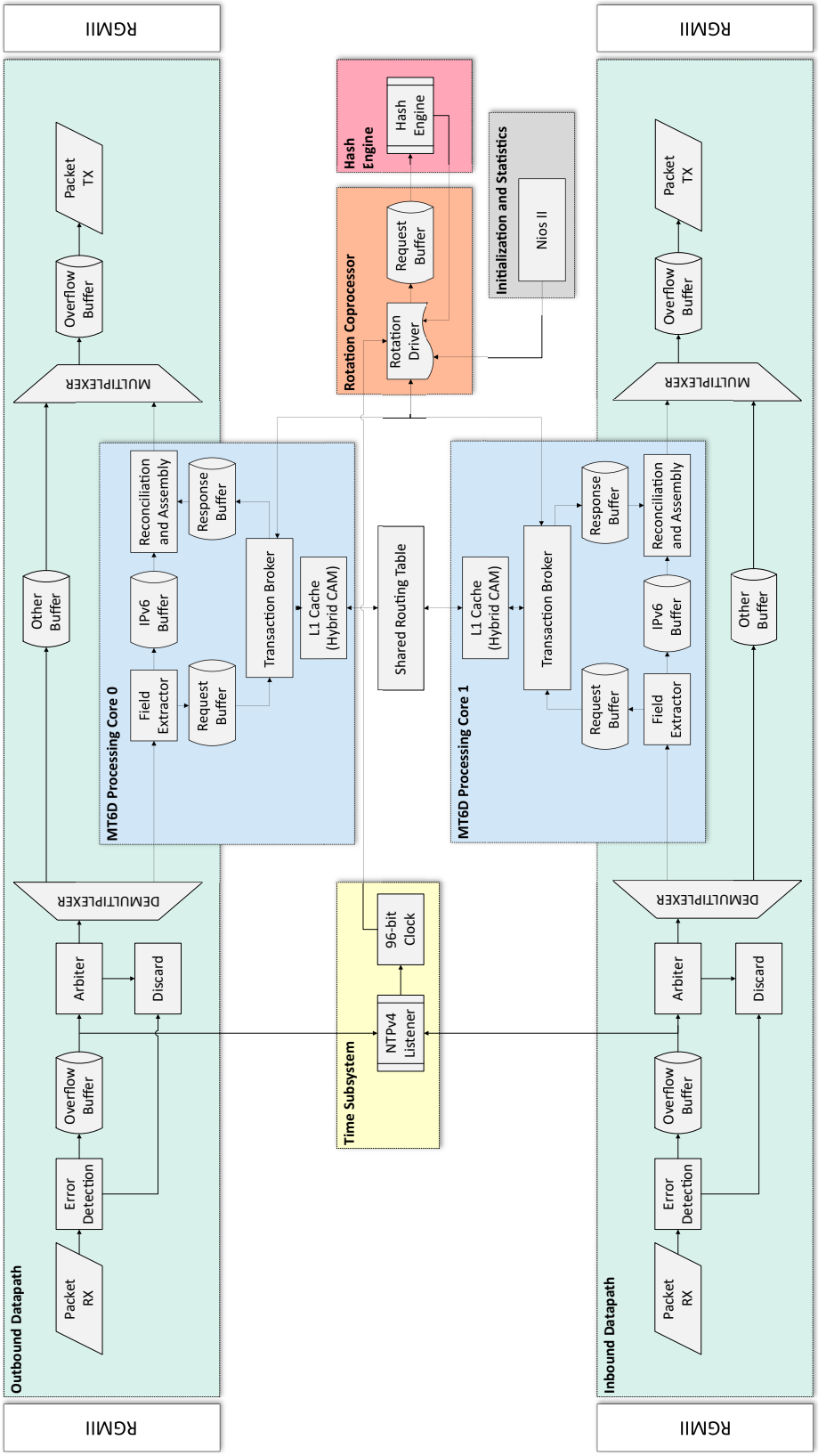


Figure B.1: The general system architecture of Network Security Processor and Hardware Engine for MT6D (HE-MT6D), with each subsystem and its comprising modules shown.