UNIVERSITY OF PATRAS - SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING

ΠΑΝΕΠΙΣΤΗΜΙΟ
ΠΑΤΡΩΝ
UNIVERSITY OF PATRAS

Division: Electronics & Computers
Lab: Very Large Scale of Integration LAB (VLSI Design)

Diploma Thesis

of the undergraduate student of the Department of Electrical and Computer Engineering of the School of Engineering of the University of Patras

Georgios Gkitsas

Registration number: 6732

Title

Software/Hardware Co-design and FPGA Implementation of the Security Protocol IPSec for the Internet Protocol version 6 (IPv6)

Advisor

Assistant Professor Georgios Theodoridis

Thesis Number:

Patra, 31 March 2014

Abstract

Over recent decades, computer and network security have garnered substantial attention from both academia and industry, owing to the escalating frequency and magnitude of cyber threats, the expansion of corporate and governmental networks, and the pervasive reliance on computer systems across diverse domains of human activity.

In the context of numerous solutions emerging in this rapidly evolving domain, the Internet Protocol Security (IPsec) protocol suite has emerged as a highly promising approach for safeguarding network traffic at the IP layer of the TCP/IP internet protocol stack. While its adoption initially commenced on a limited scale, the impending transition from IPv4 to IPv6 mandates the integration of IPsec implementations into every networking system, signaling a significant step forward in bolstering Internet security.

Furthermore, recent trends in the development of application-specific systems have increasingly embraced embedded system solutions. Field Programmable Gate Array (FPGA) chips have gained prominence as instrumental tools in the development and validation of embedded systems, with the ultimate implementation of such systems being an Application-Specific Integrated Circuit (ASIC), due to its competitive advantages in speed and power efficiency.

This diploma thesis undertakes the task of designing an embedded system that implements the IPsec protocol suite. The hardware/software co-design methodology is applied aiming to exploit the synergy between the two to benefit from the best of both worlds. The target device is a Xilinx Virtex 5 FPGA platform. A Microblaze processor is utilized for the software execution while optimized hardware accelerators are designed for the computationally intensive cryptographic components CBC-AES-128 and HMAC-SHA1-96. The implementation is following the protocol specifications dictated by the corresponding RFCs (Request For Comments), with all mandatory features being implemented. Finally, the system is verified and evaluated within a real-world computer network environment.

Acknowledgements

I would like to thank my supervisor Assistant Professor Georgios Theodoridis and the Ph.D. candidate George Athanasiou for all their help and advice with this thesis.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Embedded Systems . . . 1
- 1.2 FPGA's . . . 1
- 1.3 Communication Networks . . . 2
  - 1.3.1 Protocol Stack . . . 2
  - 1.3.2 Network Security . . . 4

2 Cryptography and Security Protocols 7
- 2.1 Introduction . . . 7
- 2.2 Cryptographic Algorithms . . . 7
  - 2.2.1 Symmetric Cryptography . . . 8
  - 2.2.2 Public-Key Cryptography . . . 9
  - 2.2.3 Hash Functions . . . 10
  - 2.2.4 Keyed Hash Functions . . . 10
  - 2.2.5 Pseudorandom Number Generators . . . 11
- 2.3 Key Management . . . 11
- 2.4 Security Hardening . . . 11
  - 2.4.1 Padding . . . 12
  - 2.4.2 Salt . . . 12
  - 2.4.3 Initialization Vectors . . . 12

3 The IPsec Protocol Suite 13
- 3.1 Overview . . . 13
  - 3.1.1 Outbound Packet Processing . . . 15
  - 3.1.2 Inbound Packet Processing . . . 15
  - 3.1.3 Transport and Tunnel Modes . . . 18
- 3.2 Security Associations (SA) . . . 19
- 3.3 Security Policy Database (SPD) . . . 19
- 3.4 The Authentication Header (AH) Protocol . . . 20
  - 3.4.1 Outbound Packet Processing . . . 20
  - 3.4.2 Inbound Packet Processing . . . 21
- 3.5 The Encapsulation Security Payload (ESP) Protocol . . . 21
  - 3.5.1 Outbound Packet Processing . . . 22
  - 3.5.2 Inbound Packet Processing . . . 23
- 3.6 Key Management . . . 23

4 System Overview **25**
4.1 Design Goals . . . 25
4.2 Design Decisions . . . 25
4.2.1 Native Implementation . . . 25
4.2.2 Hardware and Software . . . 26

5 Hardware Components **27**
5.1 Architecture . . . 27
5.2 Custom Cryptographic IP Cores . . . 27
5.2.1 CBC-AES-128 . . . 27
5.2.2 HMAC-SHA1-96 . . . 43
5.3 Hardware-side Interface . . . 50

6 Software Components **51**
6.1 lwIP . . . 51
6.2 IPsec . . . 51
6.2.1 IPsec . . . 52
6.2.2 AH and ESP . . . 53
6.2.3 SADB and SPD . . . 54
6.3 Integration with lwIP . . . 57
6.4 Software-side Interface . . . 58

7 FPGA Implementation and On-Chip Verification **61**
7.1 Tools . . . 61
7.1.1 Xilinx Suite and Modelsim . . . 61
7.1.2 Wireshark . . . 61
7.1.3 Scapy . . . 62
7.1.4 Linux ipsec-tools . . . 62
7.2 Final System Design and Implementation . . . 62
7.3 On-Chip Verification . . . 63
7.4 Evaluation . . . 66

8 Roadblocks and Solutions **71**

Bibliography **75**

List of Figures

1.1 Structure of a Xilinx Virtex-5 Configurable Logic Block (CLB) . . . 2
1.2 (left) The OSI stack, (right) the TCP/IP stack . . . 3
1.3 Packet transformation through the protocol stack . . . 4

2.1 Symmetric Encryption . . . 8
2.2 CBC Mode . . . 9
2.3 Public Key Encryption . . . 9
2.4 Public Key Signing . . . 10
2.5 HMAC Operation . . . 11

3.1 IPsec's position in the network stack . . . 13
3.2 Host-to-Host in Transport Mode . . . 14
3.3 Network-to-Host in Tunnel Mode . . . 14
3.4 Network-to-Network in Tunnel Mode . . . 14
3.5 Outbound packet processing flowchart . . . 16
3.6 Inbound packet processing flowchart . . . 17
3.7 AH (up) and ESP (down) packet formats in Transport Mode . . . 18
3.8 AH (up) and ESP (down) packet formats in Tunnel Mode . . . 18
3.9 Structure of the AH protocol header . . . 20
3.10 Structure of the ESP protocol header . . . 22

5.1 High-level system architecture . . . 27
5.2 (left) AES encryption algorithm, (right) AES decryption algorithm . . . 28
5.3 AES input bytes, state array, and output bytes . . . 29
5.4 Multiplicative inverse in $GF(2^8)$ . . . 29
5.5 The RotWord operation . . . 32
5.6 Combined implementation of SBOX and InvSBOX . . . 34
5.7 Combined implementation of SubBytes and InvSubBytes . . . 35
5.8 Combined implementation of MixColumns and InvMixColumns . . . 35
5.9 Implementation of MixOneColumn . . . 36
5.10 Implementation of XTimes . . . 37
5.11 Implementation of XTimes2 . . . 37
5.12 Implementation of one round of AES-128 . . . 39
5.13 Implementation of one round of AES-128 Key Scheduler . . . 40
5.14 Implementation of the AES-128 Key Scheduler . . . 40
5.15 Implementation of CBC-AES-128 . . . 42
5.16 The control FSM of CBC-AES-128 . . . 42
5.17 The HMAC algorithm . . . 44
5.18 SHA1 message schedule . . . 46
5.19 sha1_core module implementation . . . 46
5.20 sha1 module implementation . . . 47

5.21 sha1 control FSM 48
5.22 hmac_sha1_96 module implementation 48
5.23 hmac_sha1_96 control FSM 49

6.1 Software architecture diagram for the custom IPsec library 52
6.2 Integration of inbound IPSec packet processing in lwIP 57
6.3 Integration of outbound IPSec packet processing in lwIP 58

7.1 Complete system architecture. 63
7.2 ECHO server in Transport Mode 64
7.3 Transmitted packet of ECHO request in ESP Transport Mode 65
7.4 Decrypted payload of the ECHO request in ESP Transport Mode 65
7.5 ECHO server response in ESP Transport Mode 65
7.6 Decrypted payload of ECHO server response in ESP Transport Mode 66
7.7 Transmitted packet of ECHO request in AH Transport Mode 66
7.8 ECHO server response in AH Transport Mode 66
7.9 Throughput comparisons across SW/HW and inbound/outbound flows 69

List of Tables

7.1 Inbound packet processing time (in number of clock cycles) 67
7.2 Outbound packet processing time (in number of clock cycles) 68
7.3 Throughput (KB/s) 69
7.4 Area utilization 69

1. Introduction

1.1 Embedded Systems

An embedded system constitutes a specialized computing system engineered to execute predetermined application-specific tasks, contrasting with the versatility of general-purpose computers. Its defining attributes encompass the inclusion of at least one microcontroller, compact physical dimensions, low cost, and adherence to strict resource limitations. It is crucial to emphasize that the primary differentiation between an embedded system and a conventional computer stems not from their size but from their designated functionalities. The complexity of embedded systems can vary, ranging from rudimentary configurations with minimal peripherals to intricate systems present in airplanes.

At the core of an embedded system lies at least one microprocessor or microcontroller programmed to run a specific software application. This process of ”specialization” in embedded systems enables their designers to optimize these systems, yielding enhancements in size, cost-efficiency, execution speed, or power efficiency. Predominant architectures prevalent in embedded systems are ARM, MIPS, PowerPC, Microblaze, and X86.

1.2 FPGA’s

FPGA (Field Programmable Gate Array) is a type of general-purpose programmable integrated circuit that features a multitude of logic components such as Look-up Tables (LuTs), logic gates, counters, memory registers, PLL generators, and sometimes it can also include analog functions. The basic unit of an FPGA is the Logic Block (LB) which is a predetermined collection of such basic components. Additionally, there are blocks responsible solely for the input/output of the FPGA (I/O Blocks). LBs and IO Blocks are interconnected either with fixed wiring or through switch matrices.

In the process of FPGA programming, specific logic elements are configured to perform a specific function, while the requisite interconnections are activated to facilitate the desired data flows. After programming, the FPGA operates akin to a custom-integrated circuit built to execute the programmed functionality.

In the case of Xilinx FPGAs, LBs are called Configurable Logic Blocks (CLBs) and are partitioned into smaller units called slices. The configuration and composition of CLBs and slices vary across FPGA families. Depending on the specific FPGA variant, a CLB may accommodate 2, 4, or 6 slices. Each slice comprises LUTs and possibly predefined logic components, such as small multiplexers, logic gates, and adders. Depending on the FPGA family, LuTs may be (4 inputs/1 output) or (6 inputs/2 outputs). The composition of predefined logic components also varies, from generic logic gates to Digital Signal Processing (DSP) components. An example CLB is shown in Figure 1.1.

FPGAs are available either as standalone chips suitable for integration onto printed circuit boards (PCBs) or as part of integrated solutions where the FPGA is embedded within a specialized board. These integrated solutions typically come equipped with a module responsible for the FPGA programming and offer a range of standard interfaces including serial ports (UART), and Ethernet, as well as interfaces for external memory and storage devices.

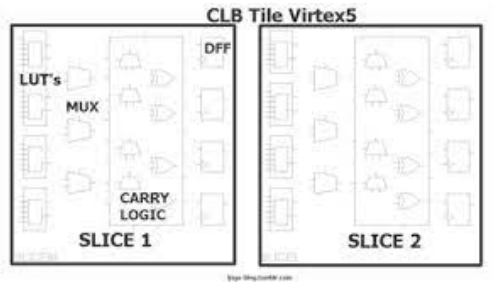

Figure 1.1: Structure of a Xilinx Virtex-5 Configurable Logic Block (CLB)

The primary advantages of FPGAs over traditional implementations include:

- FPGAs occupy a strategic middle ground between software-based solutions and Application-Specific Integrated Circuits (ASICs), offering faster implementations compared to software-based approaches, while remaining more cost-effective than ASICs.
- FPGAs liberate designers from reliance on component manufacturing and distribution chains, as the behavior of the device is not inherent to its physical structure but is determined by the designer's configuration.
- The reprogrammable nature of FPGAs fosters reusability and facilitates upgradability, enabling iterative refinement of designs without necessitating hardware replacements.

1.3 Communication Networks

A computer network is defined as a set of interconnected devices purposed for facilitating communication among them. Different types of networks exist, depending on their intended functions and operational constraints. The Internet is the largest network of interconnected computing devices.

Various technologies are available to choose from when connecting devices, including Ethernet cables, optical fibers, and wireless mediums leveraging electromagnetic waves such as microwaves. The selection of a particular technology depends on several criteria, with the most important considerations being performance and cost-effectiveness.

In addition to establishing physical interconnectivity among systems, the formulation of precise rules and communication protocols is required for any communication attempts to be successful. These communication protocols serve as the foundational framework underpinning the proliferation and diversification of network-centric applications.

Another important characteristic of networks is the transmission of information in the form of packets. Each packet encapsulates either the entirety or a portion of the transmitted data. Alongside the information intended to be transmitted (payload), packets include one or more headers containing information needed for the transmission of the packet itself. Intermediate network nodes leverage this header information to facilitate packet routing, whereas the payload content is exclusively intended for consumption by the ultimate recipient.

1.3.1 Protocol Stack

Networks are structured upon the foundational principle of modularity, wherein their operational framework is compartmentalized into discrete and ideally autonomous functions. This modular approach not only reduces the inherent complexity of these systems but also enables flexibility and adaptability since modifications don't require a comprehensive overhaul of the entire architecture but rather focus on relevant segments.

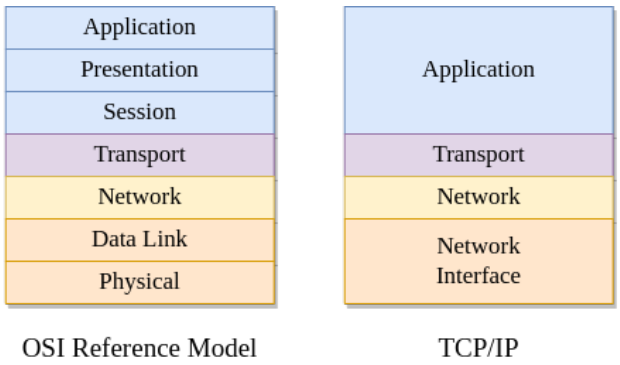

Figure 1.2: (left) The OSI stack, (right) the TCP/IP stack

Central to network operation is the protocol stack, comprising a hierarchical arrangement of layers dedicated to information processing of the transmitted information. This processing aims to ensure a seamless exchange of information between systems (hosts). This requires the existence of an implementation of the protocol stack at each end of the communication. Each layer within the stack assumes a distinct function and provides services to the layer above. During the transmission phase, information processing commences from the highest layer and cascades downward, reaching the lower layer where the physical transmission takes place, whereas, in the reception phase, the reverse path is followed, with the upper layer providing the information to the host application.

The protocol stack's architecture is standardized by the OSI (Open Systems Interconnection) model, originating from a proposal developed by the International Standards Organization (ISO) to initiate global standardization of network protocols across diverse layers. The OSI model abstractly delineates these layers, their functionalities, and their interconnections. It refrains from specifying protocols but instead outlines the functions at each layer should perform. Consequently, a multitude of protocols exists for each layer, tailored to distinct purposes and addressing diverse requirements. The OSI layers are outlined as follows:

- Physical Layer
  Responsible for data transmission across the physical communication channel, the Physical Layer is tasked with encoding/decoding data into the channel's respective code, such as electrical pulses (e.g., 0-5V for cable transmission) or optical pulses (for optical fiber transmission). Its primary objective is to ensure accurate reception of transmitted bits without degradation or misinterpretation.

- Data Link Layer
  Assuming control over data transmission from the Physical Layer, the Data Link Layer ensures reliable data delivery, encompassing error detection, correction, and recovery procedures. It segments data into discrete frames for transmission and conducts error checking upon reception of each frame. Additionally, it may include traffic regulation mechanisms to regulate data flow and prevent congestion.

- Network Layer
  It orchestrates the routing of information through intermediate systems to facilitate communication between non-directly connected systems and hosts belonging to disparate local networks.

- Transport Layer
  Catering to scenarios where multiple applications or users seek access to the network, the Transport Layer manages the multiplexing and demultiplexing of data, ensuring accurate delivery to the intended recipient(s).

- Session Layer
  Enabling users from different systems to establish sessions, the Session Layer offers services such as dialogue control, token management, and synchronization.
- Presentation Layer
  In contrast to lower layers that focus on bit transmission, the Presentation Layer concerns itself with the syntax and semantics of transmitted information. It manages abstract data structures to facilitate interoperability among systems employing different data representations, facilitating standardized encoding for data exchange
- Application Layer
  Comprising a suite of protocols catering to diverse user requirements, the Application Layer facilitates functions such as file transfer, email communication, and web page presentation. This is the layer where data is created and where transmitted data gets received.

The Internet utilizes the TCP/IP stack, incorporating the TCP (Transmission Control Protocol) and IP (Internet Protocol) as its primary protocols. While adhering to the OSI model, the TCP/IP stack merges the transport and session layers within the application layer. Additionally, the lower layers (physical and data link) are often merged as well, forming a cohesive network interface layer. Figure 1.2 illustrates the OSI and TCP/IP stacks, highlighting their similarities and differences.

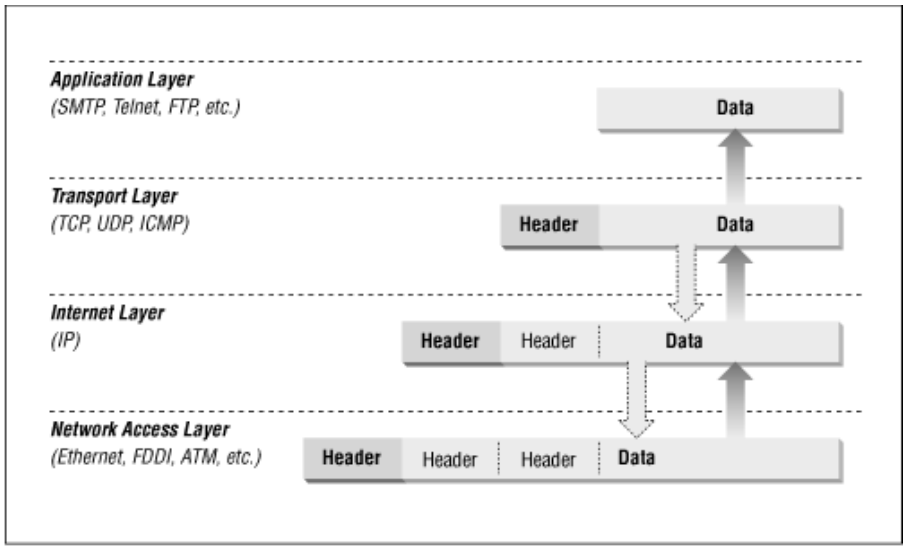

Figure 1.3: Packet transformation through the protocol stack

During transmission, data originates at the application layer and traverses sequentially through the protocol stack until it reaches the physical layer for transmission. At each layer, the data received from the layer above is augmented with a suitable header, and in some cases, additional data may be appended to the ”tail” of the packet. This iterative process, known as encapsulation, ensures that each layer includes information needed by its corresponding layer on the other side of the communication. During reception, the data first reaches the physical layer and moves upwards in the stack. Each layer successively processes and strips away its corresponding header, thereby progressively unveiling the core content of the packet. The core content is ultimately delivered to the application layer without any headers. The abstract representation of the packet format at each layer is depicted in Figure 1.3.

1.3.2 Network Security

The domain of network security encompasses the establishment of secure communication channels among networked systems, the regulation of access to network services and resources, and the safeguarding of individual systems and the network as a cohesive entity. As part of the broader context

of information security, network security is grounded in fundamental concepts such as confidentiality and authentication, tailored specifically to the realm of network environments.

Central to network security is the design, analysis, and deployment of security protocols. In the design phase, the objectives and services the protocol needs to provide are defined. Subsequently, the protocol details are specified, including packet header formats, request-response mechanisms, and algorithm choices. Furthermore, during the design phase, different operational scenarios are delineated and scrutinized.

In the analysis phase, the efficacy of the design in meeting predefined objectives is rigorously assessed, while potential flaws or oversights are identified and mitigated. The analysis is an iterative refinement process of revisions and enhancements until the protocol design satisfies its objectives.

Following analysis, the protocol is implemented. In this stage implementation and verification methodologies are selected. Of paramount importance is the secure implementation of the protocol. Secure implementation involves addressing technical flaws that could unintentionally create security weaknesses. Even if a protocol provides strong security features, an improperly implemented version could expose the system to unforeseen risks. Secure programming practices, secure hardware design, and defining access control are part of the secure implementation phase. The protocol implementation phase also includes performance and reliability considerations.

Cryptography assumes a pivotal role in the realm of network security, providing foundational tools and mechanisms for safeguarding network communications. Network security, and by extension, information security as a whole, relies on the robustness and correct usage of cryptographic algorithms.

Notable objectives of network security include data confidentiality and integrity, attestation mechanisms, resource availability, and the formulation of contingency plans for incident response and recovery, concepts that will be elaborated upon in subsequent chapters.

2. Cryptography and Security Protocols

2.1 Introduction

Cryptography is the scientific field concerned with the study, development, and utilization of techniques aimed at secure communication between parties amidst the presence of adversarial entities seeking to intercept or manipulate transmitted data. In practice, it includes the creation and analysis of algorithms and protocols that aim to provide data confidentiality, data integrity, authentication, and non-repudiation. Modern cryptography is closely related to the fields of mathematics, computer science, and electrical engineering. Key concepts in this field include:

- Confidentiality, which ensures that only selected entities can access a set of information.
- Authentication, is the mechanism ensuring the genuineness of an entity's identity.
- Data Integrity, which verifies that data remains unaltered during transmission and detects any unauthorized modifications.

A communication protocol is a definition of message formats and rules that are considered valid in communication. A protocol defines the syntax, semantics, and synchronization aspects of this communication. It stipulates the behavior of individual systems in an implementation-independent manner, allowing for protocols to be implemented in either hardware or software. For any successful communication, all participants must agree in advance on the protocols to be used.

A security protocol or cryptographic protocol protects communication using cryptographic methods. The fundamental services typically provided by cryptographic protocols are the authentication of participants, data integrity, and confidentiality, non-repudiation, and system availability (resilience to denial-of-service attacks). Alongside protocols directly protecting data, there exist protocols facilitating key exchange, negotiating communication parameters, and other auxiliary functions needed for establishing and maintaining secure communication.

Examples of cryptographic protocols include Transport Layer Security (TLS) which is used to secure HTTP traffic, the Diffie-Hellman key exchange which establishes a common secret over an insecure communication channel between two systems, and IPsec, which provides communication protection at the network layer of the TCP/IP stack.

2.2 Cryptographic Algorithms

Cryptographic algorithms serve as foundational elements in cryptography, offering essential mechanisms for secure systems. Due to their core position and heavy utilization in communication systems, attention to their efficiency is paramount.

There are various types of cryptographic algorithms, such as algorithms that provide data encryption, hashing, plausible deniability or deniable encryption, and steganography. Below we elaborate on the classes of cryptographic algorithms that are relevant to this thesis.

2.2.1 Symmetric Cryptography

Encryption is a transformation of a set of data into another, to secure this data from third-party access, i.e., aiming to provide confidentiality. The original data are called plaintext, while the data resulting from encryption are called ciphertext. The reverse transformation is called decryption.

Modern cryptography uses cryptographic keys for encryption and decryption, which parameterize the encryption algorithm. This parameterization decouples data confidentiality from cryptographic algorithm design. Assuming the algorithm is secure and implemented securely, then data confidentiality relies only on the secure protection of the key(s).

There are two basic types of encryption: symmetric and public key encryption. In symmetric encryption, a key is used for both encryption and decryption. The sender encrypts using the key, which is securely provided to the intended recipients, who in turn use it for decryption. Figure 2.1 illustrates the model of symmetric encryption.

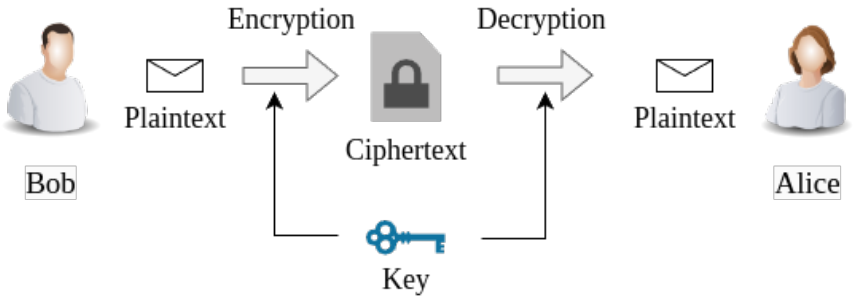

Figure 2.1: Symmetric Encryption

Two categories of symmetric encryption algorithms exist, block ciphers and stream ciphers. Block ciphers encrypt a message into equal-sized blocks, while stream ciphers encrypt it bit by bit or byte by byte. The former are suitable when all data are available beforehand, whereas the latter encrypt data as it's generated (on-the-fly).

Block algorithms operate in modes that describe how the algorithm iterates over input data larger than the block size. When the input isn't a multiple of the block size, padding is employed to reach the appropriate size. The extended text is then inputted into the algorithm. Padding methods are typically specified in the algorithm's documentation. Various operation modes for block algorithms exist, including ECB, CBC, and CFB. Figure 2.2 illustrates the CBC (Cipher-block Chaining) mode [1].

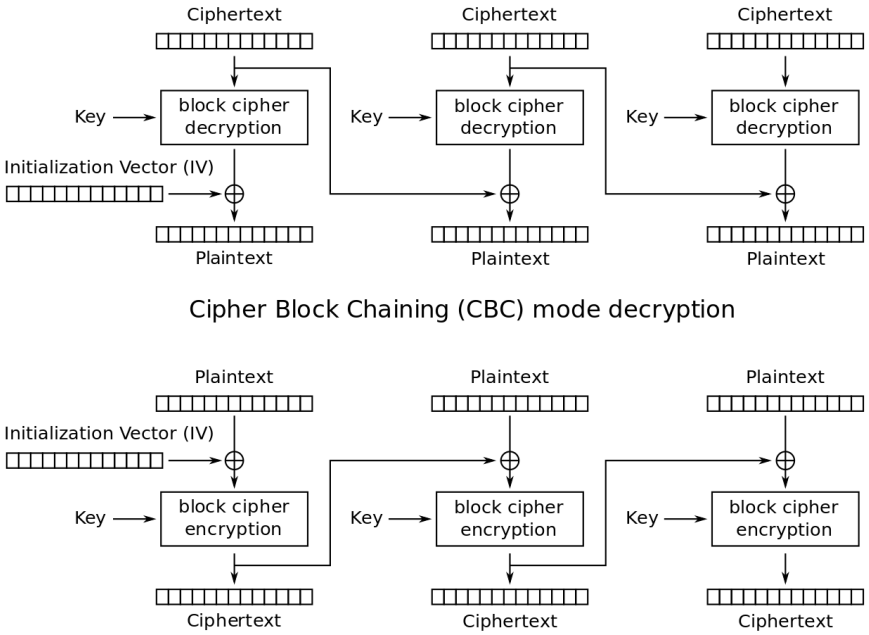

Figure 2.2: CBC Mode

2.2.2 Public-Key Cryptography

In public-key cryptography, each entity employs two keys: a public key and a private key. The public key is openly accessible, whereas the private key must be kept secret. These keys have the property that data encrypted with one key can be decrypted with the other. Furthermore, given the public key, discovering the private key is practically infeasible. Public-key algorithms serve two primary purposes: confidentiality and authentication.

When an entity wants to send information confidentially to another, it uses the other's public key to encrypt the data. Only the holder of the corresponding private key can access the plaintext. The process is illustrated in Figure 2.3.

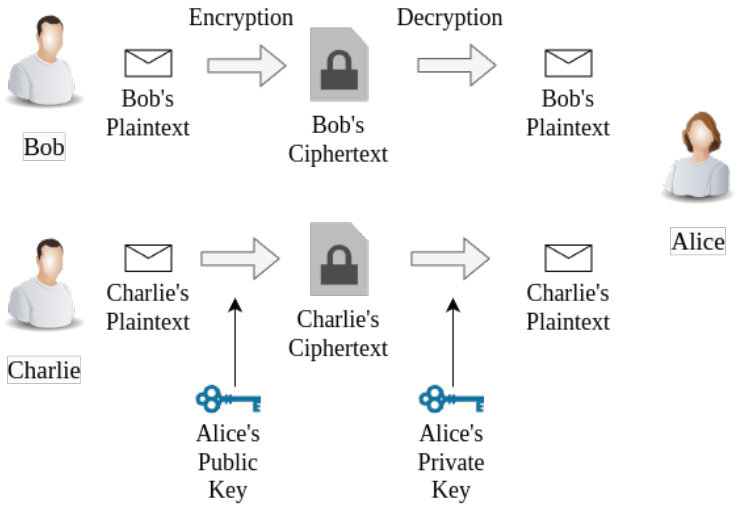

Figure 2.3: Public Key Encryption

For authentication, the sender applies their private key to the data, generating a signature that is ideally impossible to forge. Upon receipt, the recipient verifies the signature using the sender's public

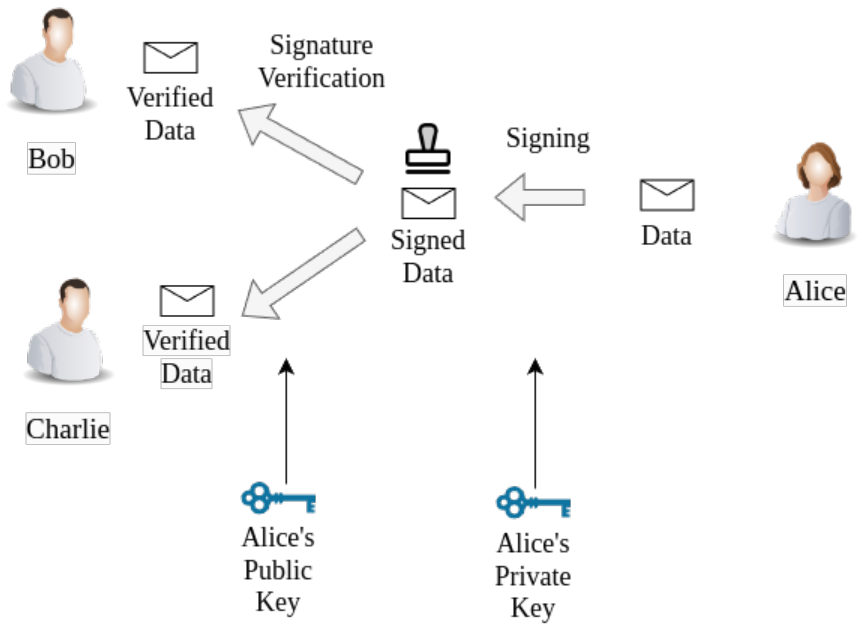

Figure 2.4: Public Key Signing

key. This process is illustrated in Figure 2.4.

Examples of applications of public key algorithms are digital certificates and secure key exchange (e.g., Diffie-Hellman) in TLS.

2.2.3 Hash Functions

A hash algorithm is a deterministic algorithm that maps variable-sized data to fixed-sized data, called hash or digest. Due to the finite number of potential outputs compared to the infinite input space, hash algorithms inevitably encounter collisions, where distinct inputs yield identical outputs.

In cryptographic applications, cryptographic hash functions adhere to additional stringent criteria:

- Difficulty in finding an input that generates a specific output.
- Difficulty in modifying the input while keeping the output the same.
- Difficulty in discovering two inputs that produce identical outputs.

The higher the degree of the above difficulties, the greater the hash algorithm's security. Cryptographic hash algorithms primarily serve serve data authentication and integrity purposes. Notable examples include MD5, RIPEMD, and SHA [2].

2.2.4 Keyed Hash Functions

To ensure data integrity and authentication, message authentication codes (MACs) are commonly employed. A MAC consists of a small piece of information appended to the message to enable verification upon receipt. Cryptographic hash algorithms are often used in conjunction with a key (Hash-based MAC – HMAC) which is accessible solely to the communicating endpoints. Any cryptographic hash algorithm, such as SHA-1 and MD5, can compute an HMAC, resulting in algorithms like HMAC-MD5 and HMAC-SHA-1, respectively. The process is depicted in Figure 2.5.

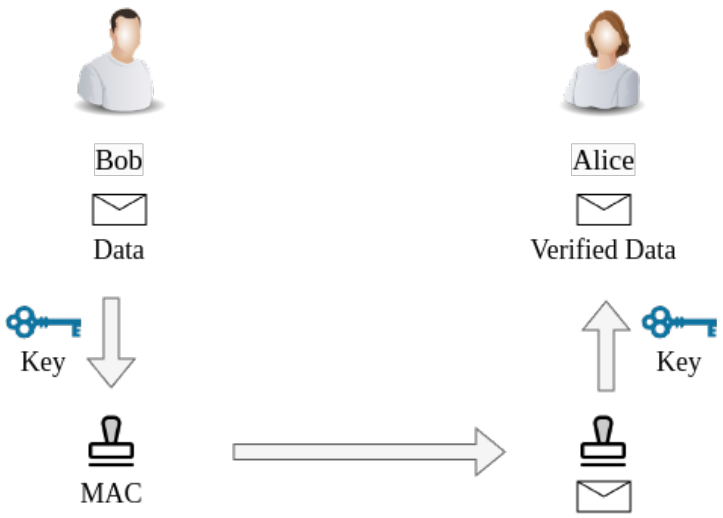

Figure 2.5: HMAC Operation

2.2.5 Pseudorandom Number Generators

Random Number Generators (RNGs) are algorithms designed to generate sequences of numbers that mimic randomness. Two categories of RNGs are True RNGs and Pseudo RNGs. True RNGs are generators that usually generate numbers from random physical phenomena while Pseudo RNGs usually rely on a small amount of physical randomness, called seed, which they use to further generate numbers that appear random but are not truly random as they still exhibit patterns and repetitions.

In cryptography, RNGs play a vital role when randomness is required for security. They are commonly utilized in tasks such as cryptographic key generation and initialization vector creation.

2.3 Key Management

Key management encompasses key creation, distribution/exchange, storage, and replacement—a crucial process for system security often challenging to implement due to policy considerations and external factors.

Before initiating secure communication, entities must establish communication parameters, including cryptographic keys for symmetric encryption. Key exchange facilicates the need to establish such keys over insecure communication channels. A notable key exchange protocol is the Diffie-Hellman exchange.

Keys undergo a lifecycle, necessitating periodic replacement. This practice mitigates risks associated with key compromise. Regular replacement limits the impact of key exposure and adds complexity for malicious entities, as it reduces the volume of data encrypted with a single key, thereby limiting available data for cryptanalysis.

2.4 Security Hardening

Several considerations are involved when trying to improve the security of a cryptographic system. One of the most important parameters in security is the key size. It directly correlates with security strength; larger sizes correspond to increased security levels. Furthermore, encryption algorithms are evaluated based on the important properties of confusion and diffusion. Cryptographic hash algorithms should exhibit collision resistance, making collisions difficult to find. Similarly, an effective pseudorandom number generator should produce sequences with extended periods before repetition.

Beyond these algorithmic properties, additional techniques are employed to bolster security at the implementation level, such as side-channel countermeasures and formal verification techniques.

2.4.1 Padding

In addition to configuring appropriate input sizes for block algorithms, data extension also helps conceal the actual size of the data packet. Additional padding can be used as a way to increase security albeit at the cost of throughput.

2.4.2 Salt

The term ”salt” is used to denote additional data added to the input of hash algorithms. Adding salt to the input results in a different output from the algorithm, even with the same input. This method mitigates attacks such as rainbow tables.

2.4.3 Initialization Vectors

An Initialization Vector (IV) is random data the size of a block which is placed at the beginning of the plaintext data, with the aim of further enhancing the security of block algorithms. They introduce randomness into the encryption process since for encrypting the same text under the same key, the result will depend on the IV value. The requirement for IVs is that they need to be unpredictable.

3. The IPsec Protocol Suite

3.1 Overview

IPsec (Internet Protocol Security) comprises a suite of protocols designed to secure communications at the IP (Internet Protocol) level across both IPv4 and IPv6 [3] versions. It provides authentication and/or encryption services for each IP packet within a communication session. Additionally, IPsec encompasses protocols for mutual authentication during session initiation and cryptographic key negotiation. Optional features include packet replay protection. Its specifications are outlined in various RFCs (Request for Comments), detailing protocol components and their interactions.

Operating at the Network layer of the TCP/IP stack [4], IPsec serves as an end-to-end security mechanism. It safeguards communication between hosts (Host-to-Host), between Security Gateways (Network-to-Network), or between an end and a Security Gateway (Network-to-Host).

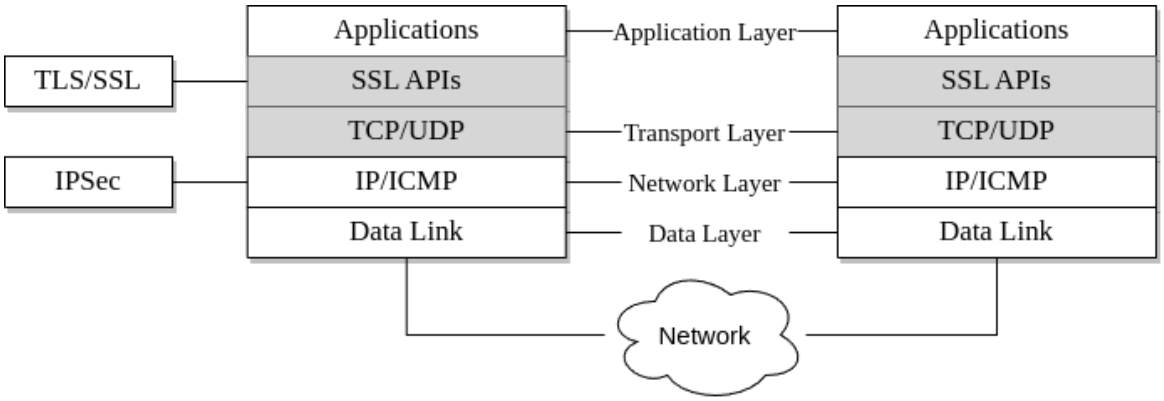

Figure 3.1: IPsec's position in the network stack

Operating at the Network layer enables IPsec to provide its services independently of which protocols are utilized at higher layers, while ensuring protection across all layers above it and, optionally including the IP layer.

IPsec defines two protocols: the Authentication Header (AH) [5] and the Encapsulating Security Protocol (ESP) [6]. AH offers data integrity, data origin authentication, and optional anti-replay protection, whereas ESP augments AH's functionalities by including data confidentiality [1]. Utilized cryptographic algorithms include HMAC-SHA1 ([7], [8], [9]) and HMAC-MD5 for integrity and authentication, along with AES [10] and 3DES for confidentiality. Nevertheless, the framework accommodates the utilization of alternative cryptographic algorithms as needed.

In IPsec, two communication modes are defined: Transport mode and Tunnel mode. Transport mode secures end-to-end communication (Host-to-Host), whereas Tunnel mode extends protection to Network-to-Network and Network-to-Host scenarios. Figures 3.2, 3.3, and 3.4 visually depict these communication modes, with the green line denoting the secured segment of the route.

Before initiating secure communication, various parameters must be established and mutually agreed upon, including cryptographic algorithms, keys, and the chosen protocol (AH or ESP). A choice

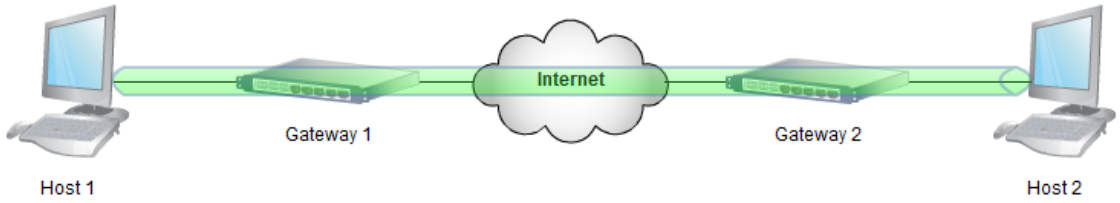

Figure 3.2: Host-to-Host in Transport Mode

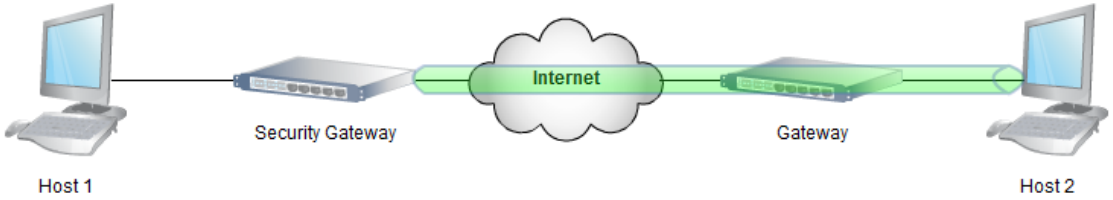

Figure 3.3: Network-to-Host in Tunnel Mode

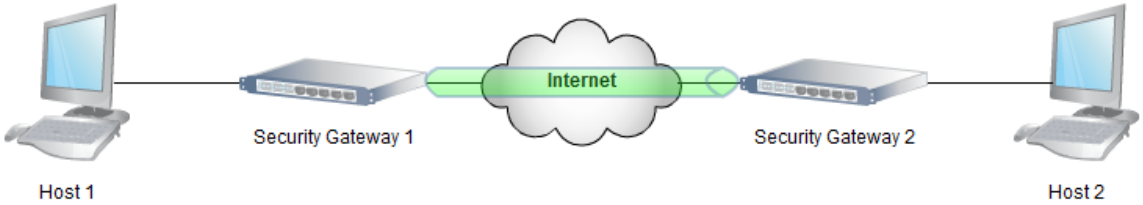

Figure 3.4: Network-to-Network in Tunnel Mode

of these security parameters constitutes a Security Association (SA), For each new secure session, an SA should be created at both ends of the communication.

The establishment of SAs within IPsec can be manual or automated. Manual configuration necessitates users to specify parameters at both ends, potentially involving intermediate Security Gateways. Manual configuration has many drawbacks such as maintenance overheads and susceptibility to human errors, hence automated methods are generally favored. The Internet Key Exchange Protocol (IKE) orchestrates automated parameter negotiation, secure key exchange, and maintenance, drawing upon the Oakley and ISAKMP protocols.

Security associations contain the parameters for secure communication. However, they do not determine whether the two ends are legitimate to communicate via IPsec. This determination, as well as whether IPsec should be used in the communication, is defined through the Security Policy (SP) mechanism. Security policies are a management mechanism for IPsec users. Thus, system users define, at a higher level, the nature of communication with other systems, without being concerned with low-level details (algorithms, keys, etc.). The actions defined in IPsec policies are three: Protecting traffic (Apply IPsec, Protect), Bypassing IPsec, or Discarding traffic. Security policies have a set of selectors to filter traffic and determine which policy it belongs to.

While security associations define parameters for secure communication, they do not ascertain when IPsec needs to be used and between which entities. These parameters are controlled through the Security Policy (SP) mechanism. SPs serve as a management framework for IPsec users, allowing higher-level specification of the nature of communication with other systems without delving into low-level details such as algorithms and keys. SPs use 'selectors' to identify what policy to apply to which communication. SPs also define what actions need to be taken for a given communication. The options for SP actions are:

- Protect (apply IPsec)

- Bypass (send/process the packet without applying IPsec)
- Discard (do not send/process the packet)

When implementing IPsec, one has to consider how to integrate it with the rest of the protocol stack. Three options exist:

- Native:
  IPsec is implemented within the protocol stack. Access to the stack code is required for integration.
- Bump-in-the-stack (BITS):
  IPsec is implemented ”below” a deployed version of the IP layer, between the network layer and the network interface. Access to the source code is not required for this choice. This approach is preferred when integrating IPsec in commercial closed-source products.
- Bump-in-the-Wire (BITW):
  IPsec is implemented in an external device rather than within the systems that utilize it. This method is common in military and some commercial applications. The embedded device acts as an ”assistant” to the host system and often has its own IP address.

3.1.1 Outbound Packet Processing

IPsec defines the flow of outbound packet processing as follows:

- Initially, information in the header of the outgoing packet is used to identify a policy that can be applied to this traffic. If no matching policy is found, the packet is discarded (Discard action).
- If a policy is found, it is applied. If the action is to Bypass, no further action is required by IPsec, and the packet continues its usual path through the stack. If the action is Discard, the packet is dropped. If the action is Protect, the SA pointed by the policy is queried, and the specific security parameters in the SA are used. If no SA is found and if automated SA creation is supported then the key manager (e.g. IKE) gets invoked. If the key manager fails or there is no support for automated SA creation, the action defaults to Discard.
- Finally, if the packet has not been discarded, it is sent.

The flowchart for the above process is presented in Figure 3.5.

3.1.2 Inbound Packet Processing

IPsec defines the flow of outbound packet processing as follows:

- Initially, information in the header of the outgoing packet is used to identify a policy that can be applied to this traffic. If no matching policy is found, the packet is discarded (Discard action).
- In the case where the policy action is Protect, the corresponding SA is searched using a Security Parameter Index (SPI) present in the incoming packet, and the security parameters in the SA are used. If no SA is found, the packet is discarded.
- After applying the SA, the processed packet is checked against its corresponding policy to verify its correct protection.

The above process is presented in flowchart form in Figure 3.6.

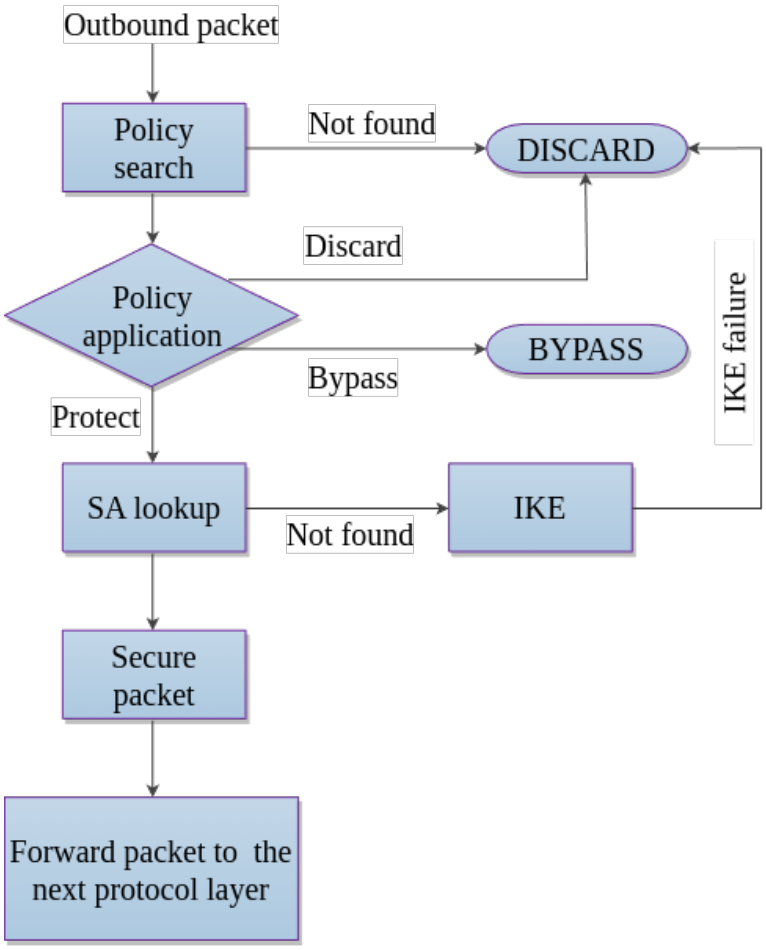

Figure 3.5: Outbound packet processing flowchart

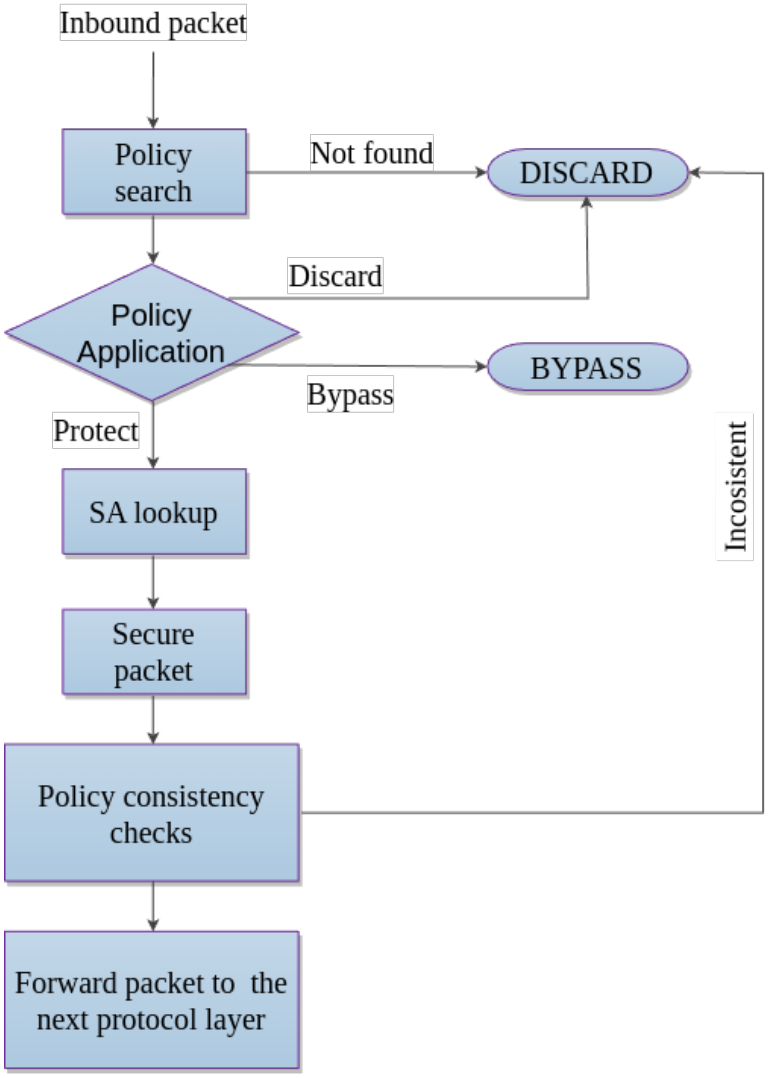

Figure 3.6: Inbound packet processing flowchart

3.1.3 Transport and Tunnel Modes

As previously stated, IPsec protocols can operate in either Transport mode or Tunnel mode. The difference in these two modes lies in which parts of the IP packet are considered the designated ”payload” of the IPsec packet. In Transport mode, only the data of the higher layer is considered as payload, while in Tunnel mode, the payload is extended to include the IP header. Figures 3.7 and 3.8 illustrate the packet format for Transport and Tunnel mode respectively, for both AH and ESP.

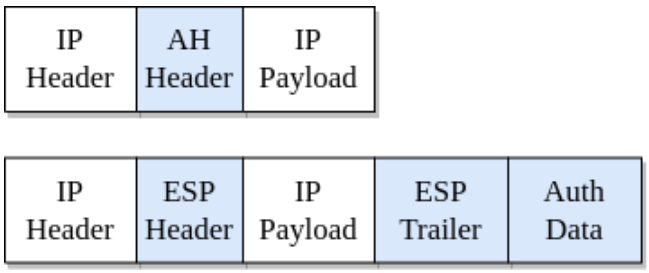

Figure 3.7: AH (up) and ESP (down) packet formats in Transport Mode

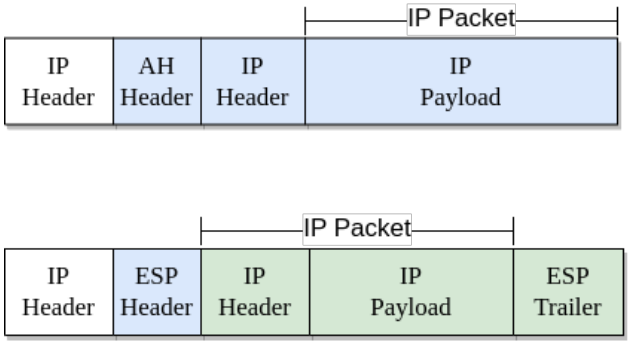

Figure 3.8: AH (up) and ESP (down) packet formats in Tunnel Mode

In transport mode, only the hosts execute IPsec. Initially, the appropriate protection is applied to the packet by the sender, and then the corresponding header is added to the data of the higher protocol. This is followed by the addition of the IP header, and finally, the packet is sent to the destination address. When it reaches the recipient, the packet is processed by IPsec, and the resulting payload is handed to the appropriate higher-layer protocol.

In tunnel mode, one or two Security Gateways (SGs) are involved. In the scenario of network-to-network tunnel mode, two SGs are used. The process starts with the packet being sent by the sender as normal. When the packet reaches the local SG, IPsec protection is applied to the entire IP packet. Then the security protocol header is added at the position right after the original IP header. Subsequently, a new IP header is added, containing the addresses of both the local and remote SGs, and finally packet is transmitted. When the packet reaches the remote SG, it is processed by IPsec, resulting in the original IP packet. Then, the IP packet is forwarded unmodified to the local network to reach its final recipient.

In the network-to-host scenario, where only a single SG is involved, two things change compared to the network-to-network scenario. Firstly, during transmission at the end of the communication where there is no SG, the host assumes SG’s processing responsibilities. In other words, the host is effectively acting as its own SG. Secondly, during packet transmission from the SG-equipped end, the SG only routes the packet without applying tunnel mode processing. Thus, in network-to-host scenarios, tunnel mode operates from the host to the SG, transitioning to transport mode from the SG-equipped end to the host.

3.2 Security Associations (SA)

As previously mentioned, a Security Association (SA) is a data structure containing information regarding the parameters governing a security session. SAs encapsulate parameters for each unidirectional session, leading to the utilization of two SAs per node in bidirectional scenarios — one for inbound and one for outbound traffic. These SAs are stored in a database known as the Security Association Database (SADB). Each SA is uniquely identified by a Security Parameter Index (SPI) which is used during search and retrieval of SAs. Both AH and ESP packets have a dedicated field within their headers designated for storing the SPI value corresponding to the applicable SA. SAs can be defined either manually by the system user or automatically through a negotiation protocol. Moreover, they have a lifetime limit, meaning they are invalidated after a certain period, and a new negotiation must start. An SA entry includes:

- The security protocol used (AH or ESP)
- The mode used (Transport or Tunnel)
- A Sequence Number for Anti-Replay
- Information about the authentication algorithm used in AH, such as the algorithm type, the key id, etc.
- Information about the encryption algorithm used in ESP, such as the algorithm's mode, the key id, the initialization vector, etc.
- Information about the data integrity algorithm used in ESP

3.3 Security Policy Database (SPD)

The defined Security Policies are stored in the Security Policy Database (SPD). SPD facilitates the search for the appropriate policy for a specific packet. The order in which policies are stored in the SPD also defines their priority of application. Each security policy is a data structure that holds information used for matching the policy to a specific type of traffic. The specific types of this information are called selectors and they include:

- lists of local IP addresses
- lists of remote IP addresses
- higher-layer protocol
- local port
- remote port

An SP entry includes:

- The type of policy to be applied. The three possible values are Protect, Bypass, and Discard. In the case of Protect, additional information is included in the SP:
  - the IPsec mode
  - for Tunnel Mode, the local and remote IP addresses of the tunnel
  - the security protocol to be applied
  - a prioritized list of algorithms that are allowed to be used. This information is useful in determining which algorithm will be used during an SA negotiation
  - the SPI of the SA that this policy can use

3.4 The Authentication Header (AH) Protocol

The AH protocol provides authentication, data integrity, and optional anti-replay protection for IP packets. Protection is provided for most of the IP packet and higher-layer protocol data, however, since some fields in the IP header can change during transit and thus their final values cannot be predicted, they are committed from AH protection.

AH defines a header format that provides information regarding the packet's protection. The format of the header is shown in Figure 3.9. The fields' purposes are:

- Next Header:
  The 8-bit standardized number (IANA) of the protocol that AH encapsulates. In tunnel mode, it can take only take the values 4 for IPv4 or 41 for IPv6.
- Payload Length:
  An 8-bit field denoting the size of the entire AH packet, expressed as a multiple of 32 bits and reduced by 2.
- Reserved:
  Reserved for future use. Mandated to be set to 0 by the sender, with no recipient verification requirement.
- Security Parameter Index (SPI):
  A 32-bit value pointing to the applicable SA.
- Sequence Number:
  An unsigned 32-bit integer denoting the packet's sequence number within the SA session. It is used by the anti-Replay mechanism. The field is mandatory, even when anti-replay is decativated.
- Authentication Data (or Integrity Check Value – ICV):
  A variable-sized field containing the outcome of the integrity/authentication algorithm. This field must be an integer multiple of 32 bits. Padding is included for proper alignment of the packet on a 32-bit boundary.

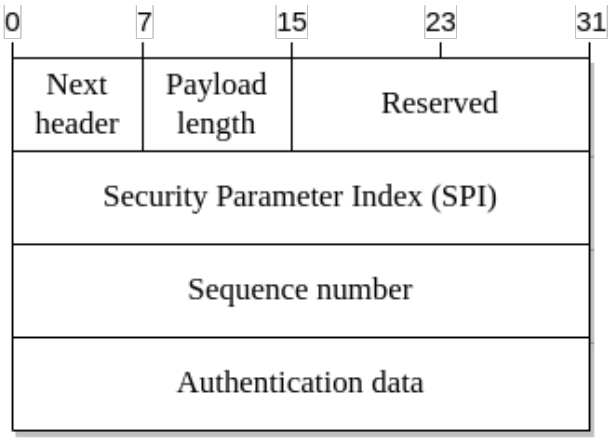

Figure 3.9: Structure of the AH protocol header

3.4.1 Outbound Packet Processing

The outbound packet processing for AH is as follows:

1. Finding SA:
   The AH protocol is applied when the SA corresponding to this traffic designates it as the protocol.

2. Sequence Number calculation:
   The sender starts the Sequence Number counter at 0 and increments it by 1 for each subsequent outgoing packet. Its value is stored in the SA. In the case of automatic SA creation and if the counter has returned to 0 after traversing all values, the SA expires and a new one must be created. If SA management is manual, then the counter restarts from 0.

3. ICV calculation:
   Calculation of the ICV involves the payload, the AH header, and a portion of the outer IP header. The value of the Authentication Data field is assumed to be all zeros for this calculation. From the IP header, only the fields that remain constant in the packet's path (immutable) are selected. The rest (mutable), their values are assumed as 0. The mutable fields of the IPv4 header are TOS, Flags, Fragment Offset, TTL, and Header Checksum. For IPv6, they are TOS, Flow Label, and Hop Limit. The calculated ICV value is placed in the Authentication Data field.

4. Padding:
   Finally, if necessary, padding is applied to ICV so that the entire packet is a multiple of 32 bits in IPv4 and a multiple of 64 bits in IPv6. Additional padding may be required to make the packet size a multiple of the block size of the algorithm used for authentication/integrity.

3.4.2 Inbound Packet Processing

The inbound packet processing for AH is as follows:

1. Finding SA:
   Firstly, SPI is used to locate an applicable SA. If no SA is found, the packet is discarded.

2. Anti-replay:
   If the anti-replay protection is enabled, the Sequence Number is checked. If found correct the processing continues; otherwise, the packet is discarded.

3. ICV verification:
   The receiver calculates the ICV in the same way as the sender. It then checks if the calculated value matches the value contained in the packet. If they do not match, then some alteration has occurred during the packet's transmission, and the packet is discarded.

3.5 The Encapsulation Security Payload (ESP) Protocol

The ESP protocol provides authentication, data integrity, confidentiality, and optionally anti-replay protection. The set of services it provides is determined by the corresponding SA. Authentication and integrity are always provided together, which makes three combinations of services possible:

- Confidentiality only
- Authentication-integrity only
- Confidentiality and authentication-integrity

ESP uses both a header and a trailer for the packet. The format of an ESP packet is shown in Figure 3.10. The fields' purposes are:

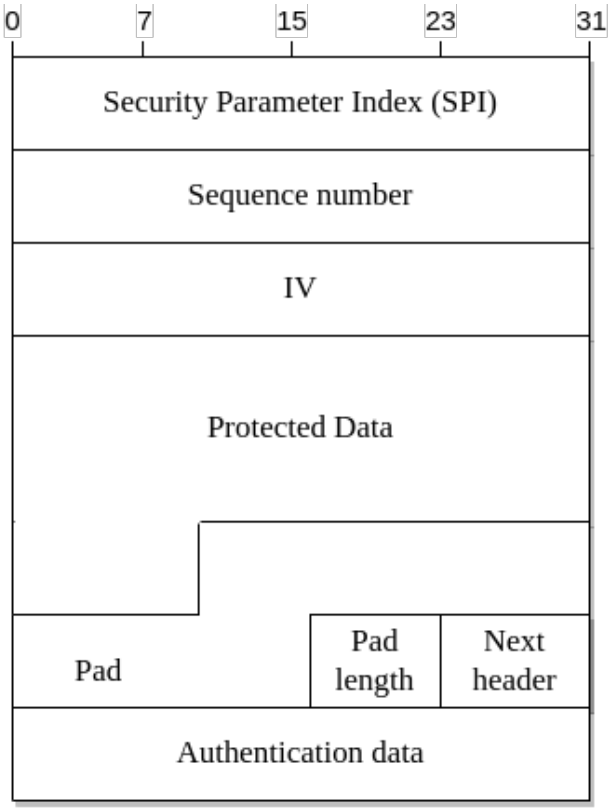

Figure 3.10: Structure of the ESP protocol header

- SPI:
  As with AH, this is the index for the corresponding SA.
- Sequence Number:
  As with AH, this is the incrementing number for the anti-replay protection.
- IV (Initialization Vector):
  This is an optional field. It is used when an IV is required by the encryption algorithm.
- Padding:
  This field has two purposes. If a block encryption algorithm is used, it aligns the packet to the block size. Otherwise, it aligns the packet to a multiple of 32 bits. The contents of the padding bytes are determined either by the encryption algorithm or the default padding is used. This default padding is the monotonically increasing integer sequence starting from 1.
- Pad Length:
  This field holds the length of the padding field in, expressed in bytes.
- Next Header:
  The IANA number of the higher-layer protocol.
- Authentication Data:
  A variable size field containing the result of the authentication/integrity algorithm. The size of this field is determined by the specific algorithm.

3.5.1 Outbound Packet Processing

During outbound packet processing, once the SA corresponding to the outgoing packet is found and if the SA specifies the use of the ESP protocol, the following actions are taken:

1. Padding is initially applied to the payload, so that the packet size becomes a multiple of 32 or 64 bits (for IPv4 and IPv6 respectively), or so that it becomes a multiple of the block size of the encryption algorithm. Additionally, the Payload Length and Next Header fields are inserted.
2. Next, if confidentiality is required, the encryption algorithm is executed on the payload. The corresponding key, and, if required, IV, are provided by the SA.
3. If an IV is required, it is placed at the start of the payload, following the ESP header.
4. The Sequence Number is then calculated and the ESP header is inserted.
5. Finally, if an authentication/integrity is required, the ICV is computed over the header and payload of the packet, and the result is placed in the trailer's Authentication Data field. The key and any other data needed by the algorithm are provided by the SA.

3.5.2 Inbound Packet Processing

To begin processing an incoming packet with ESP, the corresponding SA must first be found and must specify that ESP processing needs to take place. If no SA is found, the packet is discarded. Otherwise, the following steps are taken:

1. Initially, if anti-replay protection is used, the Sequence Number value is checked. If found correct, the packet processing continues. Otherwise, the packet is rejected.
2. If authentication/integrity is used, the ICV of the packet is computed and compared with the ICV value in the packet. The key is provided by the SA. If the ICV values match, the packet passes the authentication/integrity check; otherwise, it is rejected.
3. Next, if confidentiality is used, the decryption algorithm is executed using the key provided by the SA and, if available, the IV contained in the packet.
4. Finally, the padding is removed and the result is the plaintext payload.

3.6 Key Management

In IPsec, key management is performed using the Internet Key Exchange (IKE) protocol. IKE is a complex protocol based on the Oakley and ISAKMP protocols. Its purpose is the automated creation and management of Security Associations (SAs). Its main responsibilities include:

- Creating new SAs: This is achieved through a series of negotiations between the two communication ends regarding communication parameters such as authentication and encryption algorithms, keys, etc.
- Monitoring SA expiration and renegotiating them.

IKE is executed in two phases. First, ISAKMP is used to generate an ISAKMP Security Association which establishes a secure communication channel. Then, this secure channel is used to negotiate IPsec SAs, in pairs. The first phase has various execution modes, namely Main Mode, Aggressive Mode, and Base Mode. Each mode is defined as a series of messages consisting of various headers and payloads. The second phase has only one mode, Quick Mode. Additionally, there are other message exchanges for various IKE functions that are not part of these two phases, such as New Groups Mode, Unacknowledged Notification exchanges, and Acknowledged Notification exchanges.

Before IKE executes, the identities of the two ends must be authenticated. IKE uses three methods for authentication: a pre-shared secret key, digital signatures, or public key encryption.

4. System Overview

4.1 Design Goals

The present thesis produced an implementation of IPsec utilizing a hardware and software co-design approach. The target of the implementation is the FPGA board Virtex-5 ML505-ML509 Revision A [11], housing the Virtex-5 XC5VFX70T FPGA. For software execution, the Microblaze soft-core processor [12] is employed. Additionally, the system requires an Ethernet interface which is provided on the board along with the necessary drivers from Xilinx. Finally, an external DDR2 SDRAM memory is used for storing and executing the software. It should be noted that the system can be implemented using any board and processor, as long as an Ethernet interface, sufficient FPGA area, and enough memory for software execution are provided.

The library lwIP, an open-source implementation of the TCP/IP stack for embedded systems, is used. A port of lwIP to the specific FPGA board is provided by Xilinx. Additional modifications were made in the course of this thesis to incorporate IPsec.

For programming and debugging the system, a PC with an installation of Xilinx's ISE suite is used. Programming of the board is performed through a JTAG-to-USB cable, and debugging is performed through the board's serial port using a UART-to-USB cable.

Furthermore, a computer with the Linux operating system is used, where the IPsec Linux implementation "ipsec-tools" is installed. This computer is connected via Ethernet to the FPGA board, and both systems are configured to communicate using IPsec. This setup allows monitoring of incoming and outgoing packets. It also allows the verification of the system against a mainstream IPsec implementation.

4.2 Design Decisions

The system is implemented as an embedded system on an FPGA. This implementation environment has various constraints as well as advantages. Some of the constraints considerations include the available hardware area, the available peripherals, and the speed of the Microblaze soft-core processor. The main advantage includes the rapid development, implementation, and testing of hardware components.

Taking the above into consideration, various design choices are made aiming for the best possible utilization of the implementation environment's properties. These choices are explained in the following subsections.

4.2.1 Native Implementation

As mentioned in Chapter 3.1, there are three ways the integration of IPsec can be performed. The chosen approach for this implementation is the "Native" approach. This is because the software package lwIP (lightweight IP), is readily available for our target hardware. Additionally, this package is open-source, giving us the option to directly modify its source code.

While a ready implementation of the TCP/IP stack, removes the additional effort of implementing the necessary protocols for the system to function as a network system, it requires an in-depth study

of the lwIP code and specification in order to leverage its capabilities and establish proper interfacing between the two systems (lwIP and IPsec).

4.2.2 Hardware and Software

The available development environment enables the implementation of both hardware and software components. Certain modules can be selected for hardware implementation, while others can be executed in software. These subsystems can collaborate (hardware-software co-design), leveraging the benefits of each and mitigating their drawbacks.

In this thesis, the essential components of IPsec are implemented. Both AH and ESP protocols are integrated, covering both transport and tunnel modes. Additionally, both SADB and SPD are incorporated. The system can function as both a host and an SG. AH and ESP header processing, as well as the management of SPD and SADB databases, are implemented in software as part of the modified lwIP. The decision to develop these components in software is driven by the advantages of expedited development and debugging, facilitating better integration of IPsec with lwIP.

The cryptographic algorithms CBC-AES-128 and HMAC-SHA-1-96 are implemented in hardware. This decision is motivated by their role as latency and throughput bottlenecks, which are significantly alleviated by the faster speeds and parallel processing offered by hardware implementations.

Manual SA management is the only supported method. This choice is made due to the complexity and size of the IKE protocol, which is deemed beyond the scope of this thesis.

Additional features of the system include support for anti-replay protection and a generalized interface to cryptographic algorithms, simplifying the integration of different algorithms.

5. Hardware Components

5.1 Architecture

The system consists of a Microblaze processor, a DDR2 SDRAM memory, peripheral systems for Ethernet and UART interfacing, as well as units for cryptographic algorithms. All subsystems are connected to the processor via a PLB (Processor Local Bus) bus. Additional peripherals required during design include an interrupt controller and a timer. The interrupt controller manages interrupts from coming from the Ethernet, UART, and the timer. The timer is required by lwIP for time measurements. The architecture of the system is depicted in Figure 5.1.

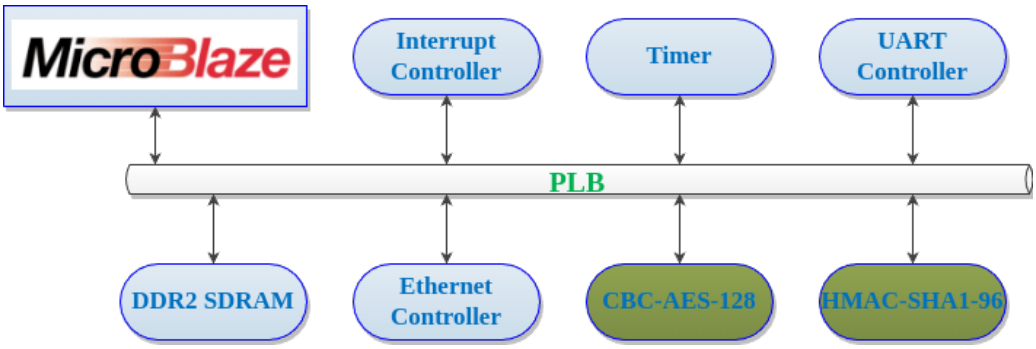

Figure 5.1: High-level system architecture

Each peripheral system is assigned its own address range when integrated with Microblaze (Address Mapped IO). The transfer of data to and from the memory of the cryptographic peripheral systems is handled by Microblaze (i.e., there is no Direct Memory Access - DMA system). This is because the system does not operate in multipacket mode which means that it is not capable of processing more than one packet simultaneously. This is because lwIP does not utilize threading when deployed in bare-metal, but instead relies on an operating system (e.g., Linux or XilKernel).

5.2 Custom Cryptographic IP Cores

5.2.1 CBC-AES-128

Algorithm Specification

The CBC-AES-128 consists of the block symmetric encryption algorithm AES-128 in CBC mode (see Figure 2.2). The number '128' indicates the size of the key in bits (the algorithms AES-196 and AES-256 are defined correspondingly). It gets a 128-bit block as input and outputs the encrypted block.

Internally, the algorithm consists of two operations: the encryption/decryption dataflow and the internal key generation dataflow. The encryption/decryption dataflow executes 10 iterations (rounds) of the basic encryption/decryption procedure when the key size is 128 bits (12 rounds for 196 bits

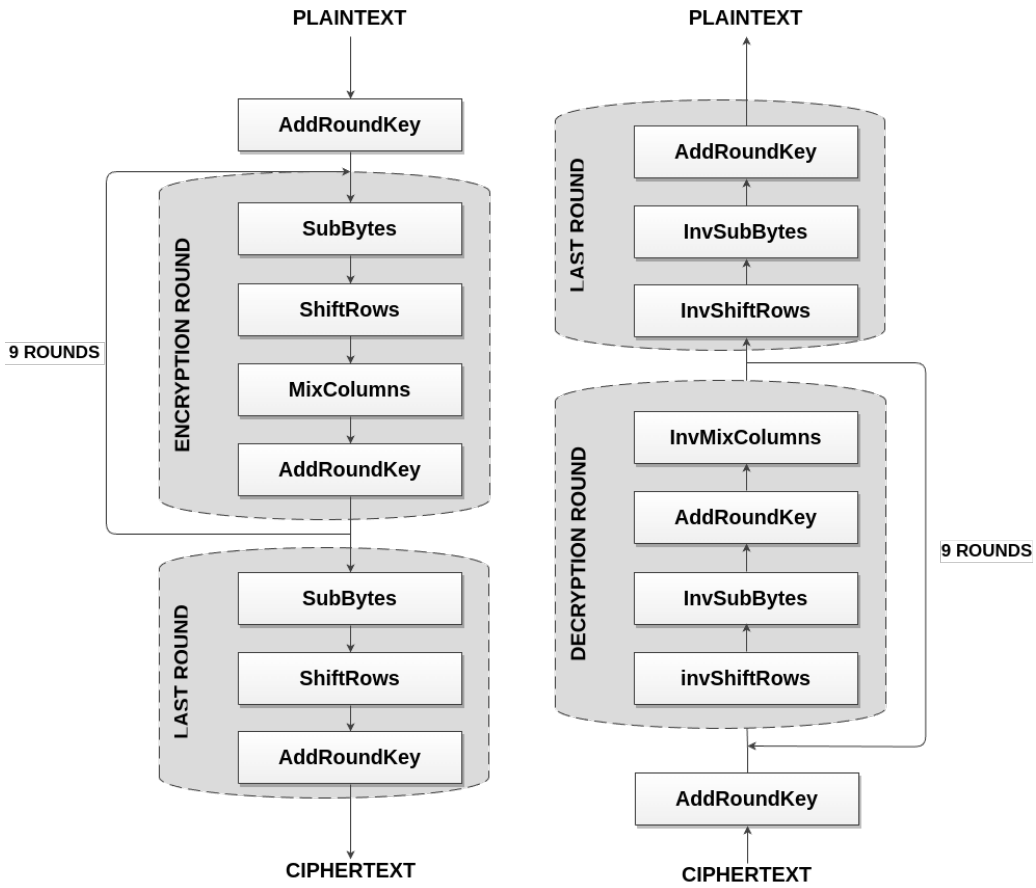

Figure 5.2: (left) AES encryption algorithm, (right) AES decryption algorithm

and 14 rounds for 256 bits). Similarly, the internal key generation involves a 10-round process. AES consists of four sub-functions:
For encryption:

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

For decryption:

- InvSubBytes
- InvShiftRows
- InvMixColumn
- InvAddRoundKey

Before elaborating on these functions, we must first introduce a convention. The algorithm's input is a block of 128 bits, which, after being subjected to several transformations, results in the output. During these transformations, the block is referred to as the ”state”.

The subscripts in Figure 5.3 denote the position of the byte in the block. The usage of the terms ”row” and ”column” should be interpreted with this convention in mind.

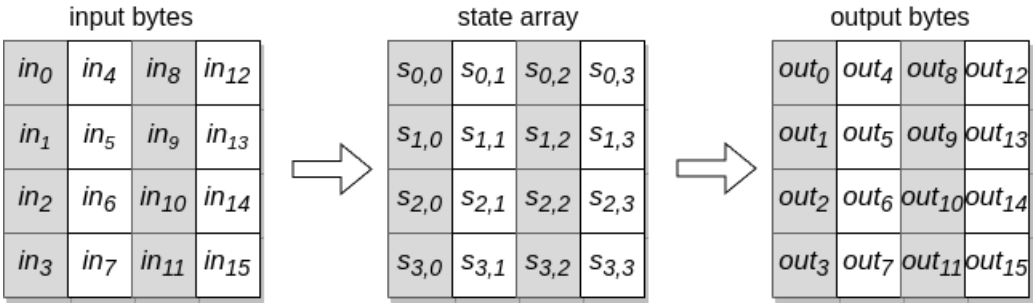

Figure 5.3: AES input bytes, state array, and output bytes

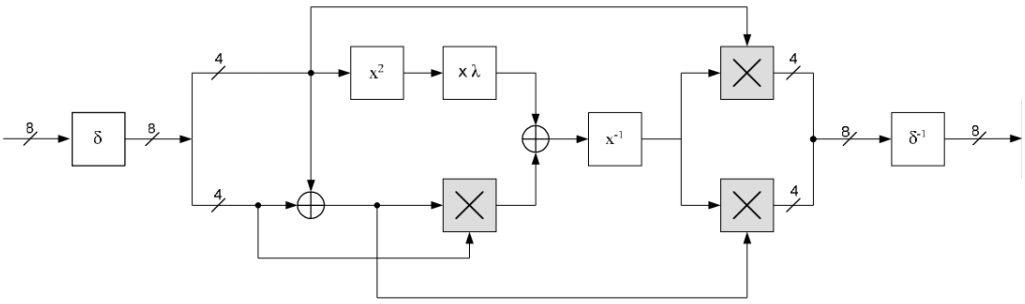

Figure 5.4: Multiplicative inverse in $GF(2^8)$

The SubBytes operation replaces the value of each byte in the block. The mapping of values is performed using the mathematical process. The calculation of the transformation consists of two parts. The first part is the calculation of the multiplicative inverse in the field $GF(2^8)$. The second part is the calculation of the inverse of the affine transformation. The calculation of the multiplicative inverse is presented in Figure 5.4. The individual operations involved are defined:

$$\delta \equiv \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{bmatrix} = \begin{bmatrix} x7 \oplus x5 \\ x7 \oplus x6 \oplus x4 \oplus x3 \oplus x2 \oplus x1 \\ x7 \oplus x5 \oplus x3 \oplus x2 \\ x7 \oplus x5 \oplus x3 \oplus x2 \oplus x1 \\ x7 \oplus x6 \oplus x2 \oplus x1 \\ x7 \oplus x4 \oplus x3 \oplus x2 \oplus x1 \\ x6 \oplus x4 \oplus x1 \\ x6 \oplus x1 \oplus x0 \end{bmatrix} \tag{5.1}$$

$$\delta^{-}1 \equiv \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{bmatrix} = \begin{bmatrix} x7 \oplus x6 \oplus x5 \oplus x1 \\ x6 \oplus x2 \\ x6 \oplus x5 \oplus x1 \\ x6 \oplus x5 \oplus x4 \oplus x2 \oplus x1 \\ x5 \oplus x4 \oplus x3 \oplus x2 \oplus x1 \\ x7 \oplus x4 \oplus x3 \oplus x2 \oplus x1 \\ x5 \oplus x4 \\ x6 \oplus x5 \oplus x4 \oplus x2 \oplus x0 \end{bmatrix} \tag{5.2}$$

$$x^2 \equiv \begin{bmatrix} x3 \\ x3 \oplus x2 \\ x2 \oplus x1 \\ x3 \oplus x1 \oplus x0 \end{bmatrix} \tag{5.3}$$

$$x\lambda \equiv \begin{bmatrix} x2 \oplus x0 \\ x3 \oplus x2 \oplus x1 \oplus x0 \\ x3 \\ x2 \end{bmatrix} \tag{5.4}$$

Additionally, the definition of multiplication is:

$$x \cdot y \equiv \begin{bmatrix} [(x3 \oplus x1)(y3 \oplus y1)] \oplus [(x2 \oplus x0)(y3 \oplus y1)] \oplus [(x3 \oplus x1)(y2 \oplus y0)] \oplus \\ \oplus (x1 \cdot y1) \oplus (x0 \cdot y1) \oplus (x1 \cdot y0) \\ [(x3 \oplus x1)(y3 \oplus y1)] \oplus [(x2 \oplus x0)(y2 \oplus y0)] \oplus (x1 \cdot y1) \oplus (x0 \cdot y0) \\ (x1 \cdot y1) \oplus (x0 \cdot y1) \oplus (x1 \cdot y0) \oplus (x3 \cdot y3) \oplus (x2 \cdot y3) \oplus (x3 \cdot y2) \oplus (x2 \cdot y2) \\ (x1 \cdot y1) \oplus (x0 \cdot y0) \oplus (x3 \cdot y3) \oplus (x2 \cdot y3) \oplus (x3 \cdot y2) \end{bmatrix} \tag{5.5}$$

Finally, we have the value mapping for the multiplicative inverse in $GF(2^4)$:

$$\begin{aligned} &x^{-1}(0000) = 0000, x^{-1}(0001) = 0001, x^{-1}(0010) = 0011, x^{-1}(0011) = 0010 \\ &x^{-1}(0100) = 1111, x^{-1}(0101) = 1100, x^{-1}(0110) = 1001, x^{-1}(0111) = 1011 \\ &x^{-1}(1000) = 1010, x^{-1}(1001) = 0110, x^{-1}(1010) = 1000, x^{-1}(1011) = 0111 \\ &x^{-1}(1100) = 0101, x^{-1}(1101) = 1110, x^{-1}(1110) = 1101, x^{-1}(1111) = 0100 \end{aligned} \tag{5.6}$$

The calculation of the inverse affine transformation takes one byte as input and computes the following operation:

$$AT^{-1} \equiv \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{5.7}$$

where x_n is the n$^{\text{th}}$ bit of the input byte. ShiftRows performs a one-byte rotation of the 2$^{\text{nd}}$ row, a two-byte rotation of the 3$^{\text{rd}}$ row, and a three-byte rotation of the 4$^{\text{th}}$ row. MixColumns multiplies each column by the matrix:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}_{10} \tag{5.8}$$

and the result replaces the initial column. Finally, the AddRoundKey performs bitwise XOR of the block and the round key. The decryption operations execute the inverse process. Thus, the InvSubBytes performs the inverse substitution of the value of each byte. To calculate the inverse value, the byte first undergoes an affine transformation:

$$AT \equiv \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{5.9}$$

and then through an inverse multiplication in $GF(2^8)$. The InvShiftRows performs the same rotations in the opposite direction (right), and the InvMixColumns multiplies by the inverse matrix of MixColumns:

$$\begin{bmatrix} 14 & 11 & 13 & 09 \\ 09 & 14 & 11 & 13 \\ 13 & 09 & 14 & 11 \\ 11 & 13 & 09 & 14 \end{bmatrix}_{10} \tag{5.10}$$

and the InvAddRoundKey is the same as the AddRoundKey.

In the key generation process, a matrix of constant values, called $Rcon$, is involved which is defined as:

$$Rcon \equiv \begin{bmatrix} 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B & 36 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}_{16} \tag{5.11}$$

Each column of this matrix participates in the corresponding round key generation. (the 1st in the 1st round, the 2nd in the 2nd, etc.). The key generation process is as follows: To generate the first column of the new key, the last column of the previous key is selected, and a RotWord (see Figure 5.5) is performed. Then, each byte of the column undergoes the SubBytes process, and finally, this column, along with the 1st column of the previous key and the corresponding column of the Rcon matrix, undergo bitwise XOR. The resulting column is the 1st column of the new key.

The value of the 2nd column is the result of the bitwise XOR of the 2nd column of the previous key with the 1st column of the new key, the 3rd column's value is the result of the bitwise XOR of the 3rd column of the previous key with the 2nd column of the new key, and the 4th column's value is the result

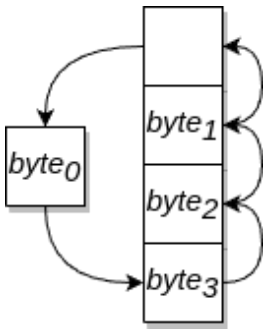

Figure 5.5: The RotWord operation

of the bitwise XOR of the 4th column of the previous key with the 3rd column of the new key. This process is repeated 10 times to produce 10 keys, one for each encryption round. The keys generated are passed to the AddRoundKey of the corresponding round. It is noteworthy that in decryption, the keys are provided in the reverse order compared to encryption.

Design Decisions

Various parameters need to be taken into account, before commencing the design and implementation of this peripheral. Since the peripheral will operate as a slave of a processor, the interfaces between them, the communication protocol, and the individual technical details must be first defined.

In a processing system, the processor undertakes various tasks. Therefore, peripherals should be designed to operate with waiting states. In the case of CBC-AES-128, the peripheral should be in a waiting state until the processor provides it with the next input or command. Additionally, when an operation is completed and there is output available for the processor, the peripheral should have a way to notify the processor and wait until the processor consumes the peripheral's output.

For the communication protocol between the peripheral and the processor, both control signals driven by the processor and status signals driven by the peripheral are needed. These signals are designed, and their behavior is defined in a way that facilitates straightforward hardware design and straightforward management from the software side.

An important design decision is choosing the data transfer method between the peripheral and the memory. Many options exist, such as DMA, the peripheral switching between master and slave modes in the main bus, or direct access to the processor's registers (a Microblaze feature provided by Xilinx, called FSL, Fast Simplex Link). Ultimately, because the system is not multipacket, the processor does not process many packets simultaneously, and therefore there is no benefit in a peripheral taking over the data transfer, as during the data transfer, the processor will be blocked in a waiting state. Thus, the design decision is that the data transfer will be performed by the processor.

Finally, regarding the means of the peripheral notifying processor, there exist two options, polling and using interrupts. The polling method is implemented since due to the lack of multipacketing, the processor would be inactive while waiting for an interrupt.

Hardware Design

The hardware design is organized hierarchically. At the lowest level, the internal operations of AES (ShiftRows, SubBytes, MixColumns, AddRoundKey) are implemented. These are used to implement the unit that executes one round of AES and another unit that executes one round of the key expansion. The complete key expansion unit is then implemented. Finally, at the top level, these subcomponents are integrated to implement the complete AES in CBC mode. Next, we elaborate on the hardware designs, in a bottom-up approach.

The implementation of ShiftRows and InvShiftRows is simply a permutation of their input bytes. The unit implementing the SBOX and its inverse (InvSBOX), receives a signal called EncDec which indicates whether the unit will operate in encryption or decryption mode. This design choice is made with resource reuse of the inverse multiplication $GF(2^8)$ in mind in order to optimize the area utilization of these components. The affine transform, inverse affine transform, and multiplicative inverse are designed using logic gates that directly implement their equations. Figure 5.6 presents the block diagram of the unit. Due to the combined implementation of SBOX and InvSBOX, the implementation of SubBytes and InvSubBytes is also combined in a unit that accepts an EncDec signal that is passed down to these operations. The block diagram of the unit of SubBytes and InvSubBytes is shown in Figure 5.7.

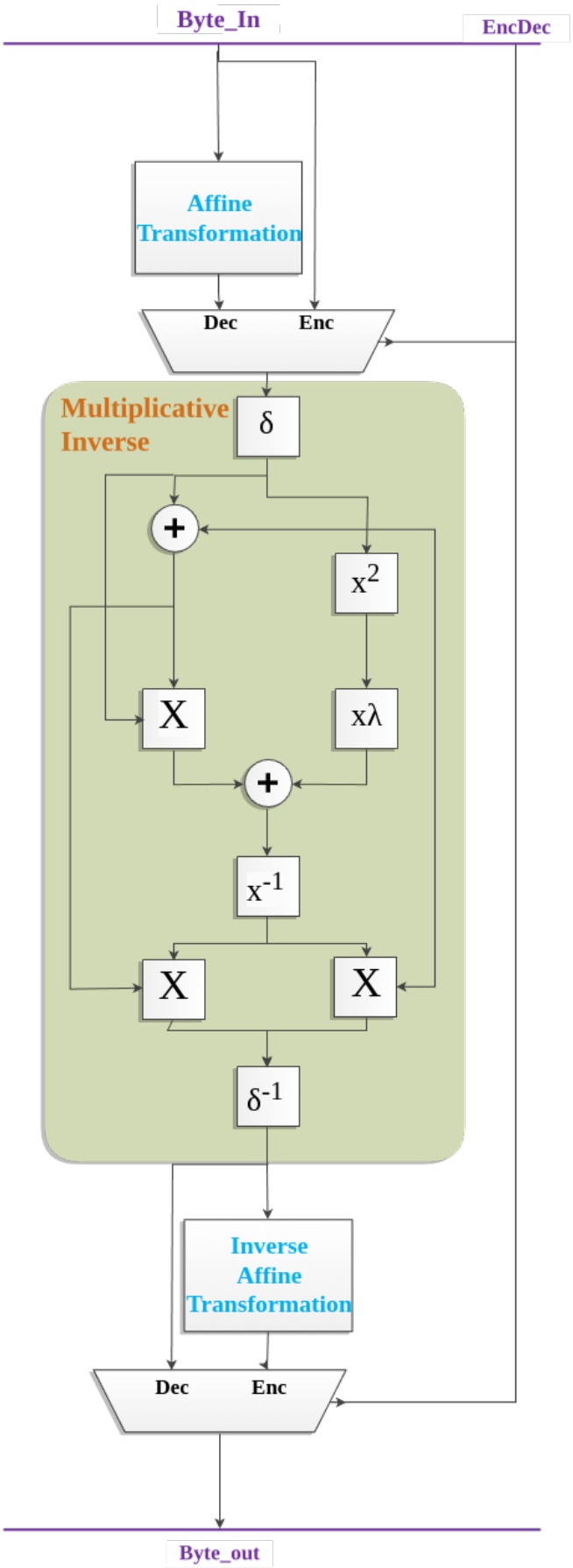

Figure 5.6: Combined implementation of SBOX and InvSBOX

The operations MixColumns and InvMixColumns are implemented in a combined unit. The

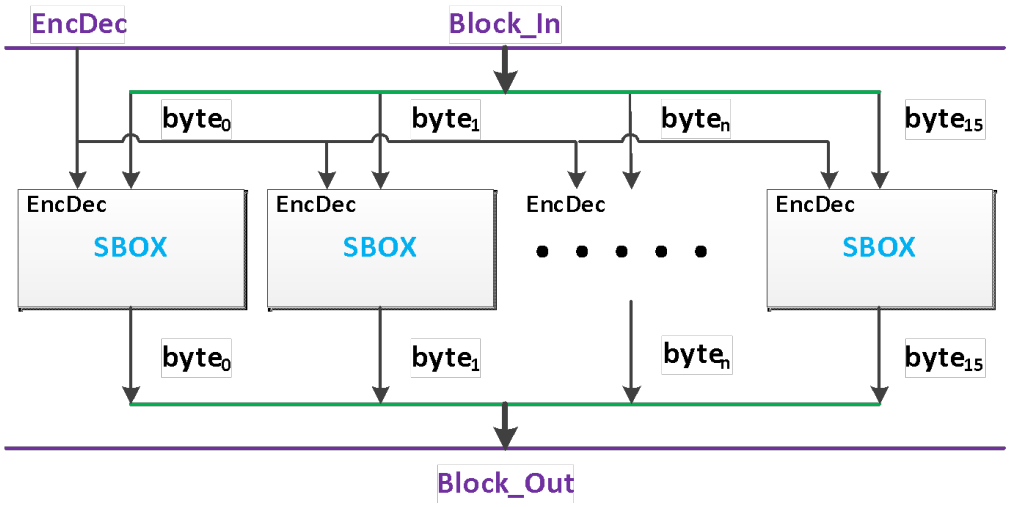

Figure 5.7: Combined implementation of SubBytes and InvSubBytes

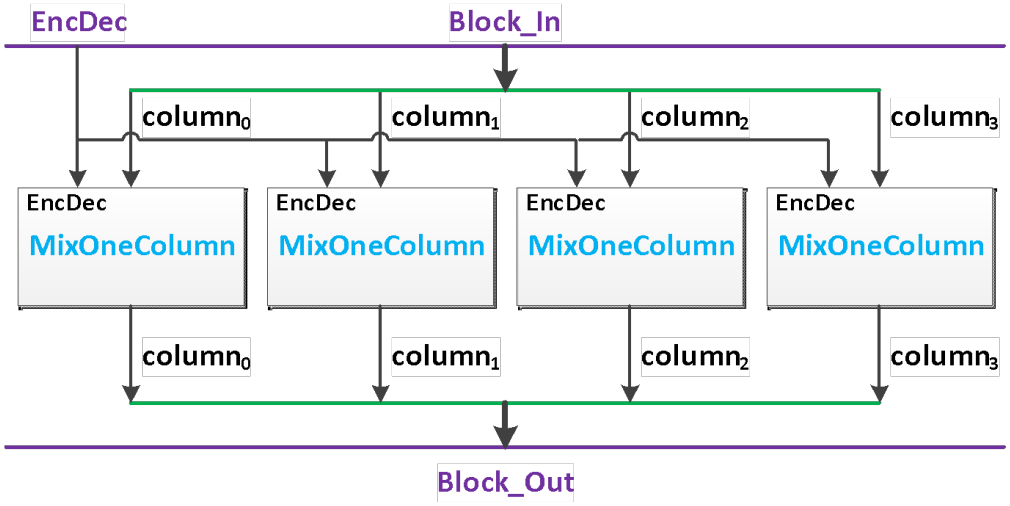

Figure 5.8: Combined implementation of MixColumns and InvMixColumns

selection of the operation is specified using an EncDec signal. Because each column is processed independently of other columns, 4 subunits named MixOneColumn, one per column, are implemented to exploit parallelism and thus reduce latency. The block diagram of the combined MixColumns and InvMixColumns unit is presented in Figure 5.8.

In each MixOneColumn, the column is multiplied by the matrix 5.8 during encryption and by the matrix 5.10 during decryption. We observe that each byte of the column must be multiplied by the numbers 02 and 03 for encryption and by 14, 11, 13, and 09 for decryption. Thus, a unit named XTimes is created, which takes one byte as input and outputs its multiples of 2, 3, 9, 11, 13, and 14. The appropriate additions of the multiples are performed, and the final results are placed in the corresponding bytes of the output column. The implementation of MixOneColumn is presented in Figure 5.9. In this figure, to distinguish the outputs of XTimes, a suffix with the byte number is used (the signal x2_0 means twice $byte_0$, x14_3 means 14 times $byte_3$, and so on).

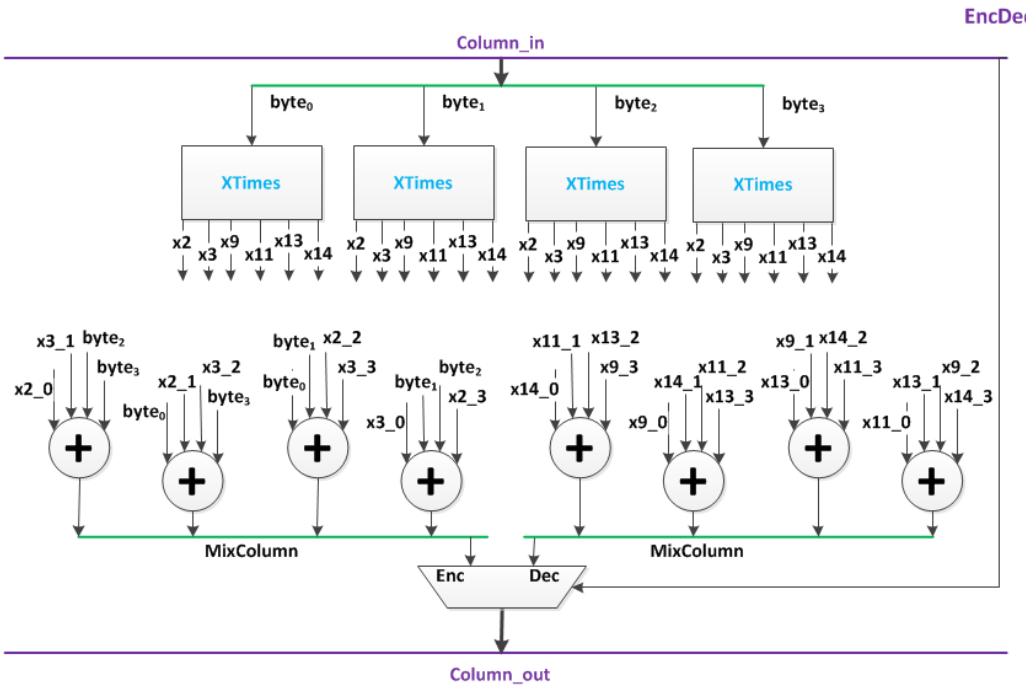

Figure 5.9: Implementation of MixOneColumn

The implementation of XTimes is presented in Figure 5.10. The individual operations are characterized by the following equations:

$$
\begin{aligned}
3x &= 2x \oplus x, \\
4x &= 2 \cdot 2x, \\
8x &= 2 \cdot 4x, \\
9x &= x \oplus 8x, \\
10x &= 2x \oplus 8x, \\
11x &= 10x \oplus x, \\
12x &= 10x \oplus 2x, \\
13x &= 12x \oplus x, \\
14x &= 12x \oplus 2x
\end{aligned}
\tag{5.12}
$$

In Figure 5.11, the unit for computing the double of XTimes, named XTimes2 is illustrated. This operation takes place in the $GF(2^8)$ field and is defined as follows: For the input byte, a left shift is performed first. If the MSbit of the input byte is 0, then the shift does not overflow, and the output is given by the shifted byte. Otherwise, if there is overflow, the value $1B_{16}$ must be added (via XOR) to the shifted byte.

The last unit in an AES round is the AddRoundKey, which is simply a bitwise XOR between the block and the current round key. The key is provided as input, and ultimately, AddRoundKey is a bitwise XOR of length 128 bits.

Finally, combining the above units, a unit for computing a round of AES-128 is implemented. Its inputs are the input block, the key of the current round, the EncDec signal for selecting encryption or decryption mode, and a LastRound signal indicating if the operation is currently executing the last AES round. LastRound is needed because the last round of AES-128 differs from the preceding 9

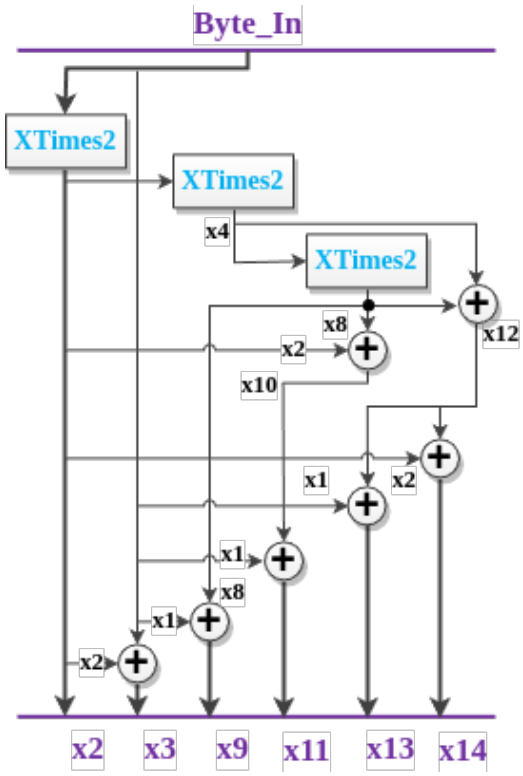

Figure 5.10: Implementation of XTimes

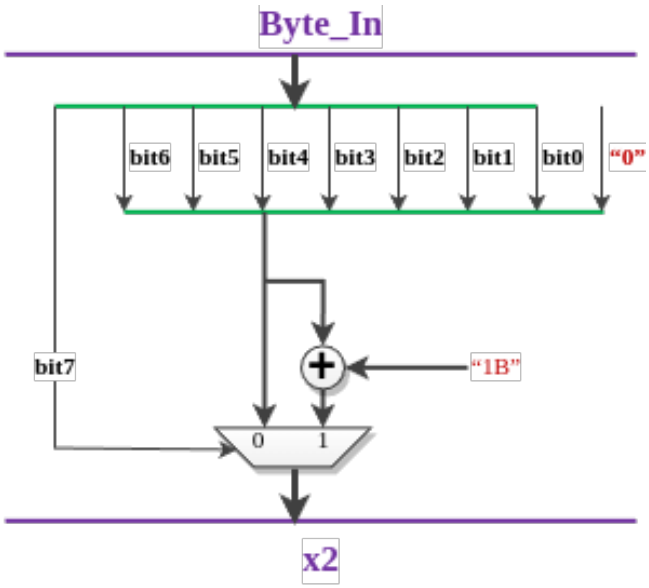

Figure 5.11: Implementation of XTimes2

rounds. The output of the unit is the processed block. The block diagram of the design is presented in Figure 5.12. The differences between encryption and decryption are (as shown in Figure 5.2):

1. Execution of ShiftRows versus execution of InvShiftRows
2. Execution of SubBytes versus execution of InvSubBytes
3. Execution of MixColumns versus execution of InvMixColumns
4. Execution of AddRoundKey after MixColumns in encryption, and vice versa for decryption.

The first difference is implemented using a MUX, which selects between the output of ShiftRows or InvShiftRows based on the EncDec signal. The second and third differences are already implemented within the common units of SubBytes and MixColumns respectively, using the EncDec signal. The fourth difference is implemented using two AddRoundKey units, one placed before MixColumns and one placed after. When in encryption mode, the first AddRoundKey is bypassed using a MUX, while the second one is used. In decryption, the second one is bypassed with a MUX while the first one is used.

The difference in the final rounds of encryption and decryption from the other 9 is (as shown in Figure 5.2) that the MixColumns and InvMixColumns operations are not performed respectively. Because these two operations are implemented in one unit, only one bypass MUX is needed with the LastRound signal as the selection signal.

For the calculation of the round keys, first, a unit named KeyScheduleRound is implemented for computing the next round key. The key calculation is identical for encryption and decryption, with the only difference being that during decryption, the keys are provided to the AES rounds in the reverse order from the order in which they are calculated. This difference is managed at a higher level of the design. The block diagram of the KeyScheduleRound is presented in Figure 5.13.

The KeyScheduleRound unit takes as inputs the key of the previous round and the Rcon value corresponding to the current round and outputs the key generated for the current round. In the design, 4 SBOX units are used, each for each of the 4 bytes of the last column of the input key. Because the inverse operation InvSBOX is not needed, a separate SBOX unit is implemented, which performs only the calculations for multiplicative inverse and inverse affine transform, to save on area utilization.

The computations executed are as follows: first, the substitution through the SBOX of the last column of the input key is performed, and the result undergoes a RotWord, which is simply a permutation of the bytes. After RotWord, a bitwise XOR with the value of Rcon is performed. From Figure 5.11, we observe that the last 3 bytes of each column of Rcon are zero, so they do not affect the bitwise XOR. Thus, only the first byte of Rcon is needed as input, and a bitwise XOR of 8 bits instead of 32 bits is sufficient. The first column of the output key is the processed last column of the input. The second output column is calculated with a bitwise XOR between the first input column and the processed last input column, the third with a bitwise XOR of the second output column with the second input column, and the fourth with the third output and the third input column.

Using KeyScheduleRound, the unit for computing all round keys, named KeyScheduler, is implemented. Its block diagram is presented in Figure 5.14. Its inputs are the initial key, a round counter indicating the current round, and the signal FirstRound indicating that we are in the first round. The output provides the current round key at any given time. The unit contains a KeyScheduleRound subunit and an Rcon subunit which takes as input the round counter and outputs the first byte of the corresponding column of Rcon (since the others are $'00'$) and feeds this value to the KeyScheduleRound.

The value of the first derived key depends on the value of the initial key while the value of any other derived key depends on the directly preceding derived key. The FirstRound signal helps differentiate between the two scenarios. It is used to route either the initial key or the preceding derived key through the feedback loop back as input to the KeyScheduleRound unit.

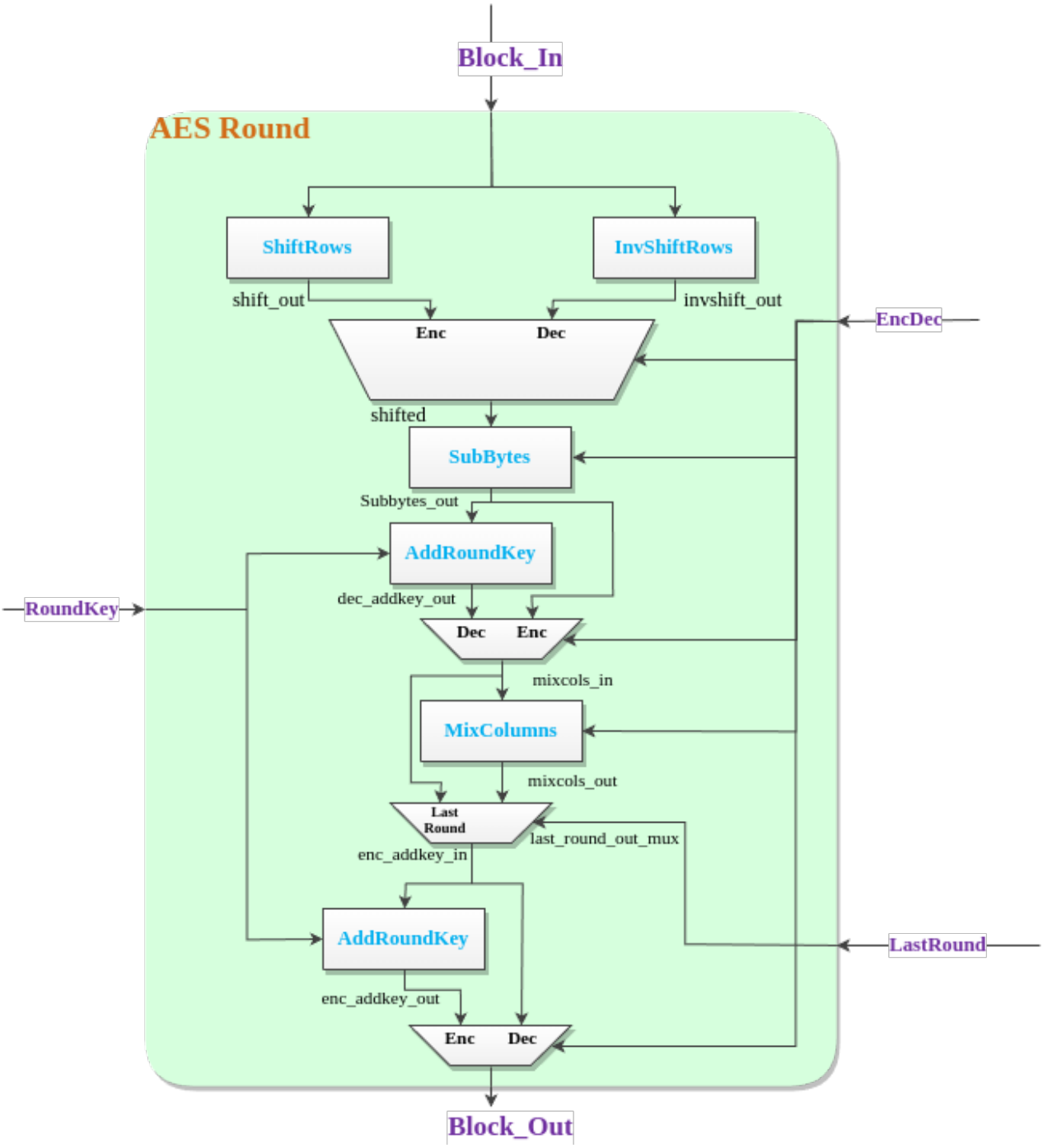

Figure 5.12: Implementation of one round of AES-128

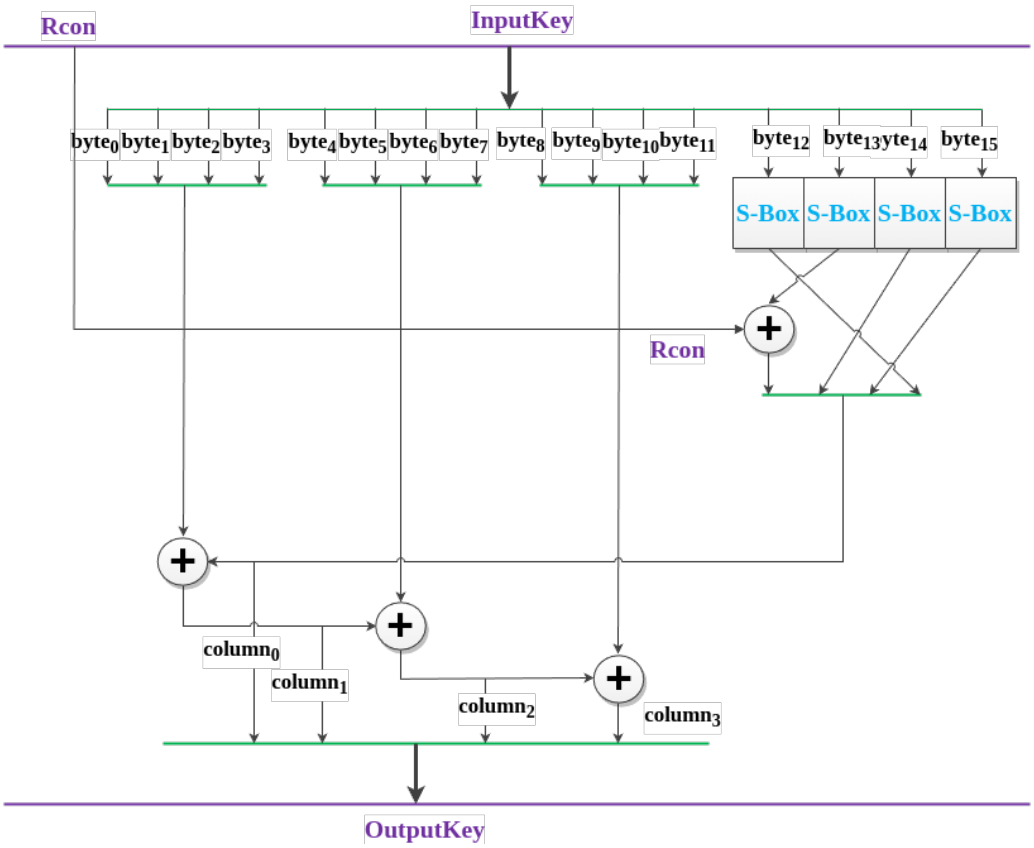

Figure 5.13: Implementation of one round of AES-128 Key Scheduler

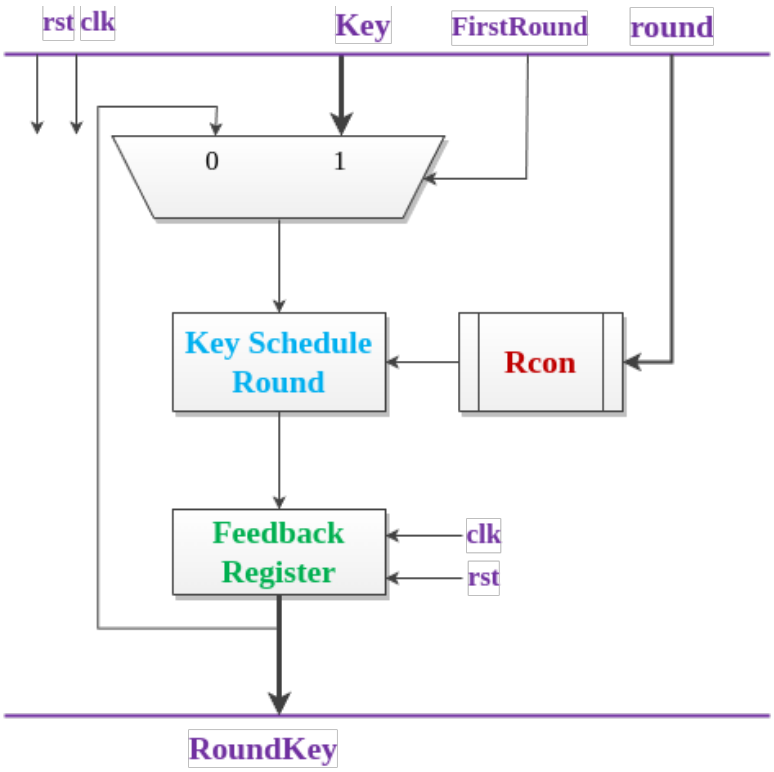

Figure 5.14: Implementation of the AES-128 Key Scheduler

Finally, the peripheral of CBC-AES-128 is implemented. Its block diagram is presented in Figure 5.15. Its inputs are:

- Block_in: the current message block
- IV: the initialization vector
- Key: the cryptographic key
- Start_new: a control signal denoting an instruction for a new encryption/decryption operation
- Next_block: a control signal indicating that the next input block is present at the Block_in
- EncDec: a control signal for selecting between encryption and decryption operations
- clk: the system's clock signal
- rst: the system's reset signal

As outputs, the peripheral provides the output block and a control signal named 'ready' indicating the end of block processing, i.e., that the output block is ready.

In Figure 5.15 the colored area on the left outlines the parts that implement the CBC operation. The part in the middle is the AES operation, and the part on the right is the key generation operation. In the bottom right, the peripheral's control circuit is presented, alongside its input and output signals.

The CBC implementation uses two bitwise XORs, one applied at the input block before it is passed to the AES operation and one applied at the output of the AES operation. The first XOR gets selected by a MUX when in encryption mode, while the second XOR is selected in decryption mode. The second input of both XORs is determined based on the combination of the current block number and if the operation is encryption or decryption. When we are processing the first block, the IV is provided to the XORs. For any other block, if we are in encryption mode, the value of the previous AES output is provided, while if in decryption mode, the value provided is the previous input block. These options are implemented with the appropriate placement of registers, MUXes, and the needed control signals managed by the control circuit.

The AES part first needs to perform an AddRoundKey to the input before continuing with the 10-round processing, as shown in Figure 5.2. The 10 AES rounds are implemented by using a feedback loop around the AES Round unit. The signal FirstRound selects which value will be used in the AES Round, either the result of the AddRoundKey or the previous output via feedback.

The core of the key generation operation is a KeyScheduler unit. When in encryption mode, where the keys must be provided in the order they are generated, the KeyScheduler directly provides its output to the AES Round, and the initial AddRoundKey is given the initial key from the input. In decryption mode, where the keys must be provided in the reverse order of the order they get generated, the last key must be given to the initial AddRoundKey, which requires the computation and storing of all the keys in advance. For this purpose, 11 registers are used as the key storage. The first 10 registers are populated by the KeyScheduler starting from the last register and in reverse order. This way the keys are stored in reverse order and are mapped correctly with their respective rounds. The initial input key is placed in the last (11^{th}) register. Finally, the first register is connected directly to the AddRoundKey to provide the last derived key, while the other registers are selected by a MUX based on the round counter to be routed to the AES Round.

The control circuit receives the signals Start_new and Next_Block and provides the output signal ready and the internal control signals set_iv (used for IV selection), LastRound, FirstRound, and Round_counter, which is the counter indicating the current round. The control circuit is described by the FSM shown in Figure 5.16:

Upon each reset, the FSM transitions to the initial Start state, where it waits for a start_new signal. When start_new arrives and depending on the value of the EncDec signal, it transitions to

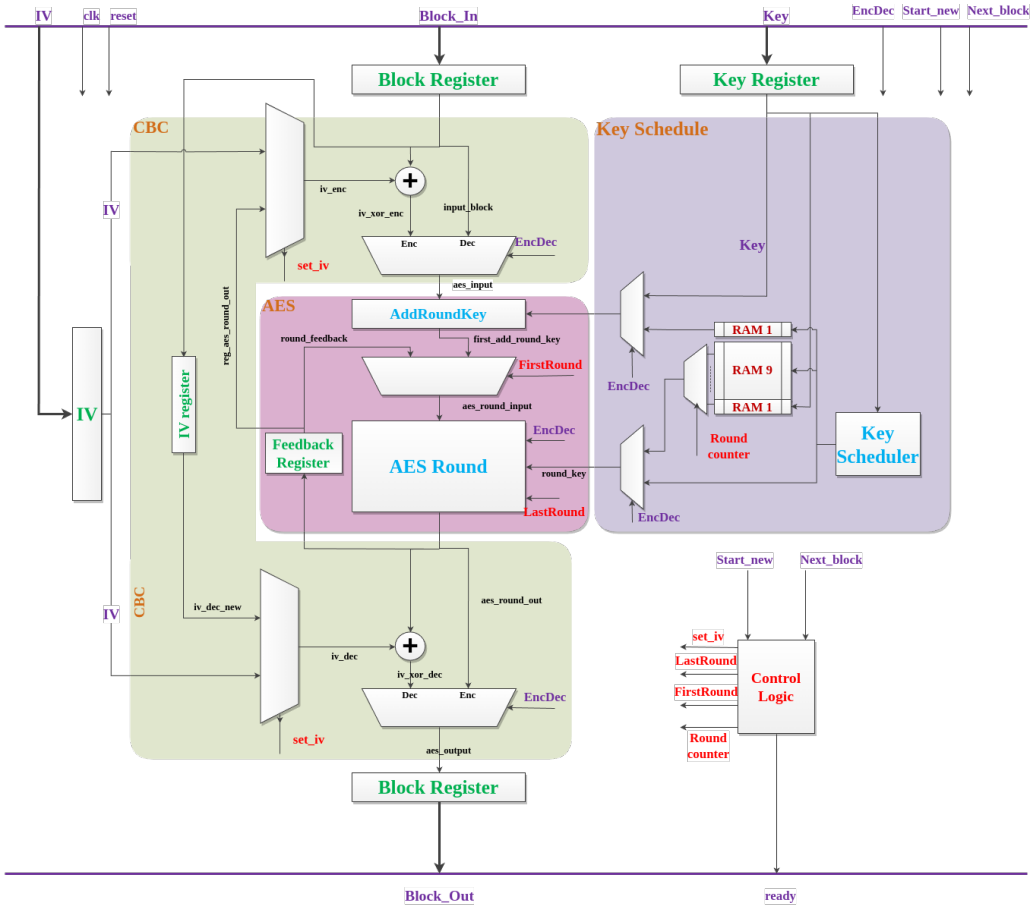

Figure 5.15: Implementation of CBC-AES-128

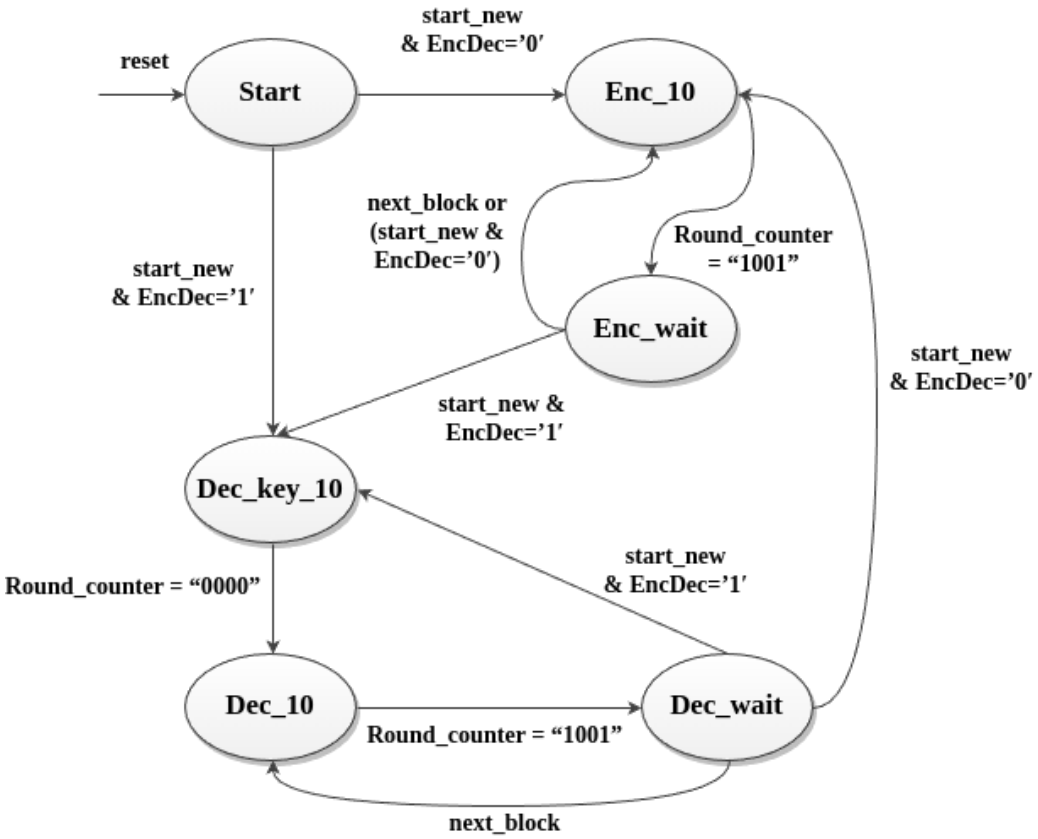

Figure 5.16: The control FSM of CBC-AES-128

either encryption mode (EncDec='0') or decryption mode (EncDec='1'). Before a start_new signal is issued, the key, the IV, and the first block of the message must be already provided at the module's inputs.

Encryption mode operates using two states. In the state Enc_10, the 10 encryption rounds of AES are executed, and when finished it transitions to the Enc_Wait state, where the peripheral waits. The transition between the two is implemented using a comparator that compares the Round_Counter with the value "1001" (counting starts from 0, so a total of 10 rounds are counted). While in Enc_Wait state, the peripheral waits for either the next block of the same message, indicated by the next_block signal, or the start of a new message indicated by the start_new signal. When a next_block is received, the FSM returns back to Enc_10 to process the new block. If start_new arrives, the next FSM state depends on the value of EncDec. If EncDec is '0', the FSM transitions to the Enc_10 state in order to encrypt the first block of the new message, while if EncDec is '1', the FSM transitions to the Dec_Key_10 state, which is the first stage of the decryption operation.

In decryption mode, we have three states: Dec_Key_10, where all the keys first are generated and stored, Dec_10, where the 10 decryption rounds of AES are executed, and Dec_Wait, which is a waiting state that serves a similar purpose to Dec_Wait. The keys must be generated once for every new message, thus Dec_Key_10 only takes place once at the start of the decryption process and there is no need to regenerate them on the arrival of every new block since the keys are kept in the key registers.

In Dec_Key_10, the Round_Counter starts from the value "1001" so that the appropriate key registers are selected and traversed in reverse order. When Round_Counter reaches 0, the decryption of the first block can commence by transitioning to state Dec_10. The Round_Counter gets increased in every round, and, after executing the 10 decryption rounds, Round_Counter is "1001". At this point the FSM transitions to the waiting state Dec_Wait. Dec_Wait is identical to Enc_Wait except from the fact that state transitions differ.

5.2.2 HMAC-SHA1-96

Algorithm Specification

HMAC-SHA1-96 is a MAC algorithm that utilizes the cryptographic hash algorithm SHA1. The number 96 in the name indicates how many bits of the output are used. The mathematical formulation of HMAC is:

$$HMAC(K, text)_t = H[(K_0 \oplus opad)||H((K_0 \oplus ipad)||text)]_t \tag{5.13}$$

where:

- $text$ represents the input text,
- H is the hash function used (in this case it is SHA1),
- K is the key,
- K_0 is the key after preprocessing,
- $ipad$ is the repetition of the byte $[36]_{16}$, with a total size equal to the block size of the hash function,
- $opad$ is the repetition of the byte $[5C]_{16}$, with a total size equal to the block size of the hash function,
- $||$ denotes concatenation,

- t is the size of the received output.

Figure 5.17, presents the HMAC operation schematically.

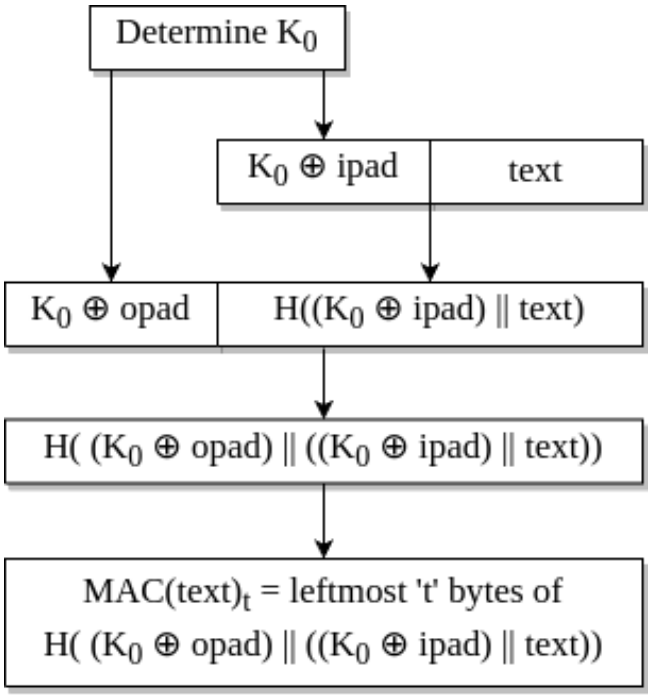

Figure 5.17: The HMAC algorithm

HMAC has a preprocessing phase, where the value K_0 gets generated from the key K. The process differs depending on the size of key K and the block size of H. When the size of K matches the block size of H, K simply takes the value of K_0. If the size of K is smaller than the size of H, a padding of '0' bits is applied, while if larger, K is first transformed through H to produce K_0.

The SHA1 algorithm has an output size of 160 bits. It divides its input into 512-bit blocks and padding is applied for inputs whose size is not a multiple of 512 bits. It executes 80 iterations to process each input block. The algorithm is described in Algorithm 1, where N is the number of input blocks, M^i is the i_{th} input block and:

$$f_t(x,y,z) \equiv \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ ROTL^1(W_(t-3) \oplus W_(t-8) \oplus W_(t-14) \oplus W_(t-16)) & 16 \leq t \leq 79 \end{cases} \tag{5.14}$$

$$K_t \equiv \begin{cases} 5a827999 & 0 \leq t \leq 19 \\ 6ed9eba1 & 20 \leq t \leq 39 \\ 8f1bbcdc & 40 \leq t \leq 59 \\ ca62c1d6 & 60 \leq t \leq 79 \end{cases} \tag{5.15}$$

SHA1 input padding includes two fixed fields: its first bit which takes the value '1', and the last 64 bits, which contain the original message's size expressed in bits in binary representation. The remaining bits in between are set to '0'. The total number of these '0's is chosen so that the original message together with the padding is a multiple of 512 bits.

Design Decisions

The design decisions for the peripheral HMAC-SHA1-96 are the same as those for CBC-AES-128.

Hardware Design

Algorithm 1 is implemented in hardware. Firstly, the sha1_core unit is implemented, which handles the processing of the 80 iterations of SHA1 for one block. Using this unit, the sha1 unit is designed, which computes the SHA1 hash of a complete input message. Finally, using the sha1 unit, the HMAC-SHA1-96 algorithm is implemented. The implementation assumes that the input is a multiple of 512 bits, and the padding, if needed, has been added beforehand. It is also assumed that any preprocessing of the key, if necessary, has already been done. All units are explained in detail below.

Algorithm 1 HMAC Preprocessing

1: **procedure** HMAC_Preprocessing(N, M)
▷ Initialize H
2: $H_0^{(0)} \leftarrow \texttt{67452301}_{16}$
3: $H_1^{(0)} \leftarrow \texttt{EFCDAB89}_{16}$
4: $H_2^{(0)} \leftarrow \texttt{98BADCFE}_{16}$
5: $H_3^{(0)} \leftarrow \texttt{10325476}_{16}$
6: $H_4^{(0)} \leftarrow \texttt{C3D2E1F0}_{16}$
7:
8: **for** $1 \leq i \leq N$ **do**
▷ Prepare the message schedule $\{W_t\}$
9: $W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ ROTL^1(W_(t-3) \oplus W_(t-8) \oplus W_(t-14) \oplus W_(t-16)) & 16 \leq t \leq 79 \end{cases}$
10:
▷ Initialize the five working variables, a, b, c, d, and e, with the $(i-1)^{\text{st}}$ hash value:
11: $(a, b, c, d, e) \leftarrow (H_0^{(i-1)}, H_1^{(i-1)}, H_2^{(i-1)}, H_3^{(i-1)}, H_4^{(i-1)})$
12: **for** $0 \leq t \leq 79$ **do**
13: $T \leftarrow ROTL^5(a) + f_t(b, c, d) + e + K_t + W_t$
14: $(e, d, c, b) \leftarrow (d, c, ROTL^{30}(b), a)$
15: $a \leftarrow T$
16: **end for**
17:
▷ Compute the i$^{\text{th}}$ intermediate hash value $H^{(i)}$
18: $(H_0^{(i)}, H_1^{(i)}, H_2^{(i)}, H_3^{(i)}, H_4^{(i)}) \leftarrow (a + H_0^{(i-1)}, b + H_1^{(i-1)}, c + H_2^{(i-1)}, d + H_3^{(i-1)}, e + H_4^{(i-1)})$
19: **end for**
20:
21: **return** $(H_0^{(N)} || H_1^{(N)} || H_2^{(N)} || H_3^{(N)} || H_4^{(N)})$
22: **end procedure**

For the calculation of W_t, the subunit shown in Figure 5.18 is implemented. It acts as a shift register, where the output is taken from its head. Initially, the 16 registers are populated with values from the corresponding parts of the input block, with the rightmost register assigned to the 1$^{\text{st}}$ block portion. Thus, for the first 16 cycles, the subunit outputs the parts of the input block. To compute the next W_t, a bitwise XOR of W_{t-3}, W_{t-8}, W_{t-14}, and W_{t-16} is used, and the result is left-shifted by 1 bit. The new value of W_t is then placed in the tail of the registers. Therefore, from the 17$^{\text{th}}$ cycle onwards, the output corresponds to the calculated W_t according to the second branch of the calculation in line 9 in algorithm 1.

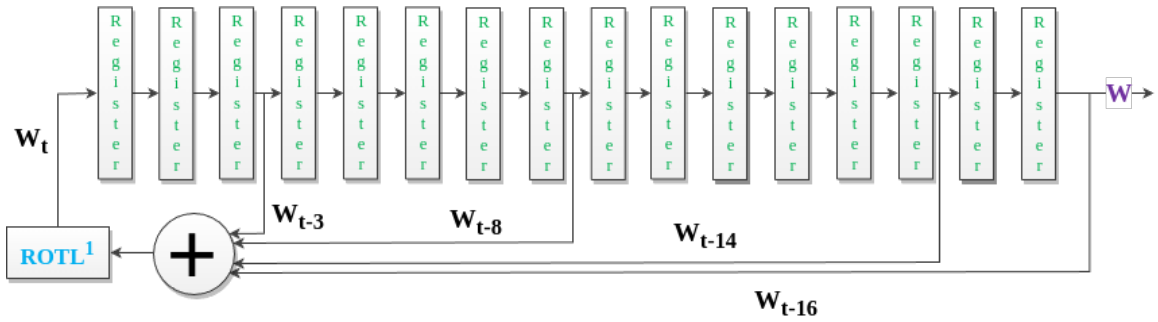

Figure 5.18: SHA1 message schedule

The implementation of sha1_core is depicted in Figure 5.19. Its inputs are the input block, the values $H_0^{(i-1)}$, $H_1^{(i-1)}$, $H_2^{(i-1)}$, $H_3^{(i-1)}$, $H_4^{(i-1)}$, and the clock (clk) and reset (rst) signals. The H values are also needed as inputs because, as can be observed in the pseudocode in Algorithm 1, the new H values are calculated using the previous ones. Thus, as will be explained in the sha1 unit description, there is an external feedback loop from the output of the sha1_core into $H_0^{(i)}$, $H_1^{(i)}$, $H_2^{(i)}$, $H_3^{(i)}$, $H_4^{(i)}$. sha1 outputs the computed hash value and a signal hash_rd which indicates that the processing of this block has finished.

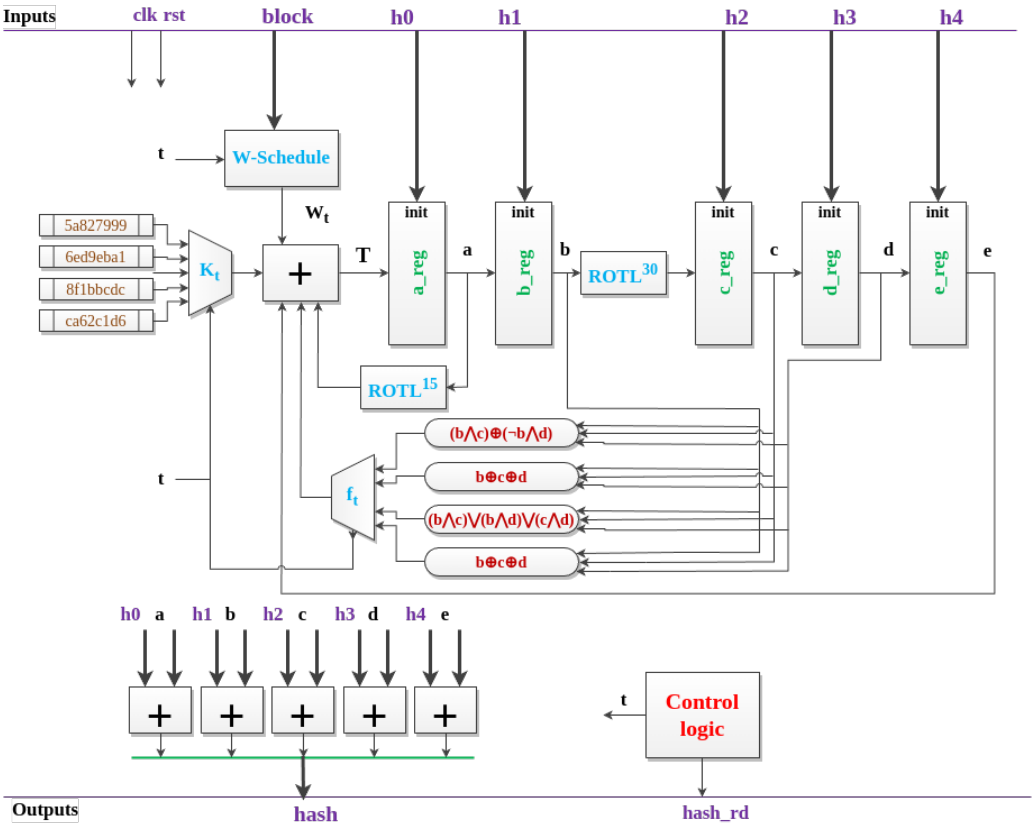

Figure 5.19: sha1_core module implementation

In Figure 5.19, the variable t corresponds to the number of current iterations. The 5 registers connected in series (a_reg to e_reg) implement the assignments in line 14. The calculation of line 13 is broken down to:

- The calculation of f_t (5.2.2) is performed through four segments of digital logic and a multiplexer among these four. The MUX selection is based on t
- The signal K_t (5.2.2) is assigned one of the four constant values depending on the value of t

- The W_t is computed by the W-Schedule unit
- $ROTL5(a)$ is calculated, and all the above, along with e, are fed into an addition unit whose output is the value of T
- the value of T is stored into the a_reg in order to implement the final assignment $a = T$

The output values (line 21) are computed using adders. The control logic includes a counter that is initialized to '0' with every rst and has a maximum value of '79', which keeps until a new rst signal is given. The value of t corresponds to the value of this counter. Furthermore, the control logic raises the output signal hash_rd when the counter reaches the value '79' to indicate that all iterations have been executed.

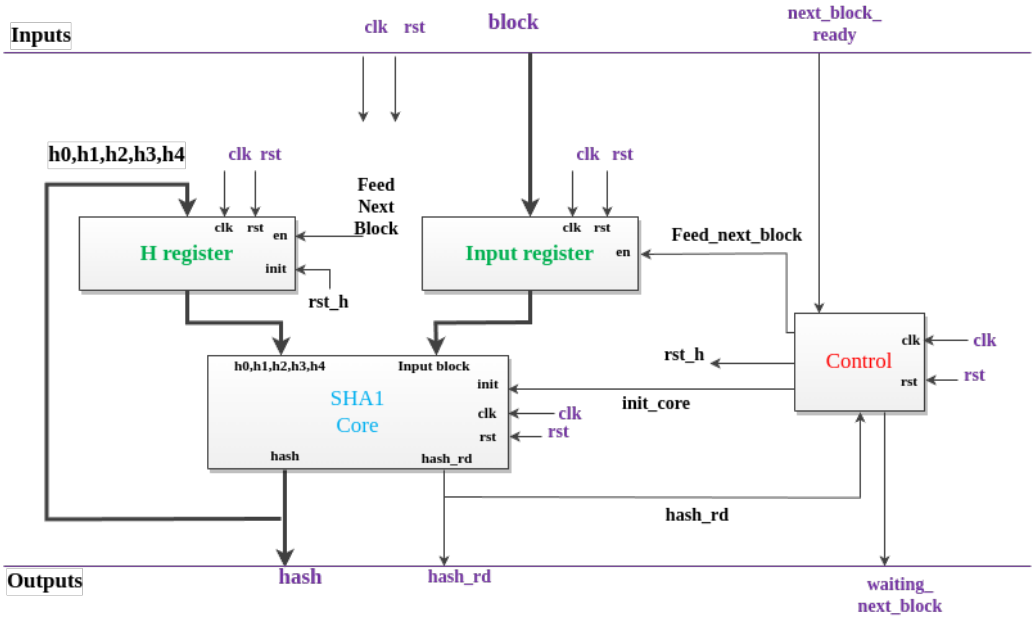

Figure 5.20: sha1 module implementation

The sha1 unit takes as input the message's current block, the control signal next_block_ready which notifies the unit that a new block is available at its input, and the clock (clk) and reset (rst) signals. It outputs the result of the hash computation, the hash_rd signal indicating the end of hash computation for the current block, and a waiting_next_block signal indicating that the unit is ready to accept a new block at its input. The block diagram of the implementation is shown in Figure 5.20.

The design consists of a sha1_core, two registers, and a control unit. The input register serves as a buffer that holds the input block, allowing a new block to be placed at the unit's input without affecting the ongoing operation. The H register is used to feed back the values $H_0^{(i-1)}$, $H_1^{(i-1)}$, $H_2^{(i-1)}$, $H_3^{(i-1)}$, and $H_4^{(i-1)}$ for processing the next block, which implements the assignments in line 11.

The operation of the control unit is described through the FSM diagram in Figure 5.21. For starting a hash computation of a new message, a reset signal needs to be issued that places us in the Init state. In this state, the H register is set to the initial values (line 2) and initializes the sha1_core unit with the internal signal Init_core. When entering the Init state, the waiting_next_block is issued.

After the first block is provided at the input the the next_block_ready signal is issued and that transitions the FSM to the Waiting state where the 80 iterations are executed for the first block. In this state, the control unit waits for the sha1_core to finish processing, which is indicated by the hash_rd signal, and for the next_block_ready signal to be given (after placing the next block at the input). When both conditions are met, the FSM transitions to the Pass Next Values state, where a pulse of the internal signal feed_next_block is issued so that the new values of H and the next block are provided

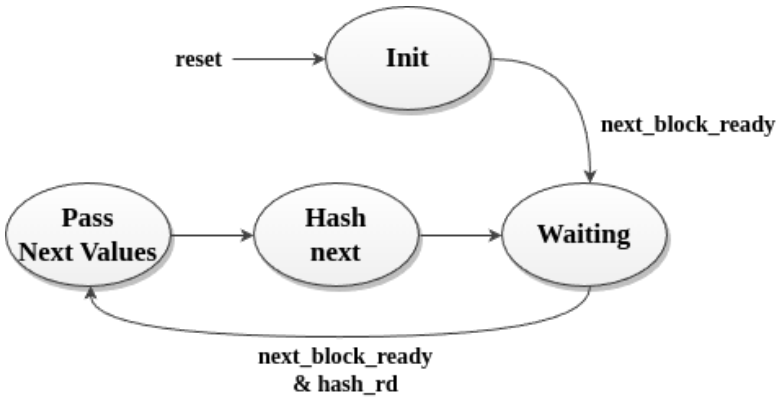

Figure 5.21: sha1 control FSM

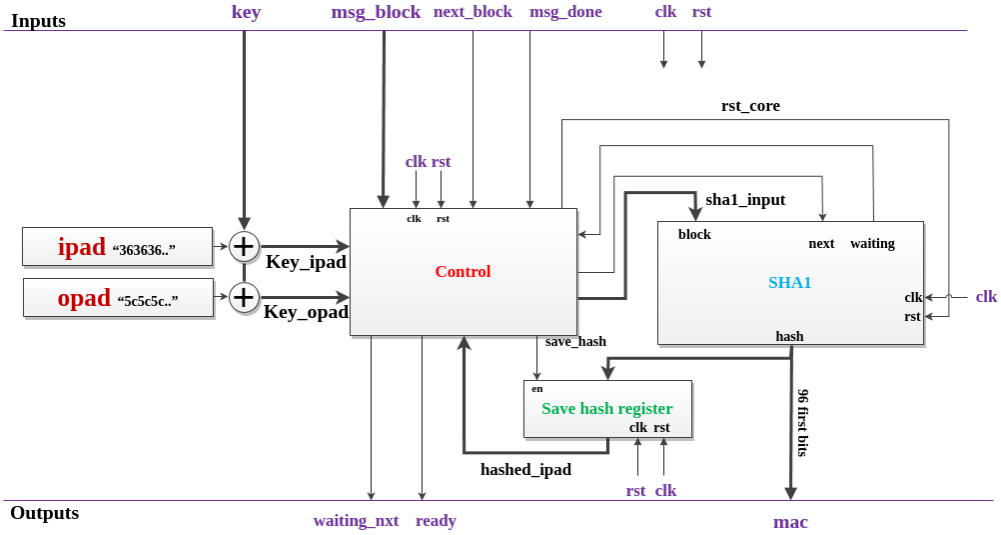

Figure 5.22: hmac_sha1_96 module implementation

to the sha1_core input. After one clock cycle, the FSM transitions to the Hash next state, where the sha1_core has taken the new values at its inputs, and then automatically transitions to the Waiting state where the 80 iterations are executed for the current block. This process repeats until all blocks are provided.

The last unit implemented is the hmac_sha1_96, whose block diagram is shown in Figure 5.22. At its input, it receives the key, one message block msg_block, the control signals next_block and msg_done, and the clock and reset signals. The next_block signal is used to indicate that the next block is at the msg_block input, while the msg_done signal indicates that we are providing the last block of the message. Its outputs include the computed MAC and the signals waiting_nxt and ready. The waiting_nxt signal indicates that the unit is ready to accept a new block at its input, and ready indicates the end of the MAC computation.

The unit consists includes the constants $ipad$ and $opad$, two bitwise XOR operations between these constants and the key, a sha1 unit, a register to store the intermediate calculation of $H((K_0 \oplus ipad)||text)$, and a control unit.

The control unit is responsible for routing the appropriate value to the input of the sha1 unit. This is depicted in Figure 5.22 with the values of $K_0 \oplus ipad$, $K_0 \oplus opad$, hashed_ipad, and msg_block being provided to the control unit. The control unit's FSM is presented in Figure 5.23.

For each new calculation of HMAC-SHA1-96, the unit is first initialized with a reset signal leading the FSM to the Start state. In this state, the sha1 unit is initialized through a pulse issued on the internal signal rst_core, while the signal waiting_nxt is kept active indicating that the unit is ready to accept the first block. Once the first block is provided and the next_block signal is issued, the FSM

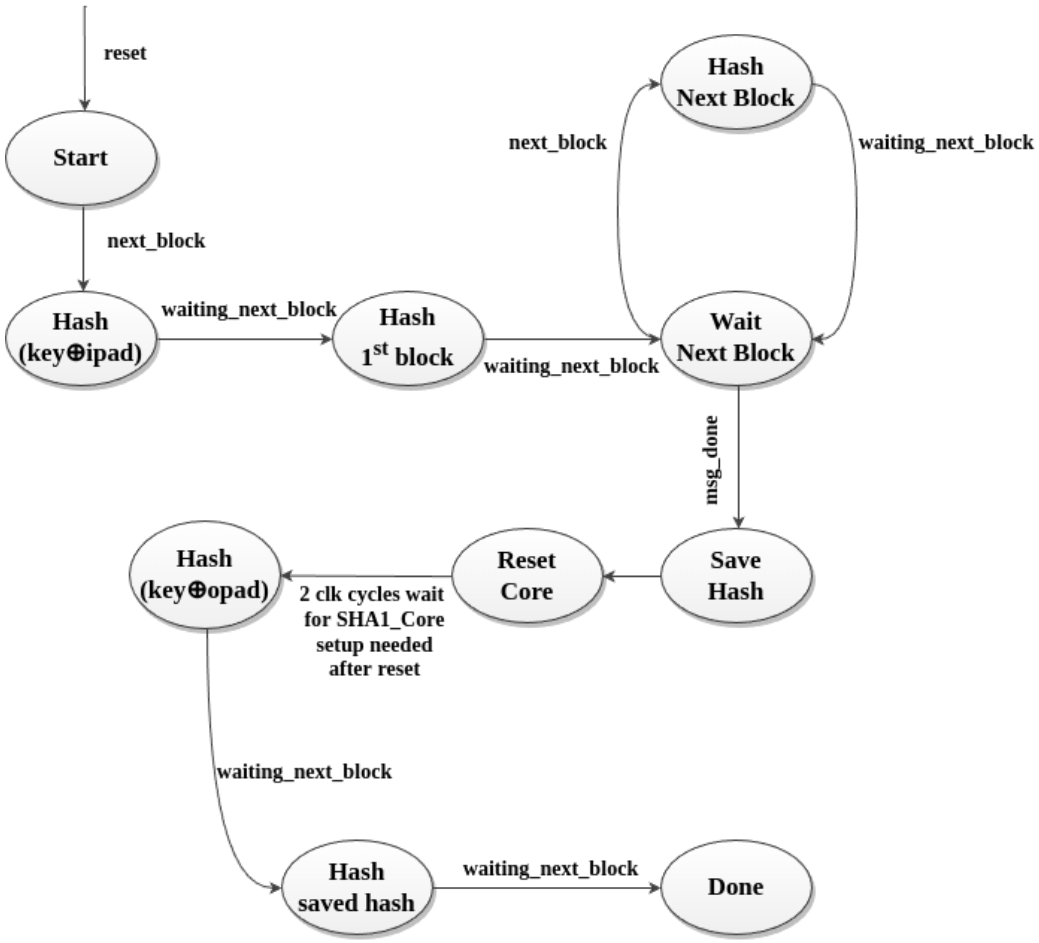

Figure 5.23: hmac_sha1_96 control FSM

transitions to the $Hash(key \oplus ipad)$ state. There the first block of the message $(K_0 \oplus ipad)||text$, $(K_0 \oplus ipad)$, is routed to the sha1 unit. Once the calculation is completed, the waiting_next_block signal of the sha1 unit is issued, and the FSM transitions to the Hash 1st block state, where the first block of the message is given to the sha1 unit. After this calculation finishes, the FSM enters the loop of states Wait Next Block and Hash Next Block. In the Wait Next Block state, the unit waits for the next block, and once provided, it moves to the Hash Next Block state to perform the next calculation on the newly provided block. This process continues until the msg_done signal is issued, indicating that all blocks of the message have been provided, and therefore, the output of the sha1 unit has the quantity $H((K_0 \oplus ipad)||text)$.

When the msg_done signal is issued, the FSM transitions to the Save Hash state, where the value $H((K_{\oplus}ipad)||text)$ is stored in the Save Hash Register using the save_hash signal. In the next clock cycle, the FSM automatically moves to the Reset Core state, where the sha1 unit gets prepared for a new calculation. When the sha1 unit is ready, the calculation $H[(K_0 \oplus opad)||H((K_0 \oplus ipad)||text)]$ starts. In the $Hash(key \oplus opad)$ state, the block $(K_0 \oplus opad)$ is given to the sha1 unit. Once the calculation is finished (i.e., when waiting_next_block='1'), the 2nd block of the message is provided, $H((K_0 \oplus ipad)||text)$. At the end of this calculation, the final MAC value is available at the output of the sha1 unit. The system moves to the Done state, where it issues the ready signal to indicate that the MAC is ready. The first 96 bits of the sha1 result are present at the output of the hmac_sha1_96.

5.3 Hardware-side Interface

The interfaces of the two peripheral systems, CBC-AES-128 and HMAC-SHA1-96, are not fully suitable for direct connection to the processor bus. Their inputs and outputs need to be represented with registers corresponding to memory locations of the processor's address space (memory-mapped IO).

Xilinx's toolset offers a procedure for creating and connecting custom-designed peripheral systems. This type of peripheral is referred to by Xilinx as a custom IP core. Through this process, these peripherals and their interfaces are defined. In the case of CBC-AES-128, the interface consists of 17 32-bit registers, of which 4 are used for the input block, 4 for the output block, 4 for the key, 4 for the IV, and the last one serves as a command/status register. The last register, when read by the processor, provides peripheral notification signals, while when written, it sets the peripheral's command signals. It is worth noting that the 4 registers of the output block could overlap with some of the input registers since the former are used only for writing and the latter only for reading. A choice to separate them is made solely to keep the design clear. For HMAC-SHA1-96, there are a total of 36 32-bit registers. 16 are dedicated to the input block, 16 to the key, 3 to its output (a 96-bit MAC), and one is the command/status register. As several bits in the command/status registers remain unused, some signals are exposed through them to assist with the debugging of the peripherals.

6. Software Components

6.1 lwIP

lwIP (lightweight IP) ([13], [14]) is an open-source implementation of the TCP/IP stack for embedded systems. Specifically, it implements the Ethernet, ARP, IP, TCP, and UDP protocols. It allows the implementation of user applications with networking requirements on embedded systems.

The most fundamental data structure of lwIP is the p_buf (packet buffer). p_buf is used for organizing packets and storing them in memory and is used in all layers of the stack. It contains the headers and data of the packet and additional information such as its size. Along with the p_buf structure, functions for its processing are provided, such as creation, deletion, extension, etc.

The datapath followed by an incoming packet starts with its reception. Here, the processor is notified through an Ethernet Controller interrupt for the arrival of the new packet. When the processor becomes available, it retrieves the packet from the incoming queue of the Ethernet Controller and creates a new p_buf for it. Next, the p_buf is passed upwards from layer to layer. Each layer performs the appropriate processes on the packet, removes its header, and passes it to the next layer. If any layer decides that the packet should be dropped due to failing necessary checks or when it reaches the last processing layer, the p_buf is deleted an memory is reclaimed. Outgoing packets follow the reverse path.

Each layer offers an interface to other layers. This interface essentially consists of the actions of receiving and transmitting a packet, and some additional layer-specific function. For example, the interface of the IP layer is shown in Listing 6.1.

```
struct netif * ip_route(struct ip_addr *dest);

err_t ip_input(struct pbuf *p, struct netif *inp);

err_t ip_output(struct pbuf *p, struct ip_addr *src, struct ip_addr *dest
    , u8_t ttl, u8_t tos, u8_t proto);

err_t ip_output_if(struct pbuf *p, struct ip_addr *src, struct ip_addr *
    dest, u8_t ttl, u8_t tos, u8_t proto, struct netif *netif);
```

Listing 6.1: IP layer's interface in lwIP

As observed, the interface consists of input and output functions, and a specialized function of the IP protocol used for routing (ip_route). Lastly, each protocol has multiple parameters that can be configured by the designer, such as the TCP timeout, the maximum packet size, etc.

6.2 IPsec

The IPsec implementation is integrated into the existing codebase of lwIP, as mentioned in Chapter 4.2.2. The software development is divided into two parts. The first involves creating in isolation a

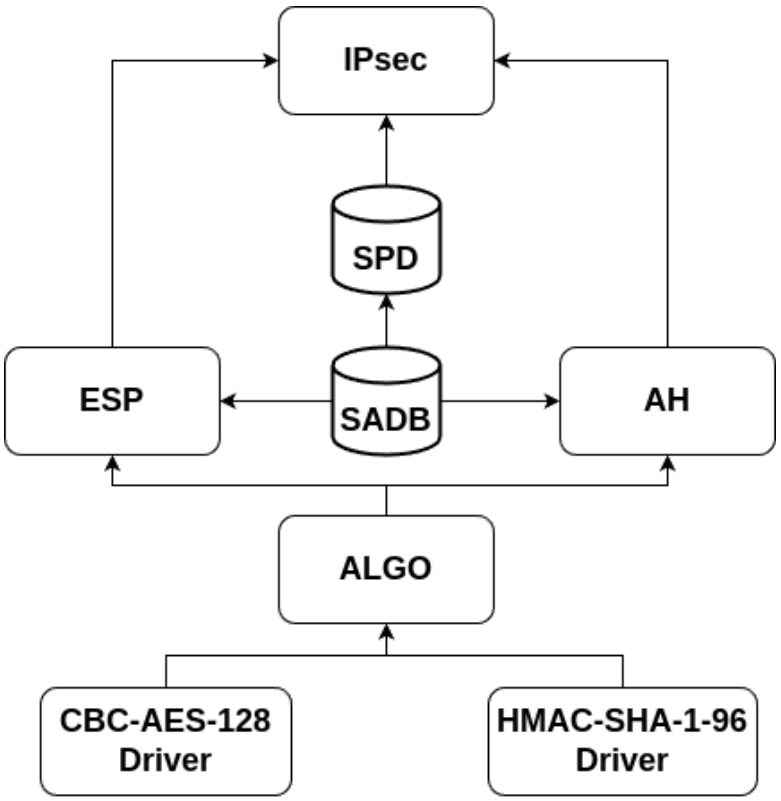

Figure 6.1: Software architecture diagram for the custom IPsec library

software library that provides IPsec functionality, and the second involves interfacing and integrating this library with lwIP. The components of the library and their relationships are illustrated in Figure 6.1. Below, the individual components of the library created for the various IPsec functions and the method of integration with lwIP are explained.

6.2.1 IPsec

The library implementing IPsec provides two functions, one for processing inbound traffic and one for processing outbound traffic. The inbound function is called by the ip_input of the IP after the packet has been processed by the IP. The outbound function is called at the beginning of the ip_output_if so that every outgoing packet is first processed by IPsec before proceeding. ipsec_output returns a code to ip_output_if so that the latter can select the operations to perform. The function signatures are shown in Listing 6.2.

```
void ipsec_input(struct pbuf *p, struct netif *inp);

u8_t ipsec_output(struct pbuf *p, struct pbuf **p_new, struct ip_addr *
   src, struct ip_addr *dest, u8_t ttl, u8_t tos, u8_t proto, struct
   netif *netif);
```

Listing 6.2: IPsec implemented interface

The input function takes the *pbuf* of the incoming packet and the Ethernet interface where the arrival occurred. Firstly, it extracts the packet fields' values and issues a SADB lookup. If the packet is not matched with any SA, it is discarded. If there is a match, depending on the protocol referenced in the SA, either esp_input or ah_input is called. After these functions finish processing the packet and no conditions for packet rejection arise, the SPD is checked to ensure that the packet complies with the policy. If the packet passes the SPD check, it is forwarded to ip_input as a pure IP packet to

continue its path in the TCP/IP stack.

The output function takes the pbuf of the outgoing packet, a pbuf to which the processed packet will be assigned as a result of ipsec_output, the IP addresses of the sender and receiver, the TTL and TOS values of the packet, and the Ethernet interface that will perform the transmission. The function returns a code, whose values correspond to:

- RET_IPSEC_ERROR if an errors occur,
- RET_IPSEC_BYPASS for applying the BYPASS policy,
- RET_IPSEC_APPLY for applying the PROTECT policy,
- RET_IPSEC_DISCARD for applying the DISCARD policy.

The ipsec_output initially searches the SPD. If no policy is found, the packet is discarded. If found, if it's a DISCARD policy, the function simply returns the RET_IPSEC_DISCARD code. If it's BYPASS traffic, no processing is performed on the packet, and the function returns with the RET_IPSEC_BYPASS code. If it's PROTECT traffic, depending on the protocol defined in the policy's SA, the packet is passed to either ah_output or esp_output. If they return without errors, the function returns with the RET_IPSEC_APPLY code; otherwise, with the RET_IPSEC_ERROR code. If an error occurs at any point in ipsec_output, it returns with the RET_IPSEC_ERROR code.

The ipsec_output returns to ip_output_if, which has been modified to interpret the return value of ipsec_output properly. Thus, when RET_IPSEC_ERROR is returned, it discards the packet and reports an error. When RET_IPSEC_BYPASS is returned, the regular IP processing is executed. If RET_IPSEC_APPLY is returned, the packet is a ready IP packet for transmission due to processing by ah_output or esp_output and is sent without further processing. Finally, if RET_IPSEC_DISCARD is returned, ip_output_if discards the packet and releases the memory.

6.2.2 AH and ESP

The main processing of packets in IPsec is performed by the AH and ESP protocols. These protocols add additional headers and populate the header fields. Packets undergo cryptographic transformations for outgoing traffic, while header fields are checked and packets undergo reverse cryptographic transformations and checks for incoming traffic.

For incoming traffic, the implementation of AH and ESP provide an input function, ah_input and esp_input respectively. These functions are called by ipsec_input, with arguments being the pbuf of the packet and the corresponding SA. Both functions start by checking the Sequence Number. Then the authentication and integrity checks follow which are mandatory for AH but optional for ESP. Next, for ESP only, the packet gets decrypted. The parameters for all these operations are found in the applicable SA. Finally, both functions consult the SA regarding the IPsec mode in which they should process the packet. In the case of Transport Mode, the IPsec header is removed, and the IP header is prepended to the payload. For Tunnel Mode, the outer header and the IPsec header are removed, leaving the inner header intact. A the end of these processes, we obtain an IP packet which is forwarded to ip_input accordingly. The interfaces for AH and ESP inbound process are shown in Listing 6.3.

```
int ah_input(struct pbuf* p, struct sa_entry* sa);
int esp_input(struct pbuf* p, struct sa_entry* sa);
```

Listing 6.3: AH and ESP inbound processing interfaces

For outbound traffic, the functions ah_output and esp_output are provided, which are called by ipsec_output. Their arguments include the pbuf of the packet, a new pbuf where the processed packet will be placed, the policy entry found by ipsec_output, as well as the IP addresses of the sender and

recipient, the values of TTL and TOS, and the interface from which the packet will be sent. TTL and TOS are provided by the Transport Layer protocols and are given to ah_output and esp_output so that they can construct the inner IP header and, if it is tunnel mode traffic, the outer IP header too. The interfaces for AH and ESP inbound process are shown in Listing 6.4.

```
err_t ah_output(struct pbuf *p, struct pbuf **p_new, struct spd_entry*
   spdentry, struct ip_addr *src, struct ip_addr *dest, u8_t ttl, u8_t
   tos, u8_t proto, struct netif *netif);

err_t esp_output(struct pbuf *p, struct pbuf **p_new, struct spd_entry*
   spdentry, struct ip_addr *src, struct ip_addr *dest, u8_t ttl, u8_t
   tos, u8_t proto, struct netif *netif);
```

Listing 6.4: AH and ESP outbound processing interfaces

Packet processing begins with the placement of the inner IP header if in Tunnel Mode. Then, for ah_output, the AH header is placed, followed by the outer IP header, and finally, the packet goes through the integrity and authentication algorithm, and the ICV value is placed in the appropriate field before being handed back to ipsec_output.

In esp_output, after the possible placement of the inner IP header, padding and IV are added, followed by encrypting all of the above together with the payload. Then, the ESP header is added, and if integrity is used, the integrity algorithm is applied. The ICV of the integrity algorithm is placed at the end of the packet, and finally, the outer IP header is added. Upon returning from esp_output to ipsec_output, the processed packet is ready to continue its transmission.

6.2.3 SADB and SPD

The SADB is implemented as a unidirectional linked list. Each entry in the SADB represents an SA. The structure implemented to represent an SA contains:

- the SPI number
- the IANA number of the protocol
- the mode corresponding to the traffic (transport or tunnel)
- the packet sequence number (Sequence Number)
- The authentication algorithm used by AH (if applicable, i.e., if AH protocol
- is used for this traffic)
- If applicable, the key is also retained.
- The authentication algorithm used by ESP (if applicable, i.e., if ESP
- protocol is used and authentication is used for this traffic simultaneously)
- If applicable, the key is also retained.
- The encryption algorithm of ESP (mandatory if ESP is used).
- If applicable, the key is also retained.
- A pointer to the next SADB entry to maintain a unidirectional linked list.

The definition of the structure is shown in Listing 6.5.

```
struct sa_entry
{
  u32_t spi;
  u8_t protocol;
  u8_t mode;    // Transport or Tunnel

  /* Current Connection */
  u32_t seqnum;

  /* Algorithms */
  struct auth_algo* ah_auth_algo;
  u8_t* ah_auth_key;

  struct enc_algo* esp_enc_algo;
  u8_t* esp_enc_key;

  struct auth_algo* esp_integrity_algo;
  u8_t* esp_integrity_key;

  /* Linked entries*/
  struct sa_entry *next;

}__attribute__((packed));
```

Listing 6.5: The SADB entry structure

The structures auth_algo and enc_algo are created to add a flexible interface for cryptographic algorithm integration. They serve as an intermediate layer between IPsec and the algorithm interfaces. If a new cryptographic algorithm needs to be used, it is sufficient to create an auth_algo or enc_algo structure (depending on the type of algorithm) so that it can be directly utilized by the specific IPsec implementation.

The SADB library provides functions for creating and adding a new SA to the database and for searching for an SA based on the SPI and protocol number values. The search function returns the entry if found. The signatures of these functions are shown in Listing 6.6.

```
struct sa_entry *sa_lookup(u32_t spi, u8_t proto);

struct sa_entry* sa_add(u32_t spi, u8_t protocol, u8_t mode,
          struct auth_algo* ah_auth_algo, u8_t* ah_auth_key,
          struct enc_algo* esp_enc_algo, u8_t* esp_enc_key,
          struct auth_algo* esp_integrity_algo, u8_t* esp_integrity_key);
```

Listing 6.6: The SADB interface

Similarly to SADB, SPD is implemented as a unidirectional linked list. Each entry in the list represents a policy and includes:

- The sender's IP address
- The sender's subnet mask
- The receiver's IP address

- The receiver's subnet mask
- The protocol's IANA number
- The type of policy (PROTECT, BYPASS, DISCARD)
- The direction of traffic (IN or OUT)
- The IPsec protocol used in the case of the PROTECT policy (AH or ESP)
- The mode (Transport or Tunnel)
- The sender's IP address of the tunnel
- The receiver's IP address of the tunnel
- The SA that characterizes the policy's traffic
- A pointer to the next entry in the SPD to implement the unidirectional linked list.

The definition of the structure is shown in Listing 6.7.

```
struct spd_entry{

  struct
  ip_addr*  src ;      /**< IP source address */
  u32_t     src_netaddr ;   /**< net mask for source address */
  struct
  ip_addr*  dest ;       /**< IP destination address */
  u32_t   dest_netaddr ;    /**< net mask for the destination address */
  u8_t    protocol ;    /**< the transport layer protocol */
  u8_t    policy ;      /**< defines how this packet must be processed*/
  u8_t    direction;    /**< defines IN or OUT traffic direction */
  u8_t    ipsec_proto;    /**< AH or ESP processing */
  u8_t    ipsec_mode;   /**< Transport or Tunnel mode */
  struct
  ip_addr*  tunnel_src;   /**< IP tunnel source */
  struct
  ip_addr * tunnel_dest;    /**< IP tunnel destination */

  struct
  sa_entry* sa ;       /**< pointer to the associated SA */
  struct
  spd_entry*  next ;       /**< pointer to the next table entry*/
}
```

Listing 6.7: The SPD entry strucutre

The SPD library provides functions for creating and adding a new policy to the database, as well as for searching for a policy based on the sender's and receiver's IP and subnet mask values and the protocol number. The search function returns the entry if found. The signatures of these functions is shown in Listing 6.8.

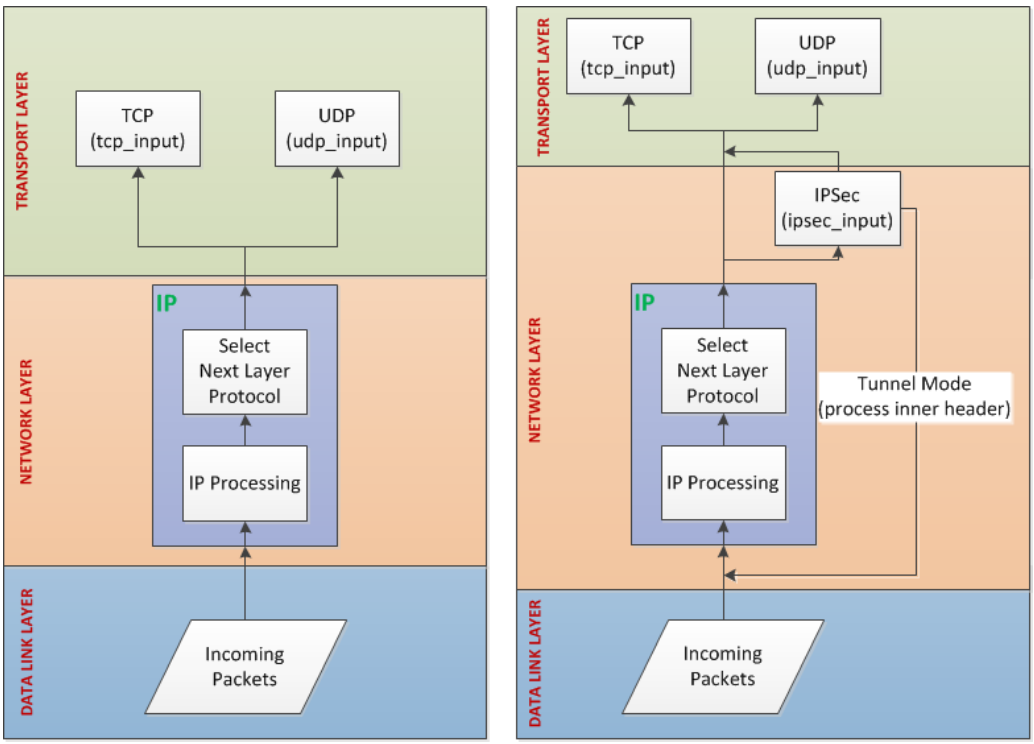

Figure 6.2: Integration of inbound IPSec packet processing in lwIP

```
struct spd_entry* spd_lookup(struct ip_addr *src, u32_t src_netaddr,
struct ip_addr *dest, u32_t dest_netaddr, u8_t proto);

void spd_add(struct ip_addr *src, struct ip_addr *dest, u8_t proto,
    u8_t policy, u8_t direction, u8_t ipsec_proto, u8_t ipsec_mode,
    struct ip_addr *tunnel_src, struct ip_addr *tunnel_dest,
    struct sa_entry* sa);
```

Listing 6.8: The SPD interface

6.3 Integration with lwIP

The software interface of the implemented IPsec library consists of two functions, one for processing incoming packets and one for processing outgoing packets.

The modifications in lwIP are made at the IP level since IPsec belongs to this layer. For inbound traffic, the packet must pass through IPsec after being processed by IP. Thus, the ip_input function is modified to hand over control to the IPsec input routine. Specifically, ip_input first processes the packet, and when finished, decides which protocol to pass the packet's payload to. Here, the option of IPsec for AH and ESP protocols is introduced. Figure 6.2 illustrates the flowchart before and after modification of the inbound processing.

The function ipsec_input serves as the interface for incoming IPsec packets. If the next hop after IP is IPsec, then the packet is handed over via ipsec_input. At the end of IPsec processing, if the packet is in Transport Mode, it is forwarded to the appropriate Transport Layer protocol. If the packet is in Tunnel Mode, then it is passed back to the IP layer for further processing of the inner IP header and subsequently forwarded to one of the Transport Layer protocols.

For outbound traffic, the packet must first be passed to IPsec, and if it passes all the checks, it is handed over to IP for transmission. In this case, changes are made to the ip_output_if function.

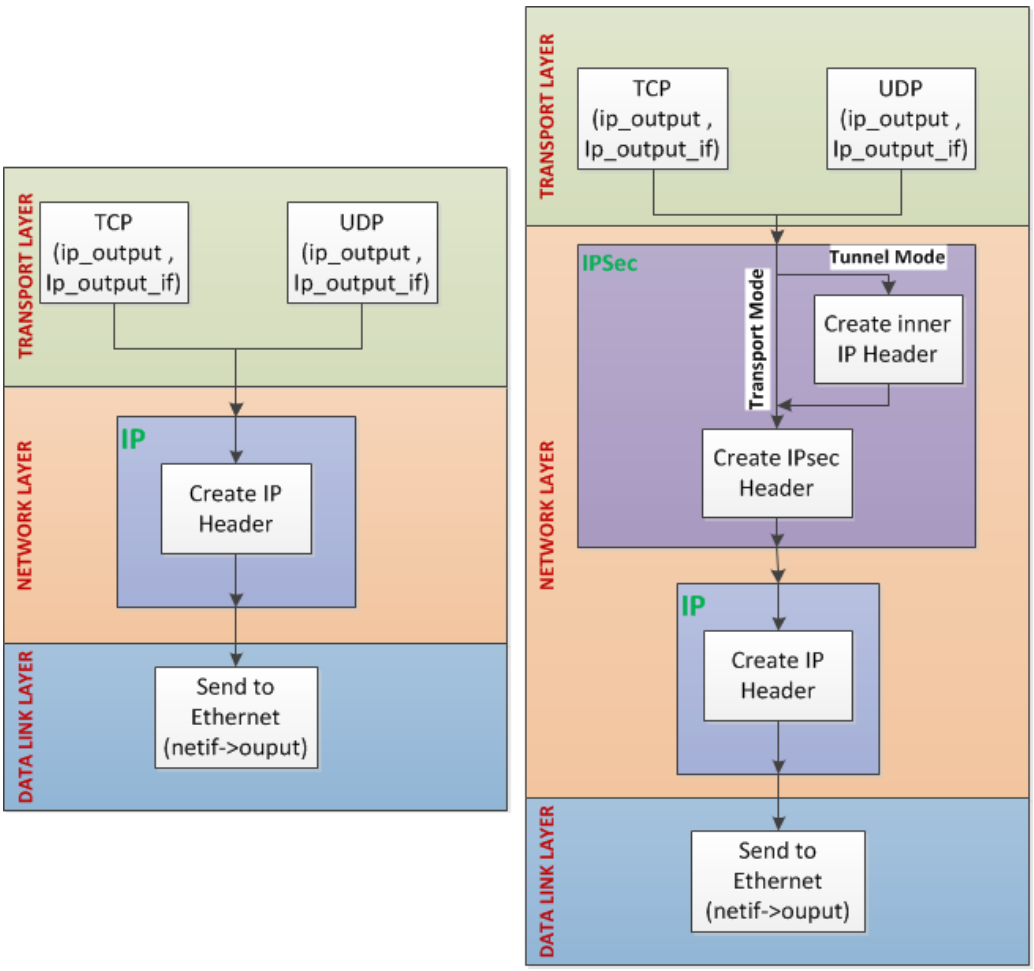

Figure 6.3: Integration of outbound IPSec packet processing in lwIP

This is more heavily modified compared to ip_input, as a considerable amount of IPsec code had to be executed before transferring the packet to the IP layer, such as searching the SPD and creating the IPsec header.

The flowcharts for outbound traffic of lwIP and modified lwIP are shown in Figure 6.3. When a packet is forwarded from one of the transport layer protocols to IP, the ip_output or ip_output_if function is invoked. During execution in the modified lwIP, the packet is first processed by IPsec and then by IP. In processing by IPsec, we have two cases: the first is when the packet belongs to tunnel mode traffic, where the inner IP header is placed first, followed by the IPsec header, and the second is when we are in transport mode traffic, so only the IPsec header is placed.

6.4 Software-side Interface

Software drivers are created for the two peripherals to abstract out the details of handling the hardware interface from the rest of the software. Specifically, these drivers handle the movement of data between memory and peripherals and utilize the communication protocol of the peripheral to control its operation. They provide certain functions to the system to seamlessly execute cryptographic operations using the peripherals.

For the CBC-AES-128 peripheral, the driver has two functions, one for encrypting and one for decrypting a message. Their inputs are the memory location from which the message starts, the size of the message, the memory location from which the key starts, and the memory location from which the IV starts. Initially, the key, IV, and the first block of the message are copied from memory to the

appropriate registers of the peripheral. Then, a start_new signal is issued along with a suitable value of the EncDec signal (’0’ for encryption, ’1’ for decryption) to notify the peripheral that a computation on a new message is starting. The driver then waits for the peripheral’s ready signal to be issued. Once issued, it copies the result to the memory locations where the original block was, copies the next block, and issues a next_block signal. It then waits again for the ready signal, then copies the result to memory, and then issues the next block. This process repeats until all input blocks are processed. It is noted here that using CBC-AES-128 with ESP relieves the driver from performing padding, as the packet already contains padding from ESP processing. The function signatures and their operation are described using pseudocode in Listing 6.9.

```
void AES128_CBC_encrypt(void* data, u32_t length,  void* key, void* iv)
{
  Copy the first 128 bits of data to the peripheral
  Copy key to the peripheral
  Copy iv to the peripheral

  Send start_new’’=1 and EncDec = ‘’0
  Wait until ready’’=1
  Copy result in the first 128 bits of data

        For all the next 128 bits blocks of data:
     Copy the next 128 bits of data
     Send next_block
     Wait until ready’’=1
     Copy result in the corresponding 128 bits of data
}

void AES128_CBC_decrypt(void* data, u32_t length, void* key, void* iv)
{
  Copy the first 128 bits of data to the peripheral
  Copy key to the peripheral
  Copy iv to the peripheral

  Send start_new’’=1 and EncDec = ‘’1
  Wait until ready’’=1
  Copy result in the first 128 bits of data

        For all the next 128 bits blocks of data:
     Copy the next 128 bits of data
     Send next_block
     Wait until ready’’=1
     Copy result in the corresponding 128 bits of data
}
```

Listing 6.9: AES driver pseudocode

For the peripheral HMAC-SHA1-96, the driver provides two functions, one for generating the MAC of a message and one for comparing a given MAC with the MAC of a message. Both functions execute the hmac-sha1-96 on the message, and their only difference is that the first one copies the MAC result to a specific memory area, while the second one compares the MAC result with another MAC and returns the comparison result. The former is used by IPsec to create the MAC of a new packet and place it in the appropriate field, while the latter is used to authenticate an incoming packet.

The procedure for handling the HMAC-SHA1-96 peripheral is as follows: initially, for each

new message (packet), a reset signal is provided. Then, the driver copies the key and the first block of the message from memory to the appropriate registers of the peripheral and issues a next_block signal. Subsequently, the driver waits for the waiting_nxt signal to place the next block and issues the next_block signal again. This process continues for all blocks, and once all the message blocks are processed, a msg_done signal is issued, and the driver is then waiting for the ready signal. Finally, once the ready signal is issued, if we are in the MAC generation function, the result is copied to memory, while if we are in the comparison function, the result is compared with the given MAC, and the comparison result is returned. It is noted that the padding of the block is calculated in the driver functions and that the key size is exactly 160 bits, so no preprocessing is needed. Below are the function signatures and their operation described using pseudocode in Listing 6.10.

```
int HMAC_SHA1_96_authenticate(void* data, u32_t datalength, void* key,
    u32_t keylength, void* icv)
{
    Padding execution,
    Send rst

    For each block of 512 bits of the message:
        Copy the first 512 bits of the message
        Send next_block='1'
        Wait for waiting_nxt
        Send msg_done
        Wait for ready='1'

        If icv == result of the peripheral
            Return 1
        Else
            Return 0
}

void HMAC_SHA1_96_calculate(void* data, u32_t datalength, void* key,
   u32_t keylength,
                    void* icv)
{
    Padding execution,
    Send rst

    For each block of 512 bits of the message:
        Copy the first 512 bits of the message
        Send next_block='1'
        Wait for waiting_nxt
        Send msg_done
        Wait for ready='1'
        Copy the result of the peripheral to icv
}
```

Listing 6.10: HMAC driver pseudocode

7. FPGA Implementation and On-Chip Verification

7.1 Tools

7.1.1 Xilinx Suite and Modelsim

For the designing process, the Microblaze programming, the custom IP cores implementation, and the board programming, the Xilinx tool suite, Xilinx ISE Design Suite 13.1 [15], is utilized. For simulating the design, Modelsim is preferred due to its capabilities and direct integration with Xilinx tools.

ISE comprises tools for hardware synthesis in VHDL or Verilog. It provides syntax checking, direct simulation, and hardware synthesis for FPGA programming. This toolset is used for the implementation of the cryptographic algorithms.

An important tool within ISE is the Embedded Development Kit (EDK), which aids in system design, offering an easy instantiation of different processors and a plethora of ready-to-use peripherals. It supports configuration options of several system parameters, such as processor cache size, peripheral addressing schemes, different bus types instantiations, and choice amongst several interfacing options between processors and peripherals. Additionally, custom IP Cores can be easily integrated into the system using EDK. The IPsec system is defined using EDK, the Ethernet, UART, and DDR2 peripherals are added, and the custom cryptographic IP cores AES-128 and HMAC-SHA1-96 are integrated.

Another crucial tool is the Software Development Kit (SDK). It helps manage the system's software, from implementing a simple routine to loading a full-fledged operating system onto processors (e.g., Xilkernel or embedded Linux). It supports programming in C, C++, and assembly. Moreover, it offers ready-to-use software systems such as the lwIP protocol stack and the Xilkernel operating system. From this tool, the software can be downloaded and run in an FPGA system. Software debugging is also performed using SDK. Finally, it includes monitoring of a serial port, allowing us to receive program messages through STDIO (standard input-output). The IPsec code is developed and integrated into the lwIP library using this tool.

Modelsim is a widely used hardware simulator program. It is directly integrated into ISE, which eliminates the need to manually include Xilinx libraries in Modelsim. Furthermore, it provides automation capabilities for simulation, based on the Tcl language, boosting productivity during design debugging and verification. Tcl is used during the CBC-AES-128 verification, in order to automatically simulate and verify the design's correctness against the FIPS test vectors.

7.1.2 Wireshark

Wireshark [16] is one of the most famous traffic analyzers. Its primary function is to monitor a network interface (e.g., Ethernet, Wi-Fi, etc.) and display the packets moving through it. It is extensively used during the IPsec implementation and testing phase to aid in software debugging.

Wireshark conveniently presents packets, recognizing a vast range of protocols and displaying all fields of each protocol in a structured manner. Additionally, packets can be displayed in their ”raw” form, i.e., in binary or hexadecimal representation, a feature that is particularly helpful in the debugging of cryptographic algorithms. For the AH and ESP protocols, it provides automatic integrity checking and decryption if the specific SA is provided. Lastly, it offers traffic filtering with a wide range of options which is useful for displaying only packets of interest and avoiding a noisy display of all packets present in real networks.

7.1.3 Scapy

Scapy [17] is a tool based on the Python programming language. It provides a command language for creating packets of any layer of the stack. It can also send the crafted packet and receive and report responses to them. With Scapy, both predefined packets such as IP, TCP, UDP, etc., can be created as well as packets from their hexadecimal form (Raw).

Scapy is used for debugging and verification of the system's inbound dataflow. By connecting a PC to the FPGA, we can create and send individually crafted packets targeting specific behaviors of the input code of IPsec.

7.1.4 Linux ipsec-tools

The ipsec-tools is the implementation of IPsec for the Linux operating system. It allows us to define SPs and SAs either manually or using IKE.

ipsec-tools is used in the final phase of the implementation as a reference system for the final verification stage. Specifically, ipsec-tools is installed on a computer which is connected to the FPGA. Then, SPs and SAs are defined manually on both the computer and the FPGA. Theis setup allows for verification of the implementation against a real-world system running IPsec.

One might notice that using Scapy in debugging is redundant since ipsec-tools can replace its functionality. The power of Scapy comes from allowing segmented testing, providing fine-grained access to every packet detail, whereas ipsec-tools would require significant effort in additional development to achieve the same.

7.2 Final System Design and Implementation

Figure 7.1 depicts the high-level diagram of the complete system. The diagram distinguishes which parts are implemented on the FPGA chip and which are located on the board. Microblaze, its peripherals, and a local memory are situated within the FPGA chip. Microblaze is connected to its peripherals via a PLB bus. All peripherals are memory-mapped. The interfaces of the CBC-AES-128 and HMAC-SHA1-96 peripherals are also presented, detailing their registers.

The FPGA chip, external RAM, the serial port (UART) interface and the Ethernet interface are located on the board.

Received incoming packets are placed in a FIFO by the Ethernet interface. The next available packet in FIFO is moved from the Ethernet interface to the Ethernet Controller when the latter is ready. The Ethernet Controller then notifies the Microblaze via an interrupt. The Microblaze interrupts its operation and executes the lwIP Ethernet code responsible for incoming packets, which stores the new packet in RAM, processes the Ethernet packet, and then passes the execution control to the protocol encapsulating the Ethernet packet. In the case of an IP packet, ip_input is executed next. In the case of an IPsec packet, the payload of the IP packet is passed to ipsec_input, which checks the SPD and SADB databases, distinguishes between AH or ESP packets, and executes the appropriate function. When it reaches the point of authentication and/or decryption, the execution control proceeds to the corresponding driver. The driver copies blocks of the payload from RAM to the peripheral, issues the

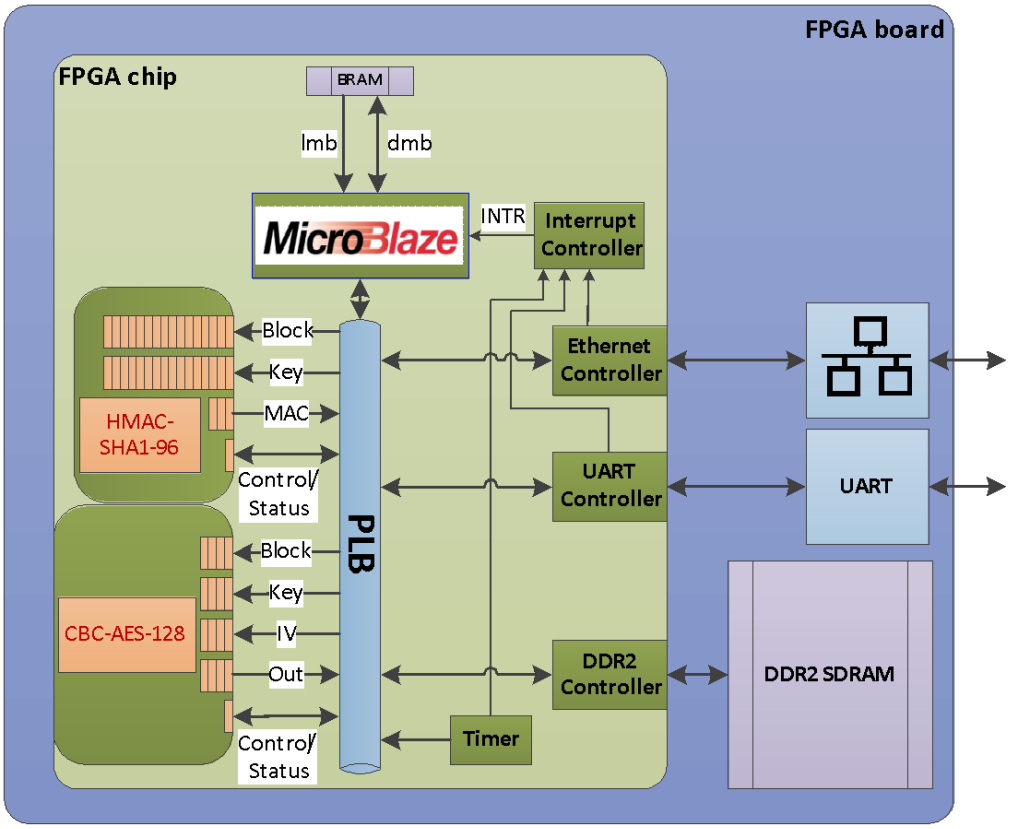

Figure 7.1: Complete system architecture.

appropriate commands, waits for a response from the peripheral, and finally copies the result back to RAM. Once IPsec processing is completed, the payload of the IPsec packet is passed to the appropriate higher-layer protocol. The process of processing outgoing packets follows the reverse flow, starting from a higher-layer packet and ending in the output FIFO and finally transmission via the Ethernet interface. Lastly, during IPsec execution, debugging messages are generated and sent through the UART interface to a PC.

7.3 On-Chip Verification

An ECHO server is implemented for testing the system's operation in a near-real application scenario. The ECHO server accepts requests that carry a message and responds to the requester, echoing back the same message. The server is deployed on the FPGA. The setup is shown in Figure 7.2.

A Ruby script served as a client of the ECHO server. It establishes a connection with the server and then sends ECHO requests. The code snippet shown in Listing 7.1, serves as an example of a client. It connects to the ECHO server and then sends 1000 randomly sized messages with a payload ranging from 5 to 50 random uppercase characters. If a disparity between the request and response arises, the transmission halts and the fault is reported. Finally, the connection is closed.

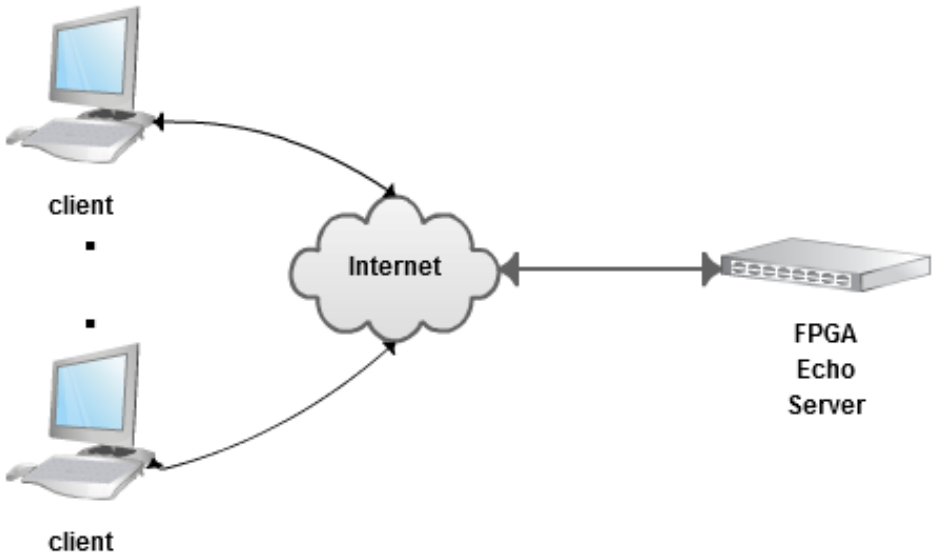

Figure 7.2: ECHO server in Transport Mode

```
require ''socket
cl=TCPSocket.new(Server_IP, 7)
for i in 0..1000
      random_message = (0..(rand(50)+5).map{ (65+rand(26)).chr }.join
    cl.puts random_message
    if cl.gets != random_message then
        puts "Unexpected "Response
        break
    end
end
cl.close
```

Listing 7.1: ECHO client Ruby implementation

The preparation of the setup includes configuring the policies in the SPD and the SAs in the SADB [18] of both systems and setting up Wireshark for collecting the relevant packets. The system is tested for both protocols (AH and ESP) using a large number of packets. As an example, the packets of one request-response scenario are presented in Figures 7.3, 7.4, 7.5, 7.6, 7.7 and 7.8.

ESP with CBC-AES-128 and HMAC-SHA1-96 in Transport Mode

Initially, an ECHO Request with the message ”abc” is sent from the client to the server located on the FPGA. The encrypted ESP packet is presented in Figure 7.3. The colored fields are, in order of appearance:

- Ethernet Header
- IP Header
- SPI
- Sequence Number
- IV
- Payload

- ICV

Figure 7.4 depicts the decrypted payload. The first field is the TCP packet, which, as indicated in the ASCII view, contains the ”abc” message. The padding, pad length, and next header follow as part of the second field. The last field is the ICV.

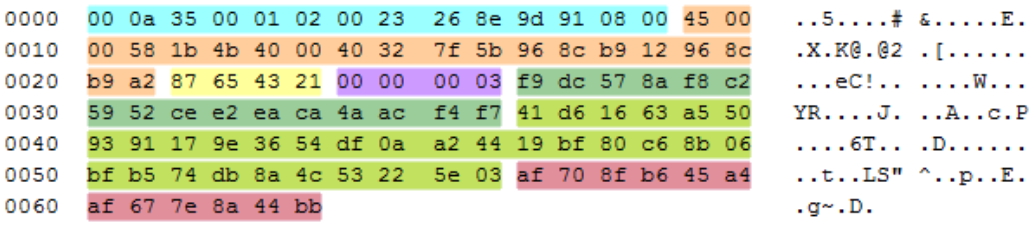

Figure 7.3: Transmitted packet of ECHO request in ESP Transport Mode

Next, the ECHO server on the FPGA responds. The response is shown in Figure 7.5.

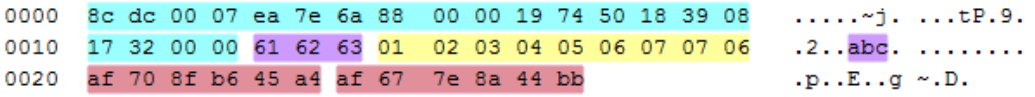

Figure 7.4: Decrypted payload of the ECHO request in ESP Transport Mode

We observe the presence of the SPI, Sequence Number, IV fields, the encrypted payload, and the ICV. The decrypted packet of the response is shown in Figure 7.6.

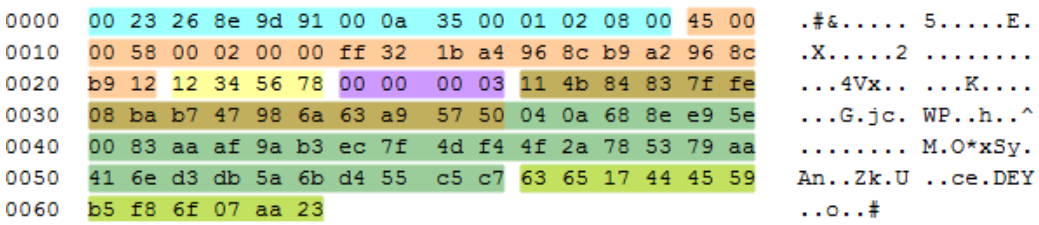

Figure 7.5: ECHO server response in ESP Transport Mode

We observe that the original message of the request, ’abc’, exists in the decrypted payload.

AH with HMAC-SHA1-96 in Transport Mode

Initially, an ECHO Request with the message ”abc” is sent from the client to the server located on the FPGA. The AH packet is presented in Figure 7.7. The colored fields are in the order of appearance:

- Ethernet Header
- IP Header
- Next Header
- AH Length
- Reserved
- SPI
- Sequence Number

- ICV
- TCP Payload
- ECHO Payload

```
0000  00 07 8c dc 00 00 19 74  ea 7e 6a 8b 50 18 08 00   .......t .~j.P...
0010  48 37 00 00 61 62 63 01  02 03 04 05 06 07 07 06   H7..abc. ........
0020  63 65 17 44 45 59 b5 f8  6f 07 aa 23               ce.DEY.. o..#
```

Figure 7.6: Decrypted payload of ECHO server response in ESP Transport Mode

```
0000  00 0a 35 00 01 02 00 23  26 8e 9d 91 08 00 45 00   ..5....# &.....E.
0010  00 43 11 15 40 00 40 33  89 a5 96 8c b9 12 96 8c   .C..@.@3 ........
0020  b9 a2 06 04 00 00 87 65  43 21 00 00 00 03 b8 4c   .......e C!.....L
0030  4d 15 2f cb d1 9a 65 43  f0 98 8c f7 00 07 e1 59   M./...eC .......Y
0040  1b 49 00 00 19 8f 50 18  39 08 6f 60 00 00 61 62   .I....P. 9.o`..ab
0050  63                                                c
```

Figure 7.7: Transmitted packet of ECHO request in AH Transport Mode

```
0000  00 23 26 8e 9d 91 00 0a  35 00 01 02 08 00 45 00   .#&..... 5.....E.
0010  00 43 00 02 00 00 ff 33  1b b8 96 8c b9 a2 96 8c   .C.....3 ........
0020  b9 12 06 04 00 00 12 34  56 78 00 00 00 03 ee 9e   .......4 Vx......
0030  63 cf bb 69 66 ef 9e 6f  0b ea 00 07 8c f7 00 00   c..if..o ........
0040  19 8f e1 59 1b 4c 50 18  08 00 a0 65 00 00 61 62   ...Y.LP. ...e..ab
0050  63                                                c
```

Figure 7.8: ECHO server response in AH Transport Mode

The FPGA's response is shown in Figure 7.8. We observe the presence of the fields in the AH header, as well as the correct content in the ECHO payload.

7.4 Evaluation

Data collection of performance metrics is performed. The purpose of these measurements is to estimate the system's performance and calculate the efficiency gains that the hardware cryptographic algorithms offer compared to a software-only implementation. The metric measured is the number of clock cycles required for processing a packet from the moment it enters the processing path we want to measure. An on-chip timer is used for counting clock cycles. When IPsec processing begins, the timer is initialized and activated. Upon activation, the timer increments its counter value by one at each clock cycle. To measure the processing time of a particular path, we store the timer value at the beginning and end. Subtracting these two quantities yields the number of clock cycles that the specific execution path took.

For comparison with a software-only implementation, the same system is modified using software cryptographic algorithms. The algorithms are implemented based on the OpenSSL package, with minor modifications for porting the code to our system. This decision is made because the implementation of the algorithms in OpenSSL is highly optimized and thus it provides a competitive comparison.

The measurements were conducted using the ESP protocol since it uses both cryptographic algorithms. The measurements are divided into two parts, one for the processing of incoming packets

Payload size (bytes)	HW/SW	AES	HMAC	ESP	IPsec	ESP Header Processing	IPsec Header Processing
3	SW	772693	279229	1054804	1057353	2882	2549
	HW	17625	49741	70174	72724	2808	2550
	Speedup	**43.84**	**5.61**	**15.03**	**14.54**	-	-
10	SW	771658	279396	1053938	1056474	2884	2536
	HW	17598	49697	70104	72640	2809	2536
	Speedup	**43.85**	**5.62**	**15.03**	**14.54**	-	-
100	SW	3062302	316804	3381984	3384517	2878	2533
	HW	59600	63569	125973	128503	2804	2530
	Speedup	**51.38**	**4.98**	**26.85**	**26.34**	-	-
500	SW	12619006	589613	13211500	13214036	2881	2536
	HW	234658	132484	369958	372492	2816	2534
	Speedup	**53.78**	**4.45**	**35.71**	**35.47**	-	-
1000	SW	24467280	947749	25417904	25420434	2875	2530
	HW	451749	226794	681356	683890	2813	2534
	Speedup	**54.16**	**4.18**	**37.3**	**37.17**	-	-

Table 7.1: Inbound packet processing time (in number of clock cycles)

and one for the processing of outgoing ones. Additionally, various payload sizes are used to provide a complete performance picture, as there are parts of the system whose processing time is dependent on the packet size. The paths measured include the total processing of IPsec, the ESP processing, and both of the cryptographic algorithms. It should be noted that the measurement of IPsec includes ESP processing, and ESP includes the cryptographic algorithms execution. The measurements are presented in Table 7.1 for inbound traffic and in Table 7.2 for outbound traffic. The quantities are expressed in the number of clock cycles.

The tables include a row labeled 'Speedup,' indicating how many times faster the hardware implementation of the measured part is compared to the software implementation. This calculation is the division of the number of clock cycles of the software implementation by those of the hardware implementation. Two additional columns are added for the processing time of only the ESP header (i.e., without the cryptographic algorithms) and for the processing time of the IPsec without ESP processing (i.e., including the IPsec header processing and searches in the SPD and SADB).

First, we observe that the system benefits from the implementation of cryptographic algorithms in hardware for both inbound and outbound traffic, for every examined packet size. This is evident from the positive values in the Speedup row of the IPsec column. Furthermore, we notice that the Speedup of IPsec increases as the packet size grows. Further analysis of the measurements reveals that the dominant factor in the system's speedup is CBC-AES-128. HMAC-SHA1-96 contributes to the speedup but to a much lesser extent. Additionally, the trend of the speedup to increase as packets grow larger is also attributed to CBC-AES-128. Conversely, HMAC-SHA1-96 appears to decrease in speedup as packet size increases, but this trend is less pronounced than that of CBC-AES-128 and does not have a significant impact on system performance. Finally, the processing of headers and searches in SPD and SADB remains consistent, as expected since their implementation remains unchanged. Any slight deviation observed between them is due to non-deterministic factors such as the search time in databases. Lastly, when comparing the measurements between inbound and outbound traffic, a difference in the processing of the ESP header is identified. This difference is attributed to the fact

Payload size (bytes)	HW/SW	AES	HMAC	ESP	IPsec	ESP Header Processing	IPsec Header Processing
3	SW	543519	269921	867940	871315	54500	3375
	HW	17600	44479	117063	120437	54984	3374
	Speedup	**30.88**	**6.07**	**7.41**	**7.23**	-	-
10	SW	813759	267691	1137568	1140943	56118	3375
	HW	17587	44484	116538	119899	54467	3361
	Speedup	**46.27**	**6.02**	**9.76**	**9.52**	-	-
100	SW	2165365	312021	2534846	2538223	57460	3377
	HW	59585	58280	175613	178991	57748	3378
	Speedup	**36.34**	**5.35**	**14.43**	**14.18**	-	-
500	SW	8911300	582429	9560500	9563879	66771	3379
	HW	234655	127235	429579	432952	67689	3373
	Speedup	**37.98**	**4.58**	**22.26**	**22.09**	-	-
1000	SW	17275837	943116	18296571	18299937	77618	3366
	HW	451705	221543	751779	755145	78531	3366
	Speedup	**38.25**	**4.26**	**24.34**	**24.23**	-	-

Table 7.2: Outbound packet processing time (in number of clock cycles)

that during header creation in outbound traffic, a random IV and padding are generated, processes that do not exist in inbound processing, which leads to increased memory copies compared to inbound processing.

From the above measurements, the system's throughput can be easily calculated using Equation 7.1, where $\#bits$ represents the number of bits processed in time.

$$Throughput = \frac{\#bits}{time} \tag{7.1}$$

Knowing the number of clock cycles and the operating frequency of the system, we can determine how much time is required to process a packet using Equation 7.2, where $f = 100MHz$ is the operating frequency and $\#clk$ is the number of clock cycles.

$$time = \frac{\#clk}{f} \tag{7.2}$$

Combining Equations 7.1 and 7.2 we get the final throughput formula in Equation 7.3.

$$Throughput = \frac{\#bits * 100 * 10^6}{\#clk} \tag{7.3}$$

Using Equation 7.3, we can calculate the throughput of our system across all packet sizes. Here, it should be noted that $\#bits$ represent the size of the packet payload, excluding the header, padding, IV, and trailer. Table 7.3 presents the throughput calculations for the aforementioned payload sizes for inbound and outbound traffic. The Graph 7.9 summarizes the table. We observe significant improvement from the use of hardware cryptographic algorithms, especially for medium to large packets. Furthermore, we notice that the throughput trend is increasing as packet size increases.

Furthermore, the area utilization of the implemented system and the peripherals are measured. Table 7.4 summarizes the results. The percentages refer to the peripheral area relative to the total system area.

#bits	Outbound		Inbound	
	HW	SW	HW	SW
3	19.93	2.75	33	2.27
10	66.72	7.01	110.13	7.57
100	446.95	31.52	622.55	23.64
500	923.89	41.82	1073.85	30.27
1000	1059.4	442.62	1169.78	31.47

Table 7.3: Throughput (KB/s)

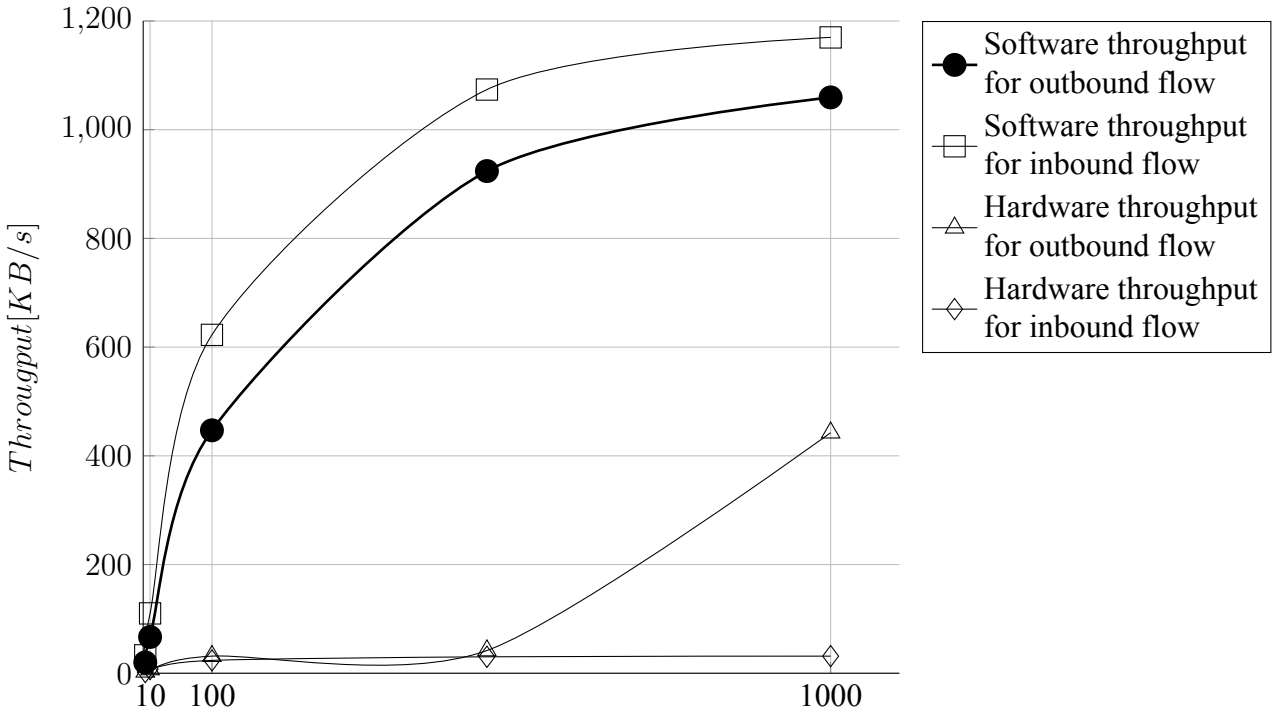

Figure 7.9: Throughput comparisons across SW/HW and inbound/outbound flows

Metrics	CBC-AES-128	%	HMAC-SHA1-96	%	System Total
Registers	1808	15.41	1535	13.08	11732
LUTs	3018	24.12	1318	10.53	12511
Slices	1108	18	456	7.41	6154

Table 7.4: Area utilization

8. Roadblocks and Solutions

During the course of this thesis, various difficulties were encountered in all phases of system development. Here, we briefly mention the most significant ones.

Memory Management of lwIP

lwIP has its own memory management library. The function used to allocate a memory block is mem_malloc. This function is utilized by lwIP in any memory allocation operation, such as pbuf_alloc, which allocates memory for a new pbuf.

During the initial phase of IPsec software implementation, memory management was handled by the native C library. However, memory allocation by the IPsec memory management didn't interoperate with lwIP memory management, resulting in allocation conflicts and memory overlaps which created unintended data overwrites. This occurred because there was no coordination between the two memory managers regarding which memory areas are allocated at a given point in time. Consequently, for example, a pbuf allocated in memory by lwIP would be lost or corrupted when memory area overlapping with pbuf was allocated by IPsec. This confusion in the implementation process was resolved by appropriately modifying the memory management of IPsec to use mem_malloc, which updates the lwIP memory manager for the memory allocations made by IPsec.

Hardware Design and Implementation

On many occasions during the hardware implementation phase, a design seemed to work perfectly in simulations but behaved unexpectedly when deployed on the FPGA chip.

During the implementation of the two peripheral modules, CBC-AES-128 and HMAC-SHA1-96, the aforementioned difficulty was encountered multiple times, leading to several reconsiderations of the design. The first attempted AES design was logically correct and passed all simulations successfully. When deployed on the FPGA though, it was outputting results that were unrelated and seemed random. The culprit was that a control signal that was driving several MUXes although logically has a constant value across all units, in reality, it suffered from delays and glitches that made it inconsistent across the system.

The synthesizer tries to adhere to the timing constraints necessary for the circuit to function correctly. However, the presence of feedback paths and the absence of registers to partition them make its job more difficult, and some constraints may not be strictly adhered to. The problem was be solved by placing registers on the feedback paths. However, this approach would divide the path into two parts, doubling the processing time of each AES round, and would also impose changes in the FSM's operation. The final solution avoids these drawbacks by decoupling processing paths and creating a main path without feedback, at the cost of a small additional area utilization. The new design makes it clear to the synthesizer which delays it should take into account and where.

Another point of difficulty was the operating frequency of peripherals. Since the entire system operates with a clock of 100MHz, peripherals should also be able to operate at a frequency of at least 100MHz. The initial implementations of peripherals did not reach the system frequency, and optimizations were made until this frequency was exceeded by an additional safety margin.

Finally, there was a subtle issue in the processor's interface with the peripherals. The peripherals' command signals are set by the processor to '1' or '0'. When a pulse needs to be issued to a control signal, the appropriate bit pattern needs to be written to the memory location that maps the control register. The pattern for a positive pulse is to first set it to '1', and then rewrite it to set it to '0'. However, a writing operation can take several clock cycles, so there is a possibility that we may not set the control signal to '0' before the peripheral reaches a state where it checks its value, resulting in an unintended command. To ensure correct operation in such cases, circuits were designed and placed in the control signals of the peripherals, which generate a one-clock cycle pulse from a rising edge. That removes the need to erase the previous command right after issuing it but instead, the previous command is erased before the new command is issued.

Drivers

During the implementation of the peripheral drivers, it was observed that although the correct commands were given to the peripherals, the peripherals were unresponsive. The culprit was the optimizations applied to the driver code by the compiler. The code for setting control signals takes the form presented in Listing 8.1.

```
control_register = value1;
control_register = value2;
```

Listing 8.1: Example of setting peripheral control signals

The compiler, seeing that the final value of the control_register is value2 and that as long as it has the value value1 it is not used anywhere, decides to optimize it away, resulting in the code shown in Listing 8.2.

```
control_register = value2;
```

Listing 8.2: Setting peripheral control signals after compiler optimization

One way to solve this issue would be to completely disable compiler optimizations. However, this will lead to the whole software being unoptimized, resulting in a significantly slower execution and higher memory consumption. Another way is to instruct the compiler to not optimize only a part of the program using the method presented in Listing 8.3.

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
// Code section we don't want to optimize
#pragma GCC pop_options
```

Listing 8.3: Instructing the compiler to not optimize a code segment

Unfortunately, these commands to the compiler were not taken into account, which may be a design choice of the compiler developers. Ultimately, the code had to be written to trick the compiler into believing that the first value of the signal was used in the program by forcing a write to both values. For this reason, a function was created that simply sets the value of its argument to the control_register variable, as shown in Figure 8.4.

```
void set_control_register(u32_t value)
{
  control_register = value;
}
```

Listing 8.4: Workaround to stop the compiler from optimizing a code segment

and the original code gets transformed to the one shown in Listing 8.5.

```
control_register = value1;
set_control_register(value2);
```

Listing 8.5: Successfully setting peripheral control signals

The last approach indeed solved the problem, as the compiler sees the control_register variable being used in a function. It should be noted here that compilers are quite smart in optimizations, and perhaps, in a different case, a completely different approach would be needed to trick it. Furthermore, the processor in our system operates without a cache. If a cache exists, then the problem cannot be solved using this method either, as changes in the values of the control_register variable would be performed in the cache, leading again to writing only the final value to the peripheral. In this case, other techniques would have to be used, such as placing the control registers outside the cacheable memory region or forcing cache refresh, etc.

Bibliography

[1] Sheila Frankel, K. Robert Glenn, and Scott G. Kelly. The AES-CBC Cipher Algorithm and Its Use with IPsec. RFC 3602, September 2003.

[2] Secure hash standard (shs), 2002-12-01 2002.

[3] Bob Hinden and Dr. Steve E. Deering. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[4] Karen Seo and Stephen Kent. Security Architecture for the Internet Protocol. RFC 4301, December 2005.

[5] Stephen Kent. IP Authentication Header. RFC 4302, December 2005.

[6] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005.

[7] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.

[8] Tony Hansen and Donald E. Eastlake 3rd. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634, August 2006.

[9] K. Robert Glenn and Cheryl R. Madson. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404, November 1998.

[10] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[11] Xilinx. ML505/ML506/ML507 Evaluation Platform User Guide, May 2011. Available at https://docs.xilinx.com/v/u/en-US/ug347.

[12] Xilinx. MicroBlaze Processor Reference Guide. Available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf.

[13] Xilinx. lwIP Library Documentation. Available at http://www.xilinx.com/ise/embedded/edk91i_docs/lwip_v2_00_a.pdf.

[14] lwIP Wiki. http://lwip.wikia.com/wiki/LwIP_Wiki.

[15] Xilinx. A Hands-On Guide to Effective Embedded System Design. Available at http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/edk_ctt.pdf.

[16] Wireshark Wiki. http://wiki.wireshark.org/.

[17] Scapy Documentation. http://www.secdev.org/projects/scapy/doc/usage.html.

[18] Setkey(8): Manually change IPsec SA/SP Database - Linux Man Page. https://linux.die.net/man/8/setkey.

[19] Sheila Frankel. Demystifying the Ipsec Puzzle. Artech House, Inc., USA, 2001.

[20] A.S. Tanenbaum. Computer Networks. Prentice Hall PTR, 2003.

[21] P.J. Ashenden and University of Adelaide. Department of Computer Science. The VHDL Cookbook. Department of Computer Science, University of Adelaide, 1991.

[22] K.C. Chang. Digital Systems Design with VHDL and Synthesis: An Integrated Approach. Systems. Wiley, 1999.

[23] Rashmi Ramesh Rachh, P. V. Ananda Mohan, and B. S. Anami. Efficient implementations for aes encryption and decryption. Circuits, Systems, and Signal Processing, 31(5):1765–1785, Oct 2012.

[24] Christian Scheurer and Nilklaus Schild. Embedded ipsec, a lightweight ipsec implementation. Master's thesis, HTI Biel/Bienne, 2003.

University of Patras, School of Engineering
Department of Electrical and Computer Engineering
Georgios Gkitsas