B.Comp. Dissertation

DEVELOPMENT OF SMALL SATELLITE SECURITY SYSTEM FOR DATA ENCRYPTION AND AUTHENTICATION

By

Chan Shi Yuan Galvin

Department of Computer Science
School of Computing
National University of Singapore

2023/2024

B.Comp. Dissertation

DEVELOPMENT OF SMALL SATELLITE SECURITY SYSTEM FOR DATA ENCRYPTION AND AUTHENTICATION

By

Chan Shi Yuan Galvin

Department of Computer Science
School of Computing
National University of Singapore

2023/2024

Project No: H100480
Supervisor: Professor Tan Keng Yan Colin
Co-Supervisor: Professor Soh Wee Seng

Deliverables:
Report: 1 Volume

Acknowledgments

I would like to express my deepest gratitude to all those who have contributed to the completion of this honors thesis. This journey would not have been possible without the support, guidance, and inspiration of many individuals and institutions.

First and foremost, I am profoundly grateful to my mentor, Yuan Fangxing, for his unwavering support, invaluable expertise, and patience throughout this project. His mentorship and constructive feedback played a pivotal role in shaping this work.

I also extend my sincere appreciation to Muhammed Anas s/o Tariq for his insightful comments and suggestions, which greatly enhanced the quality of this thesis.

I would like to thank my friends and fellow students for their encouragement, camaraderie, and the countless discussions that broadened my perspective. Your support was invaluable during the ups and downs of this journey.

Last but not least, I want to express my deepest gratitude to my family. Their unwavering love, encouragement, and belief in my abilities have been my greatest motivation. This achievement is as much theirs as it is mine.

To all those whose names I may have inadvertently omitted, your contributions are not forgotten, and your support is greatly appreciated.

This thesis is a testament to the collaborative efforts of many, and I am thankful for each and every one of you. Your encouragement and support have made this academic endeavor a fulfilling and enlightening experience.

Thank you.

Abstract

Development of small satellite security system for data encryption and authentication

by

Chan Shi Yuan Galvin

Bachelor of Computing in Computer Science

National University of Singapore

This dissertation investigates the implementation of hardware acceleration techniques for Authenticated Encryption Schemes on Field-Programmable Gate Array System-on-Chip (FPGA SoC) platforms, focusing on their application in satellite communications. The main goal of this project is to explore the integration of an existing Authenticated Encryption module with software.

The growing complexity and volume of data exchanged in satellite networks necessitate efficient and secure encryption mechanisms to protect sensitive information from unauthorized access and tampering, integrating the inherent parallel processing capabilities of FPGAs with the capabilities of a traditional SoC.

The research begins with a comprehensive review of existing Authenticated Encryption algorithms, FPGA SoC architectures, and development process. Subsequently, novel hardware designs and optimizations are proposed and implemented to accelerate Authenticated Encryption.

The implementations in this research contribute to advancing the state-of-the-art in hardware-accelerated encryption for satellite communications, offering a scalable and reliable solution for securing data transmissions in space-based networks. The developed FPGA-based Authenticated Encryption Schemes acceleration modules have the potential to be integrated into next-generation satellite platforms, ensuring robust security and performance in mission-critical applications. The insights gained from this dissertation pave the way for further research and development in hardware-accelerated encryption schemes tailored for satellite communication systems.

Contents

Chapter 1

Introduction

In an era marked by the exponential growth of space technology and satellite-based applications, secure and efficient communication between satellites and ground stations has emerged as a critical requirement. These systems, responsible for the exchange of sensitive data and mission-critical commands, demand robust security measures to ensure the confidentiality, integrity, and authenticity of the transmitted information. To meet these security challenges, hardware acceleration of authenticated encryption schemes has become a vital research and development focus.

1.1 Problem Statement

The central element in this endeavor is the utilization of Field-Programmable Gate Arrays (FPGAs) integrated into System-on-Chip (SoC) architectures, as offered by platforms like Xilinx[13]. FPGA-SoCs provide a unique blend of programmability and high-performance hardware acceleration capabilities, making them an ideal choice for implementing cryptographic algorithms that are essential for securing satellite communications.

This project is driven by the need to address the growing security concerns in satellite communication systems by proposing a novel approach: the integration of hardware acceleration with authenticated encryption schemes on Xilinx FPGA-SoCs. Authenticated encryption schemes, which encompass both encryption and data authentication, offer a holistic solution to safeguard against eavesdropping, tampering, and unauthorized access.

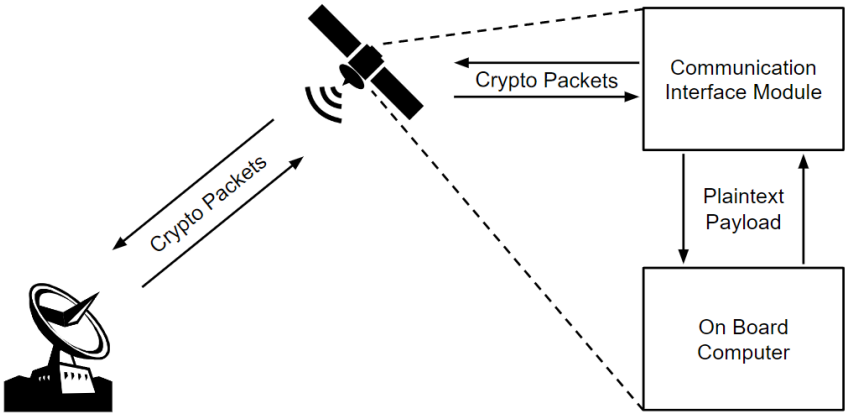

Figure 1.1: Overview of Satellite System

1.2 Overview

The key objective of the project is to develop a proof-of-concept for a security module in the Communication Interface Module (CIM) on the satellite that provides bidirectional secured communication between the ground station and the satellite, as illustrated in Figure 1.1. For downlink communication, the CIM is responsible for receiving plaintext payload data from the On Board Computer, encrypting the payload and encapsulating in a Crypto Packet, to be sent to the ground station. For uplink communication, CIM is responsible for receiving a Crypto Packet, extracting the encrypted payload and decrypting it, thereafter sending the plaintext payload to the On Board Computer.

The CIM is to be developed on an FPGA-SoC, utilizing a cryptographic module within the FPGA fabric for hardware acceleration of the authenticated encryption scheme.

1.3 Thesis Synopsis

The rest of this thesis is organized as follows. In Chapter 2, we conduct a literature review, briefly explaining pre-requisite concepts used in this project as well as various background studies done during the course of the year. Chapter 3 specifies the requirements for this project. Chapter 4 highlights work done to implement a security handler and design considerations made along the way, as well as various testings done to ensure correctness. We conclude the entire thesis as well as discuss

further directions for future works in Chapter 5.

Chapter 2

Literature Review

2.1 Cryptography

In view of a mature space environment with information security requirements for space telecommunication, advanced secured mechanisms will be embedded in the communication link between Earth's command centre and satellite. The next-generation satellites are equipped with modernised encryption scheme with complex authentication processes. Below describes briefly an overview of the security concepts that progressively builds towards the authenticated encryption scheme that is utilised in this project.

2.1.1 Advance Encryption Standard

The Advanced Encryption Standard (AES) [11] is a widely adopted and highly respected block cipher encryption algorithm that plays a crucial role in securing data in various applications, from communication and data storage to network security and more. AES is characterized by its robust security, computational efficiency, and versatility, making it a cornerstone of modern cryptography. Below describes some key feature of AES:

- Block Size: AES is a block cipher encryption algorithm with a block size of 128 bits. AES receives as input 128 bits of plaintext and produces 128 bits of ciphertext. The encryption is done through multiple rounds of substitution, permutation, and mixing performed on this fixed-size block.
- Symmetric Key Encryption: AES is renowned for its robust security. It employs symmetric key encryption, which means the same key is used for both

encryption and decryption.

- Key Lengths: AES supports different key lengths, including 128, 192, and 256 bits, with longer keys providing higher encryption complexity. The key is expanded into multiple round keys, with longer keys expanding into more round keys and consequently requiring more rounds of transformation on the block, incurring a slight time penalty. The choice of key length depends on the specific security requirements of the application.

2.1.2 Modes of Operations

Modes of operation [7], in the context of cryptography, are techniques used to apply block ciphers, such as AES, to encrypt or decrypt data that is larger than a single block. These modes introduce additional processes and strategies to ensure the confidentiality, integrity, and security of data.

Electronic Codebook

The Electronic Codebook (ECB) mode, shown in Figure 2.1, divides the plaintext into fixed-size blocks and encrypts each block separately with the same encryption key. While it's simple and parallelizable, ECB doesn't provide adequate security, as identical plaintext blocks yield identical ciphertext blocks.

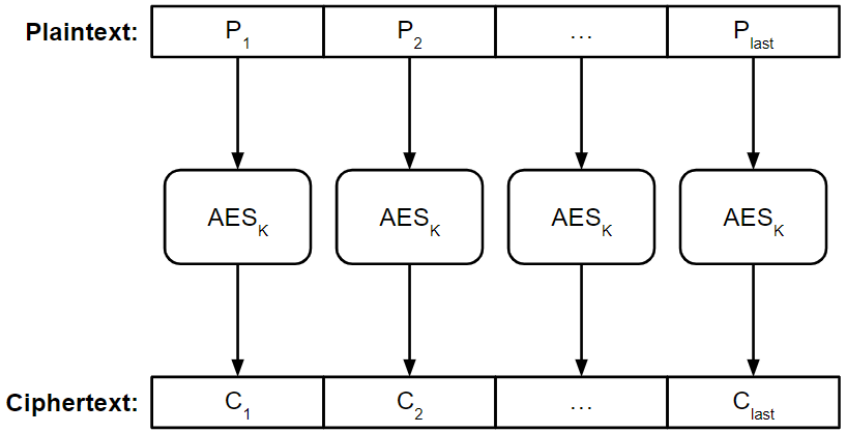

Figure 2.1: Electronic Codebook

Counter Mode

Counter mode (CTR), shown in Figure 2.2, turns a block cipher into a stream cipher by encrypting the Counter Blocks (CBs) to generate a pseudorandom keystream. CTR takes as input an Initialization Vector (IV) used to generate the first CB, and increments the CB repeatedly. The encrypted CB is then XOR-ed with the plaintext to produce ciphertext. CTR solves the issue of ECB - the CBs are always incremented, the encrypted outputs is always unique, identical plaintext blocks XOR-ed with different encrypted CBs will yield different ciphertext.

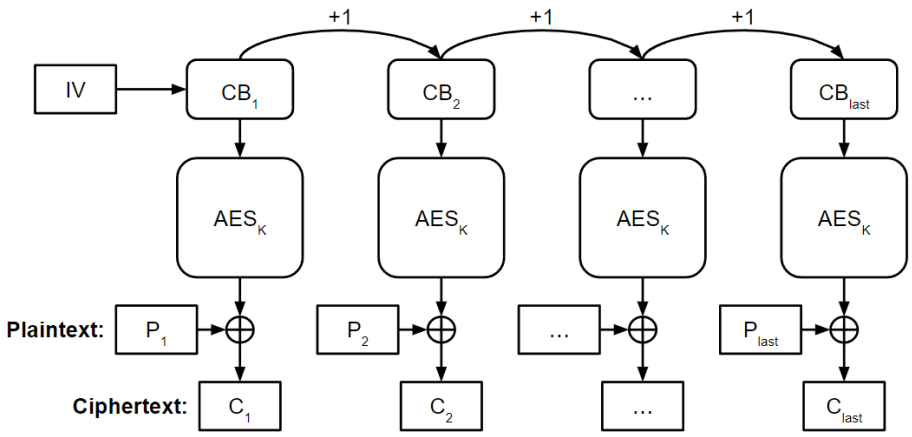

Figure 2.2: Counter Mode

Galois Counter Mode

Galois Counter Mode (GCM), shown in Figure 2.3 uses CTR to encrypt a variable-sized plaintext into ciphertext. GCM then performs an additional step of hashing the ciphertext, as well as Additional Authenticated Data (AAD), to produce a Tag used as the Message Authentication Code (MAC). The hashing function makes use of the same key that is used by the encryption algorthm in CTR, hence ensuring the Tag is generated by an authorised party. The sender then sends the ciphertext, IV, AAD and Tag, When the receiver receives the ciphertext, IV and AAD, the receiver can generate another Tag, then compare the generated tag with the received tag to verify that the data received has not been tampered with.

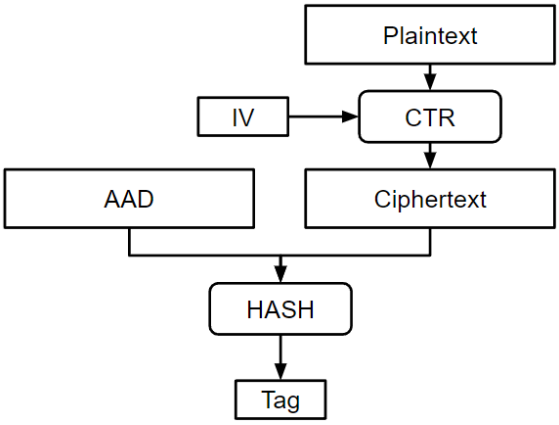

Figure 2.3: Galois Counter Mode

2.2 FPGA-SoC

A Field-Programmable Gate Array System on Chip (FPGA-SoC) is a type of integrated circuit that combines the flexibility of FPGAs with the processing power and capabilities of a traditional SoC. FPGA-SoCs are designed to provide a combination of reconfigurable hardware and software programmability in a single package. They are particularly useful in applications where both hardware acceleration and general-purpose processing are required. This project uses the Zynq 7000 SoC, that combines a dual-core ARM Cortex-A9 processor with traditional FPGA logic fabric. Below briefly describes the platform.

2.2.1 Xilinx Zynq Architecture

The high level model the Zynq architecture is shown in Figure 2.4. The Zynq comprises two main parts: a Processing System (PS) formed around a dual-core ARM Cortex-A9 processor, and Programmable Logic (PL), which is equivalent to that of an FPGA. The PL section is ideal for implementing high-speed logic, arithmetic and data flow subsystems, while the PS supports software routines and/or operating systems, meaning that the overall functionality of any designed system can be appropriately partitioned between hardware and software. Links between the PL and PS are made using industry standard Advanced eXtensible Interface (AXI) connections[6].

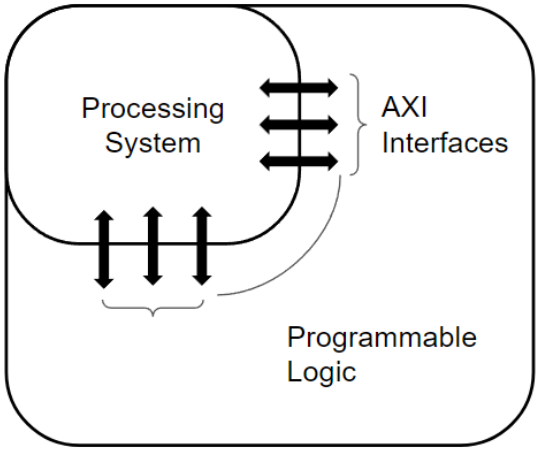

Figure 2.4: High Level Model of Zynq Architecture

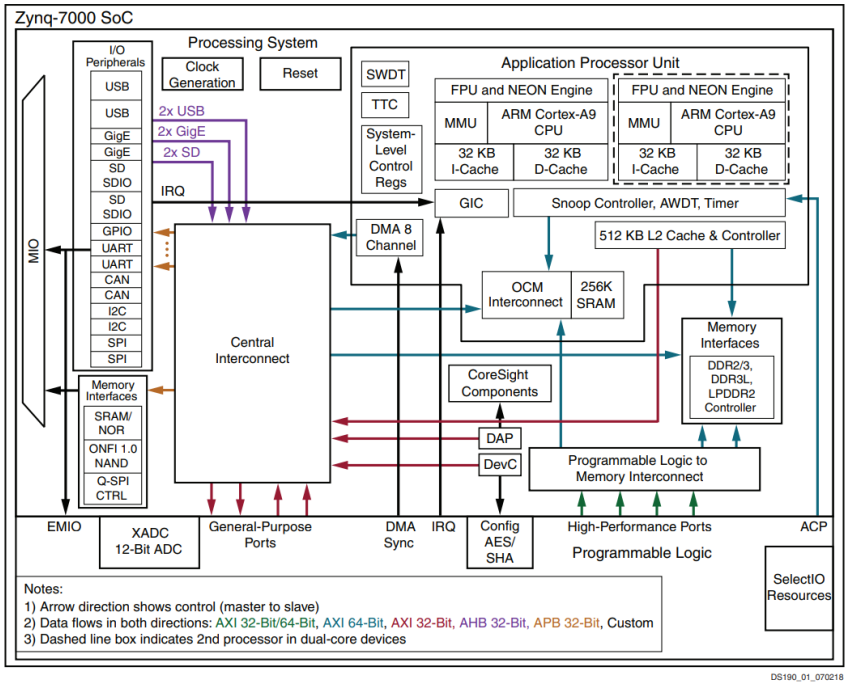

Figure 2.5: Zynq Processing System Architecture

2.3 Zynq Processing System

The Zynq Processing System encompasses not just the ARM processor, but a set of associated processing resources forming an Application Processing Unit (APU), and further peripheral interfaces, cache memory, memory interfaces, interconnect,

and clock generation circuitry[13]. A block diagram showing the architecture of the PS is shown in Figure 2.5.

2.3.1 The AXI Standard

The appeal of Zynq lies not just in the properties of its constituent parts, the PS and the PL, but in the ability to use them in tandem to form complete, integrated systems. The key enabler in this regard is the set of highly specified AXI interconnects and interfaces forming the bridge between the two parts.

There are three flavours of AXI4, each of which represents a different bus protocol, as summarised below. The choice of AXI bus protocol for a particular connection depends on the desired properties of that connection.

- AXI4[2]: For memory-mapped links, and providing the highest performance: an address is supplied followed by a data burst transfer of up to 256 data words
- AXI4-Lite[2]: A simplified link supporting only one data transfer per connection (no bursts). AXI4-Lite is also memory-mapped: in this case an address and single data word are transferred.
- AXI4-Stream[3]: For high-speed streaming data, supporting burst transfers of unrestricted size. There is no address mechanism; this bus type is best suited to direct data flow between source and destination (non memory mapped).

2.3.2 AXI4-Stream Protocol

AXI4-Stream is a protocol designed for transporting arbitrary unidirectional data from master to slave as shown in Figure 2.6. The detailed description of necessary signals are given in Table 2.1. In an AXI4-Stream, TDATA width of bits is transferred per clock cycle. The transfer is started once the master sends the TVALID signal and the slave responds by sending the TREADY signal (once it has consumed the initial TDATA). At this point, the master will start sending TDATA and TLAST. TLAST signals the last byte of the packet. So the slave keeps consuming the incoming TDATA until TLAST is asserted[12].

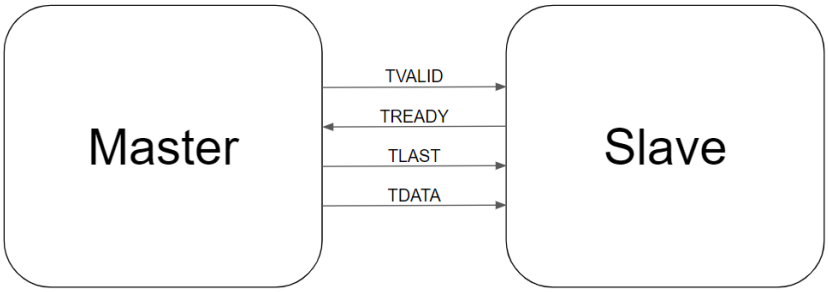

Figure 2.6: AXI4-Stream Overview

Signal	Source	Description
ACLK	Clock source	The global clock signal. All signals are sampled on the rising edge of ACLK.
ARESETn	Reset source	The global reset signal. ARESETn is active-LOW.
TVALID	Master	TVALID indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted.
TREADY	Slave	TREADY indicates that the slave can accept a transfer in the current cycle.
TDATA[(8n-1):0]	Master	TDATA is the primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.
TLAST	Master	TLAST indicates the boundary of a packet.

Table 2.1: AXI4-Stream Signals

Figure 2.7 shows the detailed interaction between master and slave, showing the handshake process using the TVALID and TREADY signals. The master asserts TVALID as it puts valid data on TDATA bus. When the slave is ready to receive data, the slave will assert TREADY. Data transfer only occurs on the positive edge of ACLK when both TVALID and TREADY are asserted.

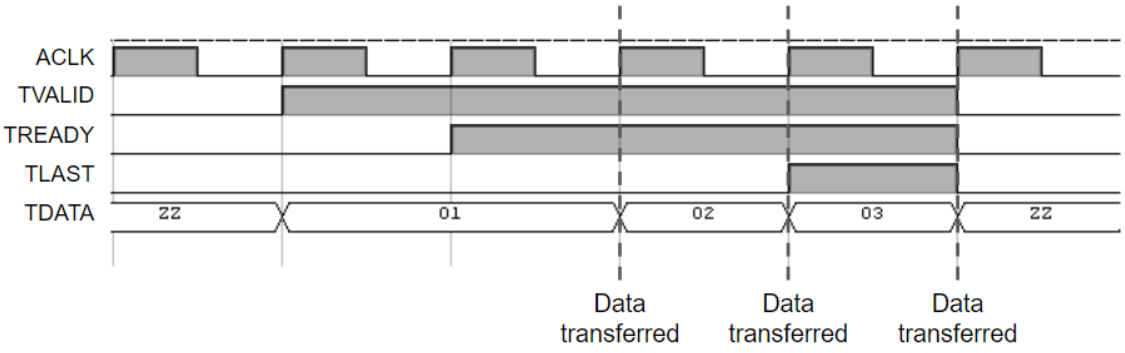

Figure 2.7: AXI4-Stream Waveform

2.3.3 Development Process for Zynq 7000

The first stage is to define the desired behaviours of the system, i.e. to create an appropriate specification from a set of requirements. This is depicted as the starting point at the top of Figure 2.8, and it forms the basis of the system design that is subsequently developed[6].

The next step is to partition functionalities into hardware or software. Hardware functionalities are implemented in the PL, while software functionalities are executed on the PS. Once the system has been partitioned, hardware development and software development are done in parallel with frequent integration testing to ensure correctness of the whole system.

Hardware development is done in Vivado Design Suite, where the task is to design Intellectual Property (IP), written in Hardware Description Languages such as Verilog. These IP are then assembled as blocks in a block design, making appropriate connections between the blocks.

Software development is done in Vitis Integrated Development Environment (IDE) through developing custom code or by reusing pre-existing software. Bare Metal Applications (software running directly on hardware without an Operating System) can be built using Board Support Packages which provide a set of low-level drivers and functions to interact with hardware developed.

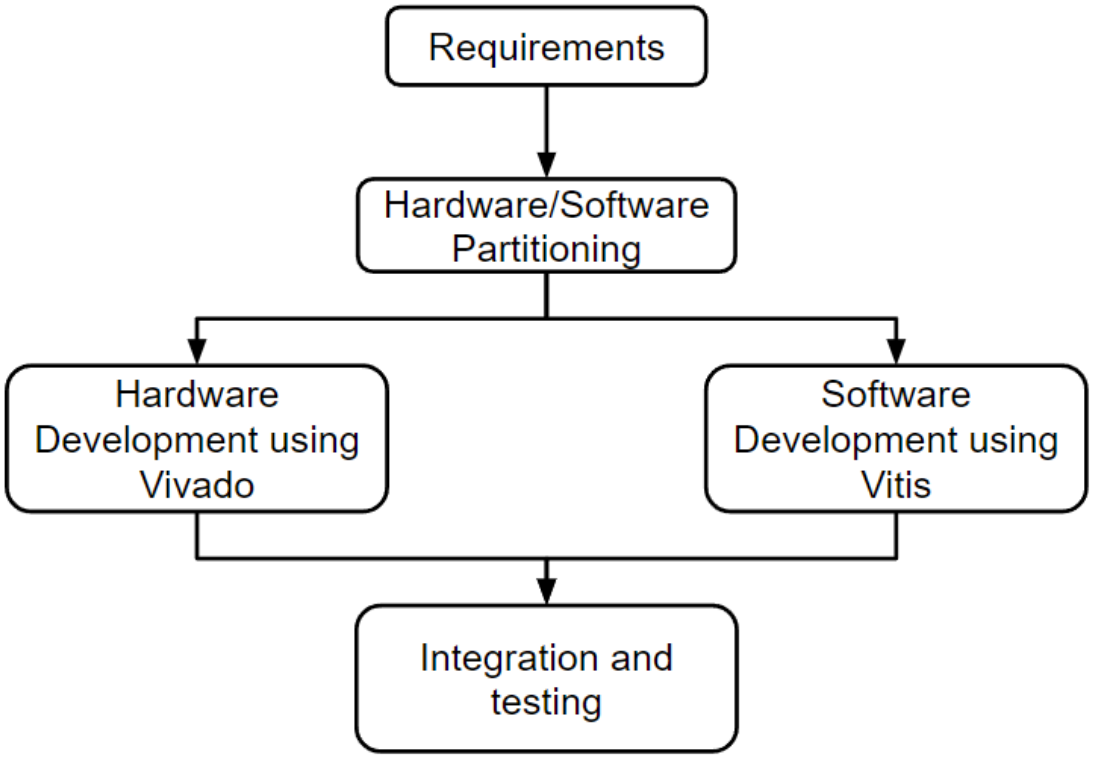

Figure 2.8: Development Process for Zynq 7000

2.4 Controller Area Network Bus Protocol

Controller Area Network (CAN) is a serial communication protocol, which supports distributed real-time control and multiplexing for use within road vehicles and other control applications [8].

The CAN bus consists of a network of nodes (devices) connected by a two-wire bus. These wires are CAN High (CANH) and CAN Low (CANL), forming a differential pair that provides noise immunity and allows for reliable communication over long distances.

The two signals, CANH and CANL are either driven to a "dominant" state with CANH $>$ CANL, or not driven and pulled by passive resistors to a "recessive" state with CANH $\leq$ CANL. A 0 data bit encodes a dominant state, while a 1 data bit encodes a recessive state, supporting a wired-AND convention.

CAN is a message-based protocol where each node on the network communicate using messages called frames, with priority-based arbitration mechanism. There are

two different frame formats - base frame format and extended frame format. The only difference between the two formats is that the CAN base frame supports a length of 11 bits for the identifier, and the CAN extended frame supports a length of 29 bits for the identifier, made up of the 11-bit identifier (base identifier) and an 18-bit extension (identifier extension). Table 2.2 describes the various fields of a CAN frame.

Field name	Length (bits)	Description
Identifier	11/29	Unique identifier of message used for arbitration, lower value has higher priority
Remote Transmission Request	1	0 for data frames, 1 for remote request frames
Identifier Extension bit	1	0 for 11 bit identifier, 1 for 29 bit identifier
Data Length Code	4	Number of bytes of data (0 - 8 bytes)
Data	0 - 64	Data

Table 2.2: CAN Frame

Chapter 3

Project Requirements

3.1 CAN Bus

Communication between different components of the satellite is done using the CAN Bus protocol. A simplified architecture is shown in Figure 3.1. The Communication Interface Module (CIM) must be able to receive plaintext data from the On Board Computer, encrypt it and encapsulate in a Crypto Packet, to be transmitted to the Radio Transceiver for communication towards the Ground station. Conversely, the CIM must be able to receive Crypto Packets from the Radio Transceiver, decrypt the payload and verify authenticity and data integrity, then transmit the plaintext payload to the On Board Computer.

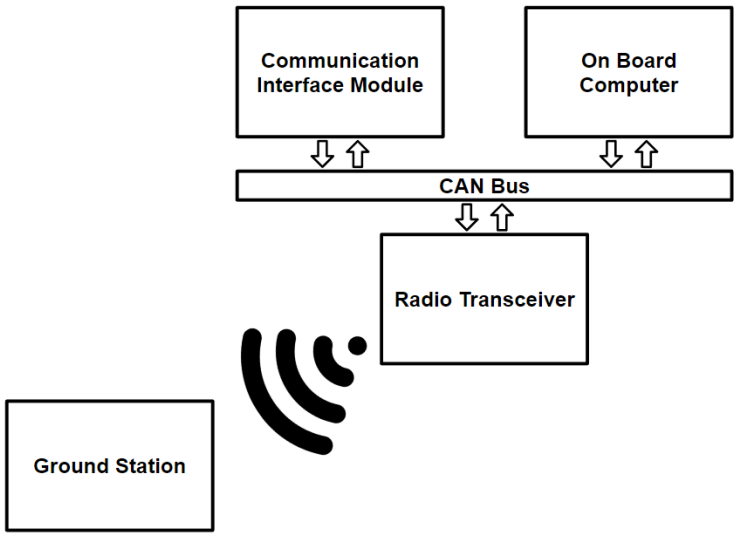

Figure 3.1: Satellite Overview

3.2 Crypto Packet

Communication between satellite and ground station will be using crypto packets following the format specified in Table 3.1.

Field	Length (bits)	Description
Crypto Header	96	Refer to Table 3.2
Payload	Variable	This is the encrypted payload
MAC	128	This is the Message Authentication Code computed over the header and the payload, also known as authentication tag generated by GCM

Table 3.1: Crypto Packet Structure

Field	Length (bits)	Remarks
Source ID	4	Refer to Table 3.3
Destination ID	4	Refer to Table 3.3
Key Index	8	Identifies which key in the keybank to use
Payload Length	16	Length of the crypto packet payload in bytes (excludes crypto header and authentication tag)
Reset Replay Counter	16	This 16 bits counter help to prevent accidental overriding of the whole replay counter. (Note: The reset replay counter and replay counter formed the whole 64 bits replay counter)
Replay Counter	48	This is the lower 48 bits value of the whole replay counter. (Note: The reset replay counter and replay counter formed the whole 64 bits replay counter)

Table 3.2: Crypto Header

Name	ID
Satellite 1 (SAT1)	1
Satellite 2 (SAT2)	2
Satellite 3 (SAT3)	3
Master Ground Station	1

Table 3.3: ID of satellite/ground station

3.3 Authenticated Encryption Scheme

AES-256 GCM is should be used for the authenticated encryption scheme as reccomended by Consultative Committee for Space Data Systems (CCSDS)[5]. A cryptographic module will be given to perform the AES-256 GCM, shown in Figure 3.2. This project requires a custom Crypto IP to be be implemented in the PL that interfaces with the given AES-256 GCM module in order to allow efficient data transfer between PS and PL. The CIM must encrypt an input string which consists of 1 block (128 bit) of AAD (Crypto Header) followed by one or more blocks of data. A 128 bit Message Authentication Code (MAC) is computed over the AAD and Payload data. Table 3.4 specifies the inputs and outputs to the AES-256 GCM module.

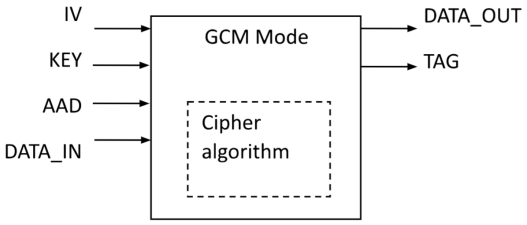

Figure 3.2: AES-256 GCM Module

Field	Length (bits)	Description
IV	96	Crypto Header
KEY	256	Key for AES256
AAD	128	0x00000000 ‖ Crypto Header
DATA_IN	Variable	Payload
DATA_OUT	Variable	Encrypted Payload
TAG	128	Message Authentication Code computed over AAD ‖ Payload

Table 3.4: IP Inputs and Outputs

3.3.1 Key Loading

Ground software will transfer 1 set of uplink keys and 1 set of downlink keys to CIM via CAN bus. Refer to Table 3.5 for the packet structure for each set of keys. Each set of keys contains 64 key entries, each key entry contains a 256 bit key used for AES-256 encryption, shown in Table 3.6. The CIM must be able to read the set of keys via CAN bus, extract the keys and store into non-volatile memory.

Field	Length (bytes)	Value	Description
Payload Hash	32	-	SHA256 hash computed over payload (Key Entries)
# of items	2	0x0001	Only 1 item
Tag	2	0x0001	0x01 for keys
Key Entries	3200	Refer to 3.6	64 keys, 50 bytes each

Table 3.5: Key Loading Packet

Field	Length (bytes)	Description
Index	2	Index of this entry within the entries
Key	32	AES256 uses 256 bits key length
IV	16	Not used

Table 3.6: Key Entry

Chapter 4

Implementations

4.1 Hardware Development

The Communication Interface Module is implemented on a Xilinx ZC702 Evaluation Board which features the Zynq 7000 FPGA-SoC. This section focuses on the hardware modules developed in this project and the hardware architecture of the implemented system.

4.1.1 Authenticated Encryption Scheme

The primary goal of this project is to explore the integration of software with a cryptographic module implemented within the FPGA fabric. Initially, the plan involved utilizing the AES-256 GCM module developed by Research Associate Yuan Fangxing. However, due to compatibility issues arising from the module's development on a different platform, we encountered significant challenges in integrating it into the project's platform. Following discussions with STAR Centre research associates, we opted to proceed with an open-source AES GCM module available online.

OpenCores is a community portal for professionals, amateurs, and enthusiasts interested in the field of digital design engineering. The site gives users open access to view, download, reuse, and share gateware designs. OpenCores specializes on bundles of structured files forming self-confined units, most commonly known as Intellectual Properties (IP) "cores", coded in Hardware Description Language (HDL).

An implementation of AES GCM, authored by Tariq Bashir Ahmad, was found

on the OpenCores portal. This implementation aims to explore hardware implementation of AES GCM mode of operation specifically targeting FPGA [1]. Verilog code was provided for the Encrypt and Authenticate Block with interfaces shown in Figure 4.1.

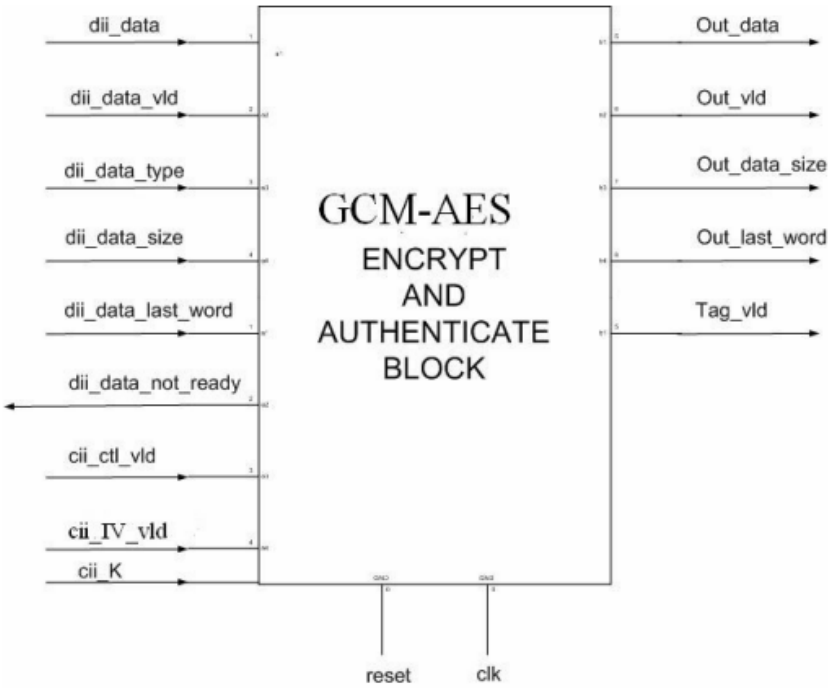

Figure 4.1: Interface diagram Encrypt and Authenticate Block

However, this implementation of AES GCM came with several limitations:

- Key size used for this implementation is 128 bits long
- Key expansion is done for each frame, regardless if the same key is used for multiple frames
- Only the Encrypt and Authenticate Block is provided, the Decrypt and Verify Block is not published on the portal
- This implementation assumes 96 bits Initialization Vector
- This implementation assumes at least one block of Additional Authenticated Data

After further discussion with the research associates at STAR Centre, we decided to move forward with this implementation as the goal of this project was to explore the integration of software with a cryptographic module implemented within the FPGA fabric, instead of optimizing the implementation of the cryptographic module. The limitations of this implementation still fits within the requirements and is still suitable to be used for a proof-of-concept.

In order to create the Decrypt and Verify Block, modifications were made to the internal Finite State Machine of the Encrypt and Authenticate Block provided. In the Encrypt and Authenticate Block, when 128 bits of plaintext data is read, the block first enters an Encrypt state to perform the AES encryption of the counter block, then XORs the encrypted output with the plaintext data to produce ciphertext. After the Encrypt state, the block then transitions into a Hash state to compute the hash over the ciphertext.

The Decrypt and Verify Block is developed by making a copy of the Encrypt and Authenticate Block and modifying the state transitions. In the Decrypt and Verify Block, when 128 bits of ciphertext data is read, the block first enters the Hash state to compute the hash over the ciphertext. After the Hash state, the block then transitions into the Encrypt state to perform the AES encryption of the counter block, then XORs the encrypted output with the ciphertext data to produce the original plaintext.

The interfaces of the Decrypt and Verify Block is the same as the Encrypt and Authenticate Block shown in Figure 4.1.

4.1.2 Crypto IP

Both the Encrypt and Authenticate Block and the Decrypt and Verify Block makes use of a custom interface to stream the key, IV, AAD and data blocks into the module. A wrapper module is developed to accept inputs from the PS over the AXI4 Stream interface, and to route the inputs to the appropriate interface for the Encrypt and Authenticate Block or the Decrypt and Verify Block. The wrapper module then reads the outputs of the Encrypt and Authenticate Block or the Decrypt and Verify Block and streams them to the PS over the AXI4 Stream interface.

The Crypto IP written in Verilog is instantiated in the PL. The Crypto IP has two AXI4 Stream interfaces: one serves as a slave interface to receive inputs, while the other functions as a master interface to transmit outputs. An illustration detailing the inputs and outputs can be found in Figure 4.2. Given that the Crypto IP has only a single master and slave interface, a state machine has been employed to manage the reading and writing of multiple inputs and outputs. The state machine for the Crypto IP is shown in Figure 4.3. Detailed description of individual states and the transition between states is documented in Table 4.1.

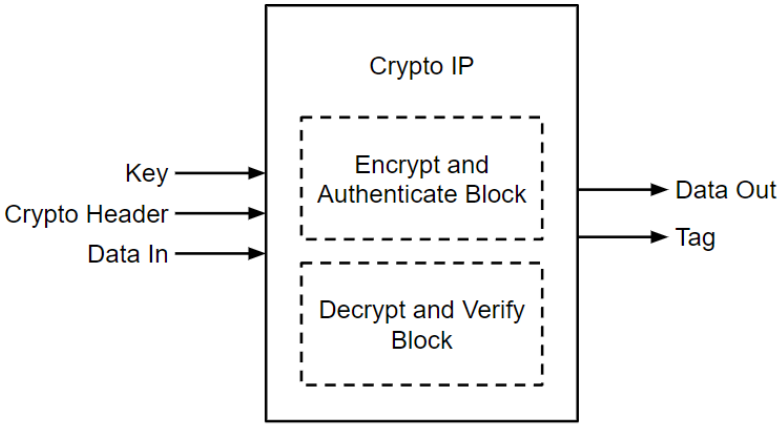

Figure 4.2: Overview for Crypto IP

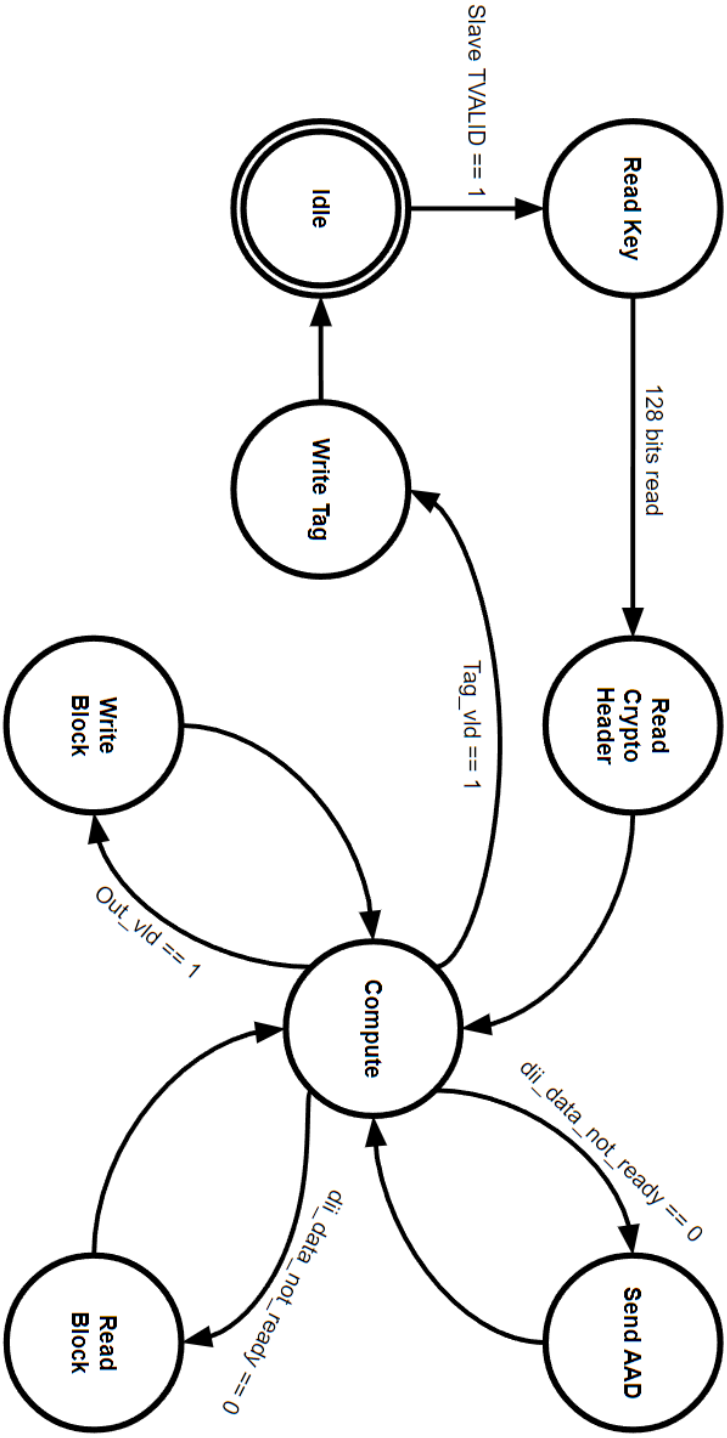

Figure 4.3: State Machine for Crypto IP

State	Description	State Transition
Idle	This is the starting state, sets all internal registers to default values to prepare for inputs	When TVALID on slave interface is asserted, transit to Read Key state.
Read Key	Asserts TREADY on slave interface to allow data tranfer. Reads 128 bits Key from TDATA.	When finished reading 128 bits, transit to Read Crypto Header state
Read Crypto Header	Reads 128 bits from TDATA. Extracts 96 bits Crypto Header to act as Initialization Vector. The remaining 32 bits is used to indicate Encrypt or Decrypt. If Encrypt, inputs are routed to the Encrypt and Authenticate Block, else the inputs are routed to the Decrypt and Verify Block.	When finished reading 128 bits, transit to Compute state. Set next state to be Send AAD state
Send AAD	Pad Crypto Header with zeroes to form AAD and send AAD into the AES GCM module	When finished sending AAD, transit to Compute state. Set next state to be Read Block state
Read Block	Asserts TREADY on slave interface to allow data tranfer. Reads 128 bits data block from TDATA and sends data into the AES GCM module.	When finished reading 128 bits, transit to Compute state. Set next state to be Write Block state
Write Block	Asserts TVALID on master interface. Writes output of AES GCM module to TDATA on master interface.	When finished writing 128 bits, transit to Compute state. If TLAST on slave interface is asserted, set next state to Write Tag, else set next state to be Read Block state
Write Tag	Asserts TVALID on master interface. Writes output of AES GCM module to TDATA on master interface. Asserts TLAST to signal end of packet.	When finished writing 128 bits, transit to Idle state
Compute	The state waits for the appropriate signals from the AES GCM Module depending on the next state If next state is Send AAD or Read Block, wait for $dii\_data\_not\_ready == 0$. If next state is Write Block, wait for $Out\_vld == 1$. If next state is Write Tag, wait for $Tag\_vld == 1$	When the appropriate signal is triggered, transit to next state

Table 4.1: States of Crypto IP

4.1.3 Block Design

Figure 4.4 describes the high-level block design of the implemented system. An AXI DMA IP is instantiated in the PL to facilitate data transfer between DDR memory and the Crypto IP using Direct Memory Access. The PS is responsible for writing the Key, Crypto Header and Data into a buffer within the DDR memory. The PS then transmits the contents of the buffer to the Crypto IP for encryption or decryption. The PS is also responsible for interfacing with the CAN Bus controller to send and receive data over the CAN Bus.

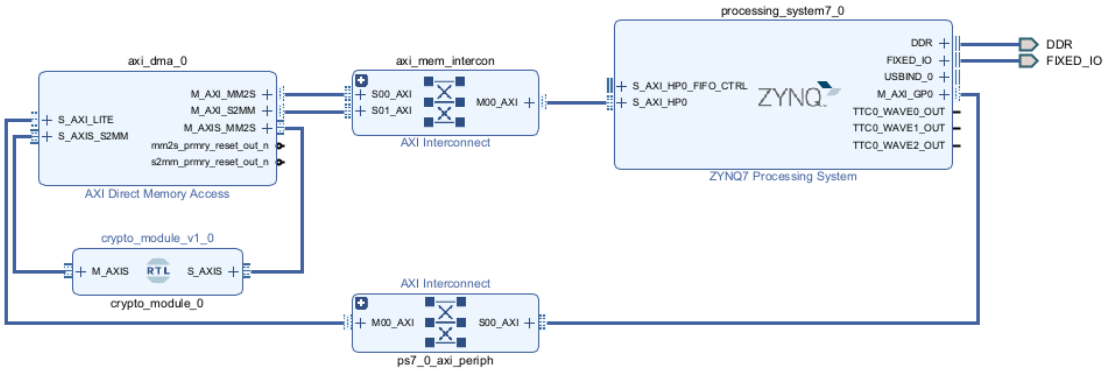

Figure 4.4: Block Design

4.2 Software Development

A Bare Metal Application developed in C, designed for execution on the PS, is responsible of several tasks. These include interfacing with the CAN Bus Controller, preparation of the key, crypto header and payload, as well as transferring the data to and from the Crypto IP using Direct Memory Access. To enhance maintainability and accommodate future developments, the application has been divided into distinct components.

4.2.1 Software Architecture

An overview of the various components in the bare metal application is shown in Figure 4.5. More details about the responsibilities and interaction between components are described in the following sections.

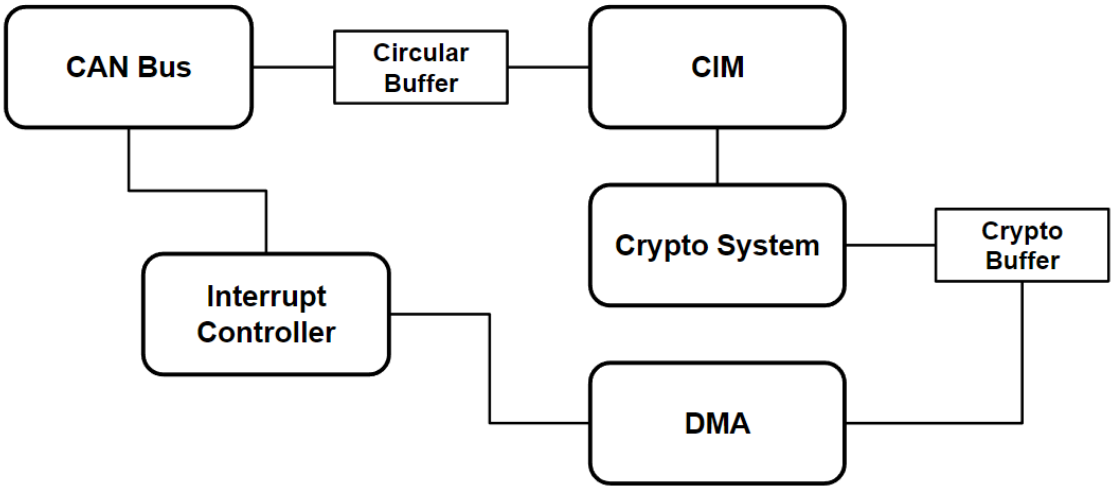

Figure 4.5: Software Architecture

4.2.2 Interrupt Controller

The CAN Bus and DMA components are interrupt driven. This component is responsible for initializing and configuring the interrupt controller within the Zynq 7000 SoC.

4.2.3 CAN Bus

This component serves as a wrapper over the CAN Bus drivers in the Board Support Package provided by Xilinx. The CAN Bus component is responsible for initializing and configuring the CAN Bus controller within the Zynq 7000 SoC. This includes enabling the CAN Bus controller, setting the baud rate and bit timing as well as enabling interrupts for the controller. This component also defines the interrupt handlers that will be triggered when the CAN Bus controller successfully transmit a frame and receives a frame. This component defines two circular buffers, when a frame is received by the controller, depending on the message ID of the frame, the frame data is inserted into the appropriate circular buffer. These two circular buffers are for storing plaintext data from the On Board Computer to be encrypted and storing Crypto Packets from the Ground Station to be decrypted.

4.2.4 DMA Component

The DMA component serves as a wrapper over the DMA drivers in the Board Support Package provided by Xilinx. The DMA component is responsible for initializing and configuring the AXI DMA IP instantiated in the PL shown in Figure 4.4. This includes setting up the interrupt system, connecting the interrupts to the interrupt handlers. The DMA component also includes utility functions to flush and invalidate caches to ensure correctness of data transferred.

4.2.5 Communication Interface Module

This is the main entry point to the bare metal application where the main function is located. The CIM is responsible for invoking the initialization functions of the various components before going into an infinite loop waiting for events. Within the infinite loop, the CIM will check if there is data present in circular buffers and pass it on to the Crypto System for processing.

4.2.6 Crypto System

The Crypto System is responsible for specifying structures associated with the Crypto Packet, as detailed in Table 3.1 and Table 3.2. The crypto_buffer, outlined in Table 4.2, holds the inputs to be sent to the Crypto IP. This component defines two dedicated crypto_buffer designed to streamline the exchange of data with the Crypto IP, one for encryption and one for decryption. The Crypto System exposes a function to accept data from the CIM and inserts it into the appropriate crypto_buffer. When the CIM passes data to the Crypto System for encryption, the Crypto System inserts the data into the encryption buffer. Similarly, when the CIM passes data to the Crypto System for decryption, the Crypto System inserts the data into the decryption buffer. When the crypto_buffer is full or a full crypto packet is received, the Crypto System then invokes the encrypt or decrypt function to perform the DMA transfers to the Crypto IP instantiated in the PL.

Field	Length (bytes)	Description
Block Number	4	Used for sending/receiving data
Block Offset	4	Used for sending/receiving data
Num Bytes	4	Keeps track of number of bytes received/sent
State	4	Used for sending/receiving data
Received Tag	16	When receiving a Crypto Packet from Ground Station, it is necessary to store the received tag, then send the crypto header and payload for AES GCM decryption to generate a tag, then compare the generated tag with the received tag for data integrity and authenicity
Key	16	128 bit Key used for AES
Reserved	4	This is set to 0 for decryption, 1 for encryption.
Crypto Header	12	Refer to Table 3.2 This field is stored in little endian, where the Most Significant Byte is stored in the lowest index
Payload and Tag	65,551	This is the buffer to accomodate the payload as well as tag generated by AES GCM. 65,535 is the maximum payload length since payload length field in the crypto header is 16 bits This field is made up of 16 byte blocks, where each block is little endian

Table 4.2: crypto_buffer

4.3 Design Considerations and Alternatives

Choice of FPGA-SoC

Current Choice	Alternatives	Rationale
Zynq 7000 SoC	PolarFire SoC	Zynq 7000 SoC offers greater ease of development due to rich ecosystem of documentation and tutorials online. The Zynq Book referenced during the Literature Review provides a comprehensive walkthrough of the FPGA-SoC and the development process. This is especially crucial coming from a computer science background with limited knowledge on hardware development.

Interfaces for Crypto IP

Current Choice	Alternatives	Rationale
AXI4 Stream	Full AXI4 or AXI4-Lite	AXI4 Stream offers greater throughput as data can be transferred every clock cycle as compared to the memory-mapped interfaces that requires sending a memory address before data can be sent. However, the PS only has full AXI4 interfaces, meaning a separate IP is required to provide memory-mapped access to an AXI4-Stream interface
1 master and slave AXI4 Stream interface	1 AXI4 Stream interface per input and output	This allows greater ease of data transfer between application running on PS and Crypto IP. The application simply needs to prepare a contiguous buffer containing required inputs and issue a single DMA Transfer.
128 bits data width	Other multiples of 8	This is done to align with the block size of AES. This choice leads to complications in the crypto_buffer management as the blocks must be stored in little endian.

Inputs for Crypto IP

Current Choice	Alternatives	Rationale
Only Key, Crypto Header, Payload	Include IV and AAD to match AES GCM module	As stated in project requirements, Figure 3.4, the IV is the 96 bits crypto header while the AAD is the crypto header padded with zeroes until 128 bits. Therefore it is redundent to transmit the same information twice. The formation of AAD by padding Crypto Header with zeroes can be done in hardware.

Data Transfer Mechanism

Current Choice	Alternatives	Rationale
AXI DMA IP	AXI Stream FIFO IP	The AXI Stream FIFO IP allows memory mapped access to an AXI4-Stream interface. This IP behaves like a FIFO queue, allowing the application running in the PS to push data onto a queue to be streamed to the Crypto IP. The outputs of the Crypto IP is streamed back to the AXI Stream FIFO IP, where the PS can pop data. However, for large payloads, this would result in inefficient use of the PS - the application is occupied pushing and popping all the data in the queue, which is a waste of resources. Instead, with the AXI DMA IP, the PS is only needs to issue a DMA transfer stating a memory address and size, the AXI DMA IP is then responsible for transferring the data from memory and streaming to the Crypto IP, while the PS is freed to do other processing.

Event Handling

Current Choice	Alternatives	Rationale
Interrupts	Polling	The CAN Bus controller is configured to interrupt the CPU upon transmitting or receiving a CAN frame. This is done to free up the CPU to do other tasks instead of continuously polling the CAN Bus controller registers to check if a CAN Frame is transmitted or received. This is also especially important since the CAN Bus controller has a limited receive queue to store CAN Frames received. Interrupts allows the application to retrieve the received CAN frame in a timely manner.
CAN Bus Receive Interrupt Handler inserts Frame Data into circular buffer	Inserting Frame Data directly into crypto_buffer	This is done to decouple the Crypto System from the CAN Bus component and to allow buffering to occur if the encryption or decryption becomes a bottleneck

4.4 Testing and Verification

To ensure correctness of AES GCM module, Crypto IP and Bare Metal Application, rigorous testing was conducted throughout the hardware and software development process.

4.4.1 Test case generation

In order to test correctness of the various components of the system, known answer tests were employed. These known answer tests were sourced from various sources and generate using python AES GCM library functions.

Test Vector 4 from [9] and several test cases from [4] was used for testing the AES GCM module from OpenCores. These were chosen due to the limitations of the AES GCM module where the Initialization Vector is assumed to be 96 bits and at least one block of Additional Authenticated Data is required.

Test Vector 4 is reproduced here for convenience:

$$
\begin{aligned}
Key &= \text{feffe9928665731c6d6a8f9467308308} \\
IV &= \text{cafebabefacedbaddecaf888} \\
AAD &= \text{feedfacedeadbeeffeedfacedeadbeef} \\
&\quad \text{abaddad2} \\
P &= \text{d9313225f88406e5a55909c5aff5269a} \\
&\quad \text{86a7a9531534f7da2e4c303d8a318a72} \\
&\quad \text{1c3c0c95956809532fcf0e2449a6b525} \\
&\quad \text{b16aedf5aa0de657ba637b39A} \\
CT &= \text{42831ec2217774244b7221b784d0d49c} \\
&\quad \text{e3aa212f2c02a4e035c17e2329aca12e} \\
&\quad \text{21d514b25466931c7d8f6a5aac84aa05} \\
&\quad \text{1ba30b396a0aac973d58e091} \\
Tag &= \text{5bc94fbc3221a5db94fae95ae7121a47}
\end{aligned}
$$

In order to test Crypto IP and Bare Metal Application, testcases had to be generate such that it follows the specification of Crypto Packet and using zero padded

Crypto Header as Additional Authenticated Data. The Python Cryptography library provides a software implementation of AES GCM encryption and decryption[10]. A Python script is developed to generate random payloads and key, form Crypto Header using payload length and performing AES GCM to form test cases. The outputs from the Python script are shown in Figure 4.6

```
Testcase 0:
key:            6428d31fc18614f1bfe1eec50054b255
crypto_header:  125c000504980d456fed9f79
plaintext:      e14416ccb7
ciphertext:     9008a54b96
tag:            33b474c2ac26216bbdf6e8224b1c9f33

Testcase 1:
key:            ee84e19cda87a76291eaaf2054aef812
crypto_header:  13360015f2cb949b8fb0013e
plaintext:      4df64bff1fa11895af337eb66b66e129 1fda3cf888
ciphertext:     f565a68f46fc5e88d8e8eacd2b3920ba 91a03324a9
tag:            63246189b304c83839391a15034e72e1

Testcase 2:
key:            47c4890cb403445547b64f6f8c85dc64
crypto_header:  126900445bff83f6766d5e73
plaintext:      dbddbec4ba171905f191f27383c6c410 11b8b18b3f93f4dce9bd84d42fc436c2 238b5974ef7ddbf8cfd0284b896f4135 c833009ba04b6bab284ebabc5a8047a6 325cf1a4
ciphertext:     0806021ccaf3d2ccb968d8e314f2bbbc 2b948577de437d7b68c6b214f5da892b 6b74c6e6545c84258e254ad422af0e3c 929b3d96ecbac109f7948438ae1bfd88 8ad1dfcf
tag:            7c7b23cbf5f820247278ecc913372a1f

Testcase 3:
key:            78ff62eda69dfc03eac40b3a00e86cef
crypto_header:  116100487e97675a3b9bbdf8
plaintext:      fc946c1935710c9c6f5d060ddf439be4 1ee8d6127a4cf636f9f93eb0b8ee6671 ae3f73da68abe6cc2e5704960f9793e6 4936cdc987758027866b079360419575 aed9a95b9eac3961
ciphertext:     c510b7814d66676c58c00b8bc52c3816 5b4689040f8b90f65d8f8c6be6a51e28 69314b3db990303ca1663aa8673d356c 91e3ca28e81b7db29ed591354a380ab9 c31f7880c6c0d298
tag:            e3bf7aa2825918b31f37708144d4e5eb

Testcase 4:
key:            a3557da8c75e9dfde2ff0bd90d0156f8
crypto_header:  21640040428fc96f2fd10ba7
plaintext:      1c6113d81c8e6421435aa83dd23af5f9 a2e75920cf12ae439445e1826ff39469 0f5075af5e3abfa7472f3744176da4bb 45bfe21d03ca895b3cfd6d1d51601351
ciphertext:     2625a9e6cff85bde930c0c53181e5148 14f8ebf6ae2217aa3d6ea33872b7db5e fe5535aab483c665d759d8fb20bc7774 6113d6201466cfed33c40d0c8db3c219
tag:            63d5bd1dfe5b78eec95386f650d876ba
```

Figure 4.6: Test Cases generated by Python Script

4.4.2 Hardware Simulation

To test the hardware modules, simulation test benches were created to pass in data and outputs were observed and compared with known answers. Simulations were done in the Vivado Design Suite.

Figure 4.7 shows the simulation of the AES GCM Encrypt module from OpenCores with inputs from Test Vector 4 from [9].

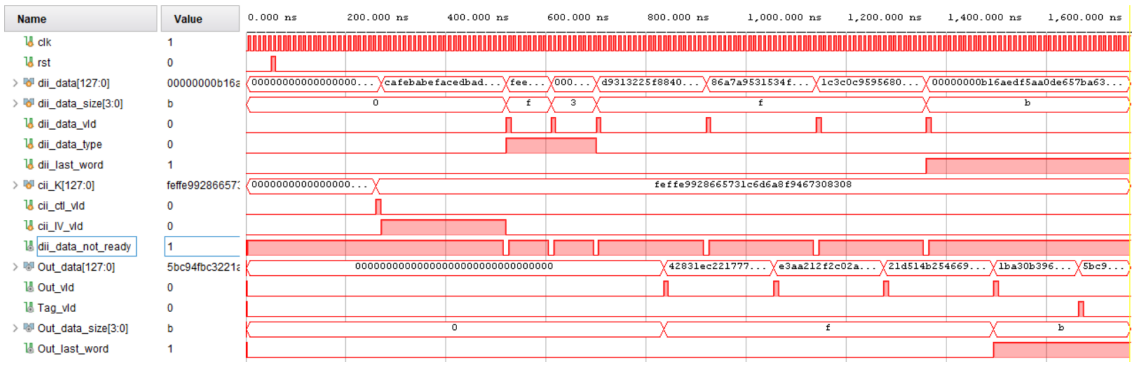

Figure 4.7: AES GCM Encrypt Module simulation

Multiple packets were simulated and a bug was discovered within the implementation. For the second packet onwards, the ciphertext generated is correct but the tag generated is incorrect. Some investigation into the Verilog code given revealed that the inputs to an internal GHASH module was not reset correctly in between packets, leading to the incorrect tag generated. Changes were made and multiple packets were successfully simulated and encrypted using the AES GCM Encrypt module.

Modifications to the Finite State Machine in the AES GCM Encrypt module was made to create the AES GCM Decrypt module. Figure 4.8 shows the simulation of the created AES GCM Decrypt module with inverse inputs from Test Vector 4 from [9]. Multiple packets were successfully simulated and decrypted as well.

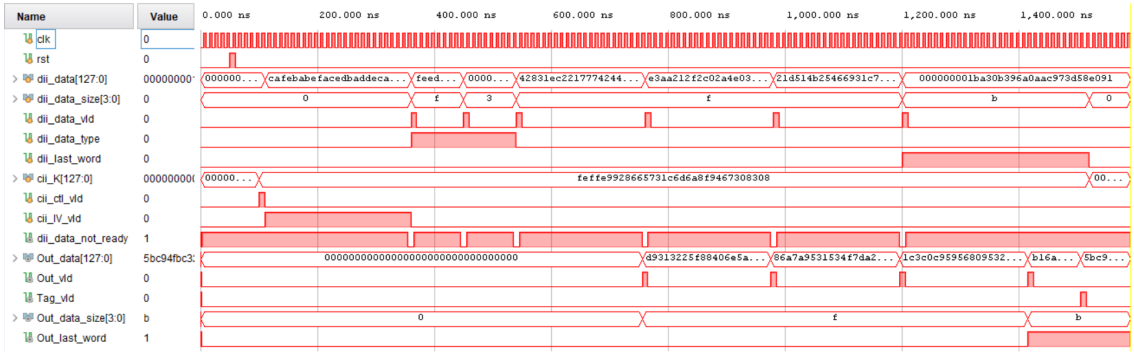

Figure 4.8: AES GCM Decrypt Module simulation

The wrapper Crypto IP module was created to convert inputs received from the

AXI4 Stream Interfaces to the custom interfaces used by the AES GCM Encrypt module and the AES GCM Decrypt module. Figure 4.9 shows the simulation of Crypto IP module with inputs from test case 1 from the test cases generated by Python script in Figure 4.6 for encryption. Multiple packets were successfully simulated, with both encryption and decryption as well.

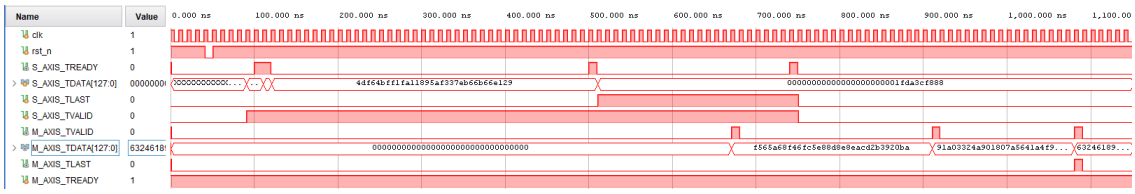

Figure 4.9: Crypto IP simulation

4.4.3 Software and Hardware Integration Testing

An initial integration testing between software and hardware is done by preparing a buffer and initializing it to hard coded values. The application then issues the DMA transfer to transfer the contents of the buffer to the Crypto IP for encryption and decryption and the outputs were validated.

Testcase 1 and Testcase 2 in Figure 4.10 are encryption tests using inputs from test case 1 and test case 2 in Figure 4.6. Testcase 3 and Testcase 4 in Figure 4.10 are decryption tests using inversed inputs from test case 1 and test case 2 in Figure 4.6.

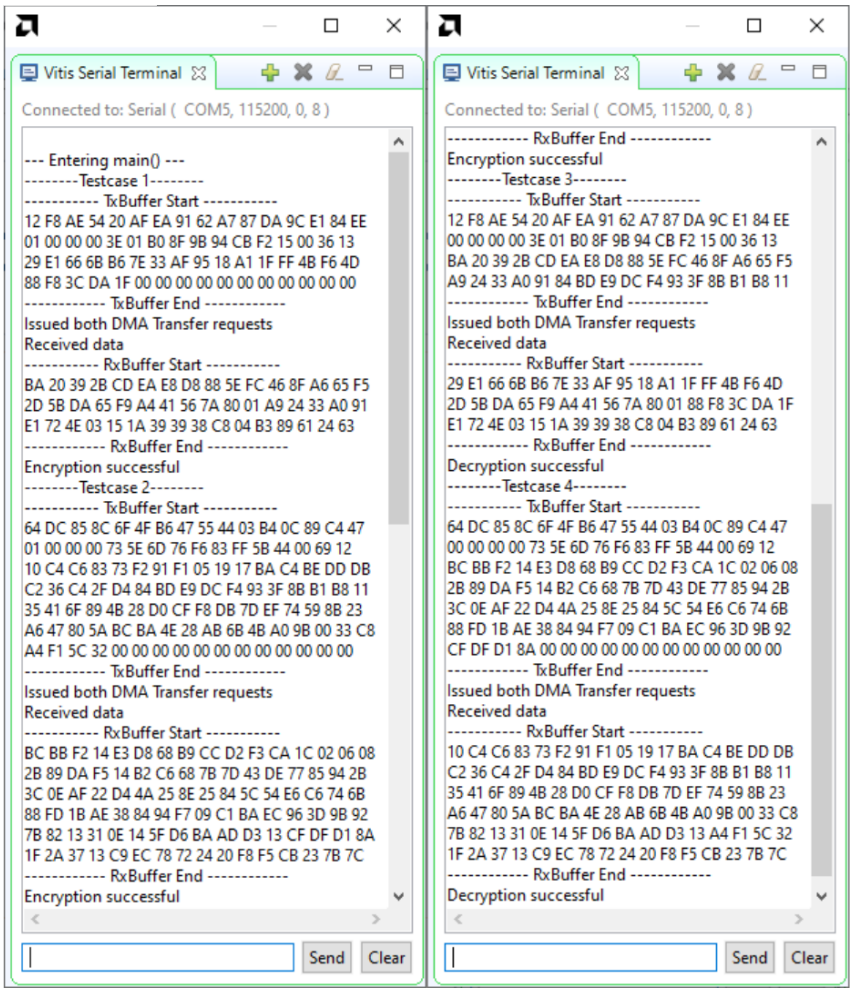

Figure 4.10: Software Hardware Integration Testing

4.4.4 End-to-End Testing with CAN Bus

Specialized hardware is required to send CAN Frames from a Desktop PC into the Evaluation board. STAR Centre has kindly provided the Total Phase Komodo CAN Solo Interface which is a USB-to-CAN adapter and analyzer. The Komodo interface is an all-in-one, high-performance solution for transmitting and monitoring CAN data. It provides fast, interactive, real-time visibility into the protocol layer of a CAN embedded system.

Total Phase also provides the Komodo GUI Software, a graphical application for use with the Komodo CAN Solo Interface. It provides access to the CAN functions of the Komodo interface in an easy-to-use graphical interface.

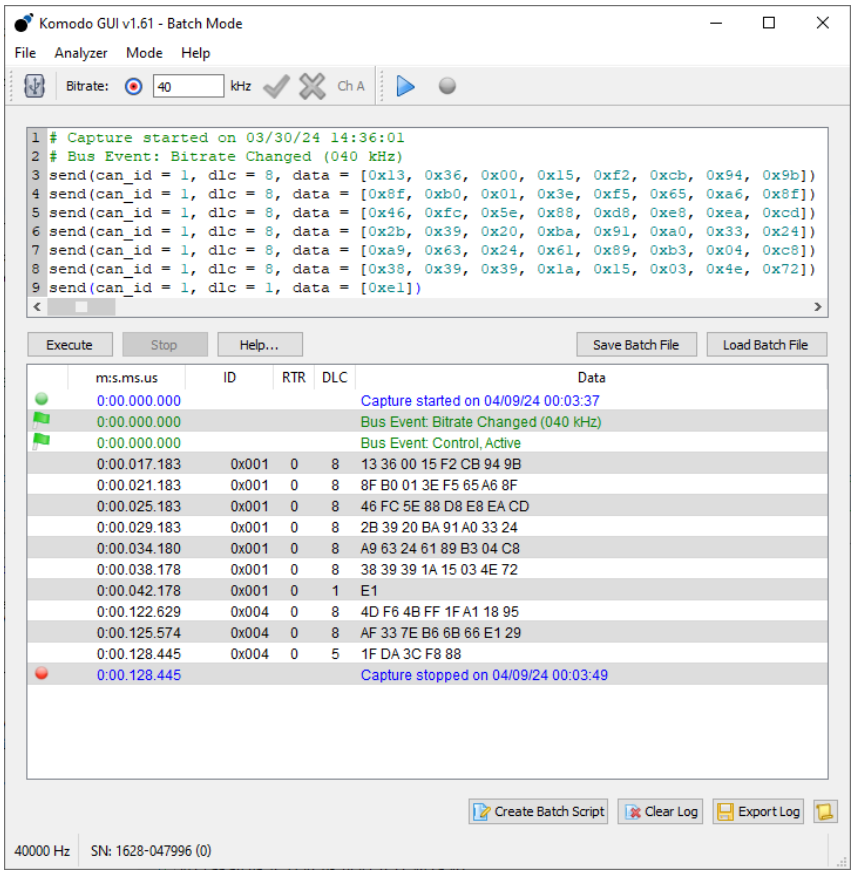

Figure 4.11: Komodo GUI Software

Figure 4.11 shows a screenshot of the Komodo GUI after sending data to the Evaluation Board through the CAN Bus. The top section is used to input a batch script, which specifies a sequence of CAN frames to be transmitted on the CAN Bus. The bottom section is a log displaying the CAN frames captured on the CAN Bus.

Test case 1 from Figure 4.6 is used for the End-to-End testing in Figure 4.11. The batch script is executed to send CAN Frames containing the Crypto Header, Ciphertext Payload and Tag to the Evaluation Board. The message ID of 1 is used to specify that these CAN Frames belong to a Crypto Packet sent by the Ground Station.

The Bare Metal Application running on the Evaluation Board receives these CAN Frames, processes the Crypto Packet and decrypts the Ciphertext Payload using the Crypto IP within the PL. Upon receiving the decrypted plaintext payload and generated tag, the Bare Metal Application then verifies data integrity and authenticity by comparing the generated tag with the received tag.

After verification, the Bare Metal Application then sends the plaintext payload out through the CAN Bus. The message ID of 4 is used to specify that these CAN frames belong to plaintext Payload to be sent to the On Board Computer. These CAN frames are then captured in Figure 4.11.

4.4.5 Testing Large Inputs with CAN Bus

The Komodo GUI Software is a convenient tool to transmit CAN frames and monitor activities on the CAN Bus for demonstration purposes. However, the nature of graphical application makes it unsuitable for transmitting large number of CAN frames and observing large outputs from the Evaluation Board.

Total Phase provides the Komodo Software API that is used to interface with the Komodo I2C/SPI Embedded Systems Interface. It provides APIs in a number of languages for maximum development flexibility. The Python language was chosen for this project for its ease of use and simple syntax.

A Python script is created to generate large test cases and interface directly with the Komodo CAN Interface drivers to transmit CAN frames to the Evaluation Board. Testcases with payloads of maximum size (65,535 since payload length field in the Crypto Header is 16 bits long) are generated and transmitted to the Evaluation Board over the CAN Bus. The Bare Metal Application developed is able to successfully receive large Crypto Packets, perform the decryption and validation, then transmit the decrypted payload through the CAN Bus.

4.5 Data Reporting

This sections provides a summary of the various reports generated by Vivado Design Suite when implementing the system.

Table 4.3 shows the utilization of various resources in the FPGA. These resources include Look Up Tables (LUT), Look Up Table Random Access Memory (LUTRAM), Flip Flops (FF) and Block Random Access Memory (BRAM).

Figure 4.12 shows the power analysis from implemented netlist, with a total On-Chip power of 1.797W. Activity is derived from constraints files, simulation files or vectorless analysis.

Figure 4.13 shows the timing summary report for the implemented design.

Resource	Utilization	Available	Utilization(%)
LUT	8940	53200	16.80
LUTRAM	240	17400	1.38
FF	9206	106400	8.64
BRAM	15	140	10.71

Table 4.3: Utilization Report

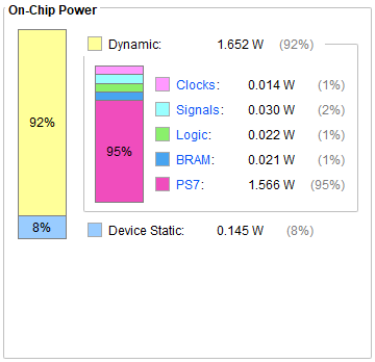

Figure 4.12: Power Analysis

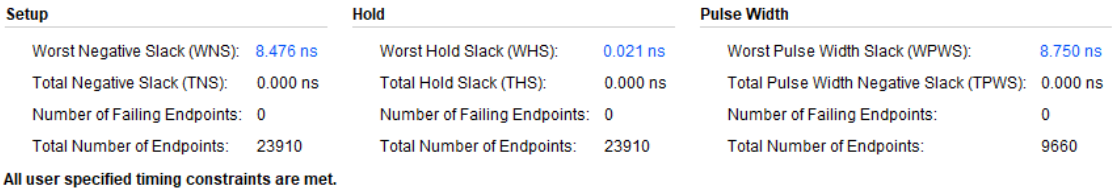

Figure 4.13: Timing Analysis

Chapter 5

Conclusion and Future Work

In conclusion, this dissertation has addressed the critical need for efficient and secure communication in satellite systems through hardware acceleration of Authenticated Encryption Schemes. The implementations presented here not only contribute to the field of hardware acceleration but also have practical implications for enhancing the security and performance of satellite communication systems. As we continue to explore new frontiers in satellite technology, further research in this area will be instrumental in advancing secure and reliable communication for future satellite missions.

5.1 Future Works

5.1.1 Integration with AES-256 GCM Module

As mentioned in 3.3, a key size of 256 is recommended by the CCSDS. Work has to be done to develop a AES-256 GCM Module for the Xilinx Zynq7000 FPGA-SoC platform. The Crypto IP wrapper module developed in this project will then have to be modified to interface with the new AES-256 GCM Module.

5.1.2 Full Duplex Encryption and Decryption

In the current implementation of Crypto IP wrapper module, it only allows either encryption or decryption to occur at one point of time. Since the resource utilization of the FPGA resources are low, it is possible to separate the encryption and decryption into separate modules with individual AXI4 Stream Interfaces. This

allows full duplex operations where there can be simultaneous data transfer to both the encryption and decryption module.

5.1.3 Storage Using Non-Volatile Memory

Encryption Keys are to be stored in multiple locations for redundancy, ideally in Non-Volatile Memory. Payload data may also be stored in Non-Volatile Memory should the encryption become a bottleneck and the on board computer continues to send payload data to CIM.

5.1.4 Self-Test

Self-tests check if the Crypto IP is working properly by feeding hardcoded test values and checking the output against the hardcoded expected output. CIM must perform encryption and decryption self-tests upon power up. Optionally, the CIM can perform encryption, decryption and key integrity self-tests at periodic intervals

Bibliography

[1] T. B. Ahmad, "Gcm-aes block specification",

[2] ARM, "Amba axi and ace protocol specification: Axi3, axi4, and axi-lite, ace and ace-lite", *ARM*, 2013.

[3] ARM, "Amba® 4 axi4-stream protocol", March 3, 2010.

[4] "Boringssl test vectors", [Online]. Available: `https://github.com/google/boringssl/blob/master/crypto/cipher_extra/test/aes_128_gcm_tests.txt`.

[5] CCSDS, "Ccsds recommended standard for cryptographic algorithms", *CCSDS*, 2019.

[6] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, "The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc", Strathclyde Academic Media, 2014, ISBN: 099297870X.

[7] M. Dworkin, "Recommendation for block cipher modes of operation methods and techniques", *National Institute of Standards and Technology Special Publication*, 2001.

[8] ISO, "Road vehicles — controller area network (can)", (ISO 11898-1:2015).

[9] D. A. McGrew and J. Viega, "The galois/counter mode of operation (gcm)",, 2005.

[10] "Python cryptography library", [Online]. Available: `https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM`.

[11] N. I. of Standards and Technology, "Advanced encryption standard (aes)", *Federal Information Processing Standards Publication*, no. 197, 2001.

[12] Xilinx, "Vitis high-level synthesis user guide", (UG1399) October 19, 2023.

[13] Xilinx, “Zynq-7000 soc data sheet: Overview”, DS190 (v1.11.1) July 2, 2018.