ESE 441 Senior Design, Spring 2018
Electrical and Computer Engineering Department, Stony Brook University

Hardware/Software System for Accelerating Cryptography: An Application in Internet Of Things

Authors:

Hsiang-Ju Lai
Edward Wang
Alex Lin

Project Advisor:

Prof. Peter Milder

05/14/2018

Abstract

The report discusses in detail the hardware and software design of a crypto-accelerator for IoT devices. The design is realized on a development board with an SoC which interfaces with a given IoT device with one of the myriad interfacing ports on the the board. The device acts as a middleman between IoT devices and the remote users to provide an extra layer of security encryption. An IoT camera module is used as an example in this project. The processor retrieves and processes the data from the camera, utilizes the AES crypto-accelerator built on the FPGA for fast encryption, and sends the ciphertext to the remote users.

Table of Contents

Section 1. Goals and Impacts **2**

Section 2. Background **4**

2.1 Survey 4

2.2 Project Planning 5

Section 3. System Implementation and Testing **7**

3.1 Design Constraints 7

3.2 Implementation Problems 8

3.2.1 FPGA Logic Capacity & Timing Issues 8

3.2.2 Endianness 9

3.2.3 Physical Memory Allocation 9

3.2.4 PetaLinux Package Manager 9

3.2.5 Vision Processing Library 10

3.2.6 Encryption Library 10

3.3 Final Implementation 10

3.3.1 Usage of Equipment, Parts, Software Packages 10

3.3.2 System Overview 12

3.3.3 Network Communication Module 14

3.3.3.1 Communication Protocol 15

3.3.3.3 Encryption/Decryption Implementation 16

3.3.4 Cryptography Accelerator 17
3.3.4.1 PS-PL Communication 17
3.3.4.2 AES Encryption Process and Logic 21
3.3.4.4 Advanced eXtensible Interface Stream (AXIS) Protocol 29
3.3.4.5 Kernel Space Memory Allocator 30
3.3.4.6 User Space DMA/AES Driver 31
3.3.4.7 AES128 User Program 32
3.3.5 IoT Device Interface 32
3.3.5.1 Detail Description of the Interface 33
3.3.5.2 IoT Vision Processing Unit 34
3.4 Testing 35
3.4.1 Functional Simulation 35
3.4.2 System Verification 36
3.4.3 Client Test Verification 38
3.4.3.1 Android Client Verification 38
3.4.3.2 Computer Client Verification 38

Section 4. Results and Discussions 39
4.1 Result Analysis 39
4.1.1 Timing Analysis 39
4.1.1 System Throughput 42
3.4.4 Energy Efficiency 45
4.2 Multi-Disciplinary Issue 47
4.3 Professional/Ethical Considerations 48
4.4 Impact of Project on Society/Environment 48

Section 5. Summary and Conclusions 49

Acknowledgements 50

References 51

Section 1. Goals and Impacts

Data security is integral to any technology, and with the increase of Internet of things (IoT) devices, many of them have limited security due to the constraints on their computing power in the embedded environment. IoT devices have a range of different mediums. Some examples are common household devices that are connected to the Internet for users to collect information or control their house functionality. In most cases the data being collected may contain sensitive information and is not secured if it is sent through a public network without encryption. Current solutions include hosting local servers, either rack-mounted or smaller, and setting up smart hubs.

The myriad problems with rack mounted local servers start with the cost for functionality. The pricing for a local server is usually much more than a standard IoT device considering servers can perform a multitude of different services. If these services are not fully utilized, it makes it infeasible for one to pay for such a device solely for the sake of data security. Another issue is related to the server size. Servers are large, bulky appliances that require space for storage and specific storage requirements such as air conditioning and ventilation to handle the excess heat release. Smaller servers that are run from a computer or devices such as Raspberry Pi's or personal computers are less expensive than rackmount solutions. However, with a smaller device the processing speed slows down when interfacing and managing multiple IoT devices. Smaller servers are also not designed to be as secure as rack mounted options. Another major issue is how local servers do not fully protect IoT devices. By having IoT devices connect wirelessly to a server, they are still subject to hacker attacks.

Smart hubs are devices that work similar to server options by managing multiple IoT devices. Smart hubs are provided by a manufacturer and the problem with this is how the user has less control over the entire system. If not hackers, the data collected by IoT devices may not be safe since it can be collected by corporation itself. By having devices that are constantly waiting for keywords or faces to activate them, events surrounding the device will be collected. Also, not all IoT devices work with a given smart hubs. There are smart hubs such as the Panasonic hub which works only with Panasonic smart devices.

One possible solution is to have the embedded microprocessors in IoT devices to encrypt their outgoing data. Currently there exists multiple high-quality cryptography algorithms, however, many of them consume a large amount of computing power. In most embedded environments, IoT devices have limited computing capacity, thereby making it difficult to provide both efficient and secure data to users through the internet.

In response to these issues, our solution introduces a dedicated SoC, or System on Chip, with a built-in FPGA. The SoC acts as a secure server on the public internet instead of the IoT

device. The chip can interface with a number of IoT devices with the variety of ports available on the chip. The SoC also handles data security for IoT devices by either working alone or introducing an additional layer to the existing data solution that is using a local server. This device allows users a level of abstraction for their connected IoT products to perform authentication, key exchanges, data encryption and many more security features to ensure data security. This is achieved by running encryption procedures, that would normally be taxing on IoT embedded systems, onto hardware, specifically the FPGA, to increase performance and avoid overhead. This type of device is minimal in size compared to the IoT device and the cost-efficiency ratio is higher than existing solutions because it only requires a low cost embedded microcontroller while using a dedicated hardware chip for encryption. In our design, we demonstrate the application of this design by using an IoT camera.

Section 2. Background

2.1 Survey

There has been a number of research conducted prior to this project on the topic of hardware crypto-accelerators. However, only a minimal amount of these studies focus on the application of this hardware for the purpose of integrating it with IoT, or Internet of Things, devices. Therefore, both research done on IoT device security and crypto-accelerators are studied and referenced.

The first paper referenced is, “Interfacing a high speed crypto accelerator to an embedded CPU” [1]. The authors identify the challenge of choosing an efficient interface for a hardware cryptographic accelerator to satisfy the required throughput of security applications. The communication interface between the embedded CPU and the hardware accelerator is a key factor of the overall system performance. The paper compares two kinds of interfaces, the memory-map interface and the Co-Processor Interface (CPI). The former interface uses a bus controller to connect the hardware accelerator to the system bus, which allows the accelerator to perform read and write operations to the memory directly. The latter interface, on the other hand, is only accessible from the CPU, which allows the CPU to send data to the accelerator directly. The cryptographic algorithm used in the paper is Advanced Encryption Standard (AES). The performance of the systems with AES hardware accelerators is also compared with an AES customized processor. The conclusion shows that the AES-accelerators with both interfaces outperform the AES customized processor, and that the memory-map interface has better overall performance than the CPI interface [1].

Two more papers referenced are, “Lightweight Cryptography for the Internet of Things” [2] and “SIT: A Lightweight Encryption Algorithm for Secure Internet of Things” [3]. The two articles discuss the need and the importance of fast data encryption in IoT devices. Instead of creating hardware accelerator for the IoT devices, they try to solve the problem by proposing a lightweight cryptography algorithm to be run on the IoT device. Most of the proposed algorithms are based on block ciphering, from which we learn that block cipher algorithms may be the best candidates for our IoT security application since they are generally fast and require less memory. From the former paper, we also realize the two major requirements of encryption in IoT. First and foremost, efficient end-to-end communication is necessary. This includes the time efficiency, space efficiency, and energy efficiency of secure end-to-end communication. Second, the applicability of the solution to the devices with limited resources. Our proposed design should work with most IoT devices regardless of their computing resources, which means that our design should never rely on the computing power of the IoT nodes.

The feasibility of implementing a cipher accelerator on FPGA using hardware description language is also studied. According to the paper “A VHDL Implementation of the Advanced Encryption Standard-Rijndael Algorithm” by Rajender Manteena, “Field Programmable Gate Arrays (FPGAs) offer a quicker, more customizable solution (to AES)” [4]. The goal of the paper is to investigate the design of the AES algorithm on FPGA with VHDL. It starts with the AES mathematical backgrounds and computation process for both encryption and decryption. The procedure of the block ciphering process is introduced. It seems that such algorithms relies mainly on XOR and table lookup operations. Based on our experiences in digital design, we believe that both operations can be easily implemented on FPGA. In addition, the paper concludes that AES can be implemented efficiently on FPGA with optimized and synthesizable VHDL code. Therefore, we can make sure that it is feasible to create hardware cipher on FPGA using VHDL.

2.2 Project Planning

The skills required for this project involved a variety of digital design, algorithms, security, computer interfacing and programming language understanding. The project started off at deciding the development board that will be used. It was necessary to find a board that can fit the given budget with a variety of ports that can be used to interface with the different types of connections that the IoT device will require. The IoT device that connects to this also needed to be defined. The IoT device should be applicable to real life situations and same as the board, it needed to be within the budget. Following this, the board needed to be able to communicate in a server client relationship. Therefore it was necessary to understand the type of server-client relationship that was going to be required to transfer and receive data and requests between the given client and development board.

For the FPGA to ARM handshake it was necessary to understand how the development board handled the interface. This required reading documentation and watching tutorials from the official website. It was recommended to write an adder in FPGA and use a programming language like C. Likewise when doing this example project, it used the memory handler for the development board. Therefore, for the memory handler, it was necessary to understand how the AXI/DMA works and how to use it within the board.

For the server client relationship, it was necessary to understand the responsibilities for the SoC, system on chip. The chosen communication protocol was a socket to socket connection. The custom protocol required understanding all the necessary steps it took to open a socket to connect with the client and to transfer encrypted data. Looking into the development tutorials for the chosen board it was also necessary to see if the MiniZed board supported Wi-Fi connectivity to allow for this feature. There was also a need to understand the standards and constraints when it comes to sending data, requesting certain commands such as implementing the hardware acceleration of the chosen encryption algorithm.

The next section is the encryption algorithm. The cryptographic algorithm required an understanding of AES in order to understand how this algorithms operates and provide the level of security that is used as a standard for multiple systems. From this, it was necessary to find a profiling tool to locate the bottleneck of the algorithms so that we understand where the acceleration needs to occur. Also, it needed to be understood what part of the algorithm will be accelerated, and how we would interface with the given ARM chip on the development board. Understanding of digital design was necessary for the FPGA programming and using the PS-PL interface.

The project was divided into three major modules. A list of goals and steps for each module is presented below. A tentative timeline is shown in Figure 1.

1. Secure Network Communication
    a. Setup Wi-Fi configuration on the development board.
    b. Create server program that implements socket to socket connection with GStreamer camera streaming capabilities.
    c. Create client program that implements socket to socket connection with AES decryption capabilities.
    d. Create a test program that sends encrypted text from the board to client device.
    e. Improve the test program to be capable of sending encrypted images and video streaming.
    f. Try to send continuous data stream from the board to PC or mobile phone. Handle network traffics issues and estimate the data rate that can be transfer stably.

g. Create the complete server program with demo application on the client side to get encrypted stream.

2. Crypto-accelerator on FPGA
   a. Study the documentation and the tutorial videos to understand how the SoC works on the development board.
   b. Develop a simple multiplier on FPGA and write a C program running on the processor to use the multiplier. The purpose of this step is to make sure we understand the interface between the processor and the FPGA.
   c. Use profiling tools to find the bottleneck of the encryption program used in the Secure Network Communication module.
   d. Design a hardware accelerator to boost the cryptographic program. Meanwhile, develope testbenches to simulate the hardware design.
   e. Develop a kernel driver for the hardware encryptor.
   f. Create an Application Programming Interface (API) for the cryptography accelerator driver.

   g. Write user programs and scripts for the demo with IoT camera.

3. IoT Device Interface (IoT Camera)
   a. Create a C++ program to retrieve image from the IoT camera to a PC (or virtual machine) that runs linux operating system using GStreamer.
   b. Resolve any compatibility issues and build the program on the development board. Ensure the program can interface with the camera correctly on the board.
   c. Convert the IoT camera program into an API which implement the interface.

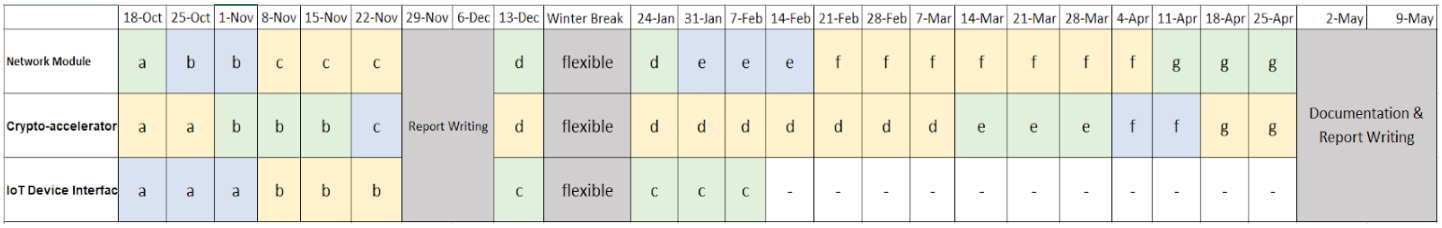

Figure 1: Tentative Project Timeline

Section 3. System Implementation and Testing

3.1 Design Constraints

Design constraints can be broken down into the constraints and standards. Constraints are generated from multiple sources from the physical board and connection to the OS and program. Starting with the driver, the Linux system implemented on the MiniZed has limitations on hardware. For example, UVC, the USB video class, is initially disabled. This means that there are a number of required drivers that need to be toggled on and the configured OS will have to be uploaded onto the MiniZed using the Petalinux tool. This brings in a new constraint as it takes a certain degree of time for the tools to configure and send the bit stream. For the network, traffic and upload/download speed is relative to the location used for the MiniZed, making it a constraint on the process when it performs its procedure. There is also potential data loss and network integrity faults, which can be mitigated by using a checksum to both manage this issue as well as block hackers who use techniques like data injection. For the hardware, the MiniZed board used has limits in its computing power, the ARM processor is a single core at around 667 MHz, FPGA logic units, memory, power consumption and port availability. Pricing also provides another limit, as the amount of funds available can dictate the quality of the final device. For the physical device there is also considerations for an acceptable form format for the FPGA device and IoT device with its quality. There can be cases where the form format can make the unit unusable in certain cases.

Another design constraint is the standards that need to be reached and maintained for the different sections of this project. Starting at the beginning, there are IEEE standards of ethics that must be followed in the design and creation of the project. When working on the physical hardware there are standards for the type of port used, in this case:

- 1532-2002 - IEEE Standard for In-System Configuration of Programmable Devices
- 1619.2-2010 - IEEE Standard for Wide-Block Encryption for Shared Storage Media
- 1619.1-2007 - IEEE Standard for Authenticated Encryption With Length Expansion for Storage Devices
- 1667-2015 - IEEE Standard for Discovery, Authentication, and Authorization in Host Attachments of Storage Devices
- IEEE Std 2600 - IEEE Standard for Information Technology: Hardcopy Device and System Security

As for the MiniZed device, since it uses Ubuntu, there are Linux standards for the drivers used. As mentioned before, the UVC is an example of one of those drivers. However, it’s not just Linux, the tools used to customize the MiniZed device use PetaLinux. With Petalinux, the VHDL

programming language, have coding standards for proper implementation. These are seen when interfacing with the AXI/DMA and network devices, such as the Server/Client connection, Wi-Fi/Bluetooth protocols, IP protocols, API POST/GET call and abstraction formats for usage and implementation of the AES encryption algorithm. As for the server side, there are standards for implementing proper code and documentation for C++, OpenSSL, OpenCV and the Server Client relationship involving HTTPS, IPsec, SSL/TLS.

3.2 Implementation Problems

Although a detailed final design and a complete plan had been proposed in the previous stage, several issues were identified throughout the implementation stages. This section describes the major problems and additional constraints we have encountered and the solutions to those issues.

3.2.1 FPGA Logic Capacity & Timing Issues

The MiniZed development board is a budget, cost-optimized development platform for SoC design. While the board comes with a quite complete computer system, the Zynq-7007S SoC on the board has a relatively limited number of logic cells. When we first attempted to implement the AES encryptor design onto the chip, the implementation tools raised an error indicating the number of LUT on the board was not enough. In fact, the board has 14,400 LUTs and our original PL design required more than 15000 LUTs.

The solution we ended up taking is to revise the hardware design of the encryptor. The encryptor was based on an open-source design that implements AES128 in ECB mode, which has duplicated units for pipeline design to increase the throughput of the system. However, for our design of CBC encryption, this kind of pipeline cannot help with the throughput because each block of encryption depends on the result from the previous block. Therefore, we decided to discard the original top-level design that is using pipeline and implement our own variation, which eventually reduced the number of LUT required down to approximately 7000.

Timing was another issue we have encounter. Each round of the AES encryption requires multiple steps, which add up to a relatively large propagation delay of around 14 ns. Therefore, we had to lower the PL clock frequency down to 72 MHz and configured the synthesizer and implementation tools to seek for optimized timing design. This way, one entire round of AES encryption can be completed in one cycle.

3.2.2 Endianness

Endianness is often an issue when two systems try to communicate. The hardware AES encryption core in the original design has big-endianness, which indicates that the most

significant byte has the lowest address. However, AXI interfaces and the PS both use little-endianness, which means that the byte order must be reversed for the communication between PS-PL. Therefore, to avoid extra works for the PS, the hardware design was modified into little endian design.

3.2.3 Physical Memory Allocation

Since the software programs of the design run on an operating system that is managing the resources including the main memory, the programs can only acquire the memory they need from the OS. To utilize AXI-DMA in simple mode, one must have the physical memory address to a physically consecutive memory buffer. However, the operating system hides the physical address from the user programs, so there is no way to allocate a physically consecutive buffer and know the physical address in user mode. Linux only allows allocation of DMA buffers in the kernel. Therefore, a DMA memory allocation kernel module must be developed and be able to communicate with programs in the user mode through a simple interface.

3.2.4 PetaLinux Package Manager

The PetaLinux Tool provides convenient ways for SoC developers to quickly configure and generate a highly customizable embedded operating system distribution based Linux Kernel. The tool actually relies on a set of programs to carry out different functions. For example. Yocto and BitBake are used to download, manage, and build the packages, modules, libraries, and user applications. A recipe file as well as a Makefile must be written for a user application. The problem is that there is not too much resources and documentation available on how to write the recipes and a PetaLinux compatible Makefile. We could not figure out how the library building and linking process works under PetaLinux. As a result, the user program we have developed that links to the Gstreamer dynamic library was not successfully built on the system. As a backup plan, the Gstreamer user program gst-launch-1.0 is used to carry out the operations.

3.2.5 Vision Processing Library

The original library planned for streaming input of the USB camera module was OpenCV. However, due to memory constraints it is more beneficial to choose a smaller lightweight library, in this case GStreamer. GStreamer is a library of plugins that can placed together into a pipeline that inputs any media form, edit the input and adjust its audio or video format and output the result to a file, video player source or as standard input to another program.

3.2.6 Encryption Library

The initial choice for the encryption library was OpenSSL because of its flexibility and scalability for implementing different encryption algorithms and key generations. The difficulty in using this library are the same as for the OpenCV. The library itself exceeds the memory capacity of the embedded processor when taking into account the other libraries needed and programs. There are also challenges that arose when altering the OpenSSL code because it increased the complications when processing the software through PetaLinux. Therefore this OpenSSL is not used and the inputted stream is sent directly to the hardware cryptographic accelerator with a given key.

3.3 Final Implementation

In this section, the final implementation of the design is discussed in detail. A list of the equipment, parts, and packages is also presented. A top-down approach is used to present the implementation. In other words, the top-level implementation of the system will be discussed first, and then each module will be described respectively.

3.3.1 Usage of Equipment, Parts, Software Packages

Package Name	Company	Serial Number	Usage
MiniZed	Avnet	VPYLB1DX	Serve as the central hardware system including the processing unit and programmable logics
720p USB Camera Module	ELP	X000ZUFL9B	Capture video stream
PC	N/A	N/A	Run tools, IDEs, etc.
Ubuntu 16.04 LTS	Canonical Ltd.	N/A	Support Petalinux tools and other SoC development tools
Vivado Design Suite	Xilinx	N/A	IDE for SoC hardware development
Xilinx SDK	Xilinx	N/A	IDE for SoC software development
PetaLinux Tools	Xilinx	N/A	Setup and configure the operating system on chip
OpenSSL	OpenSSL Community	N/A	Generate keys, certificate and provide framework for calling

			encryption/decryption algorithms.
GStreamer	GStreamer Multimedia Framework	N/A	Capture the image frames from the camera and decode them to H-264 format for encryption.
Cipher	Oracle	N/A	Core Java Cryptographic Extension(JCE) framework used to decrypt the AES images on Android.
Android Kotlin	JetBrain	N/A	Develop Android client using Kotlin programming language.
Netcat	Hobbit	N/A	Create TCP connection for client and server.

Table 1: Equipment / Parts / Software Used

Table 1 is a list of all items that are used in the design. The design is to be realized on the MiniZed development board, with a USB-capable camera module connected to the board. The board has Zynq 7000S SoC, 512MB DDR3L SDRAM, and 8GB eMMC flash memory. It has a number of standard ports, such as USB ports, Arduino headers, and several configurable GPIO pins. It also comes with a built-in Bluetooth 4.1 with BLE module and supports Wi-Fi 802.11 b/g/n, which are widely used wireless standards. These ports and wireless communication modules cover almost all the methods that IoT devices communicate with each other. Therefore, we believe that this board is a reasonable choice for our design relating to IoT development.

3.3.2 System Overview

The main goal of the system is to serve as a secure abstraction layer for IoT nodes without adding much overhead. It has to be easy for an existing IoT management system to incorporate the proposed system. Therefore, the vertical scalability and horizontal extensibility must be taken into account during the implementation phase.

The system runs a secure server in the public network for remote access. An IoT camera is incorporated into the design to demonstrate how the system interfaces with IoT devices. A variety of physical ports and wireless modules on the development board are to be used to communication with local IoT devices. The data streams from all connected IoT devices are encrypted with a hardware crypto-accelerator on the PL and would be sent to the connected remote devices through the public network if requested. The remote devices can decrypt the data with a key that both side previously agreed with. The top-level system design is discussed respectively from the hardware perspective and the software perspective.

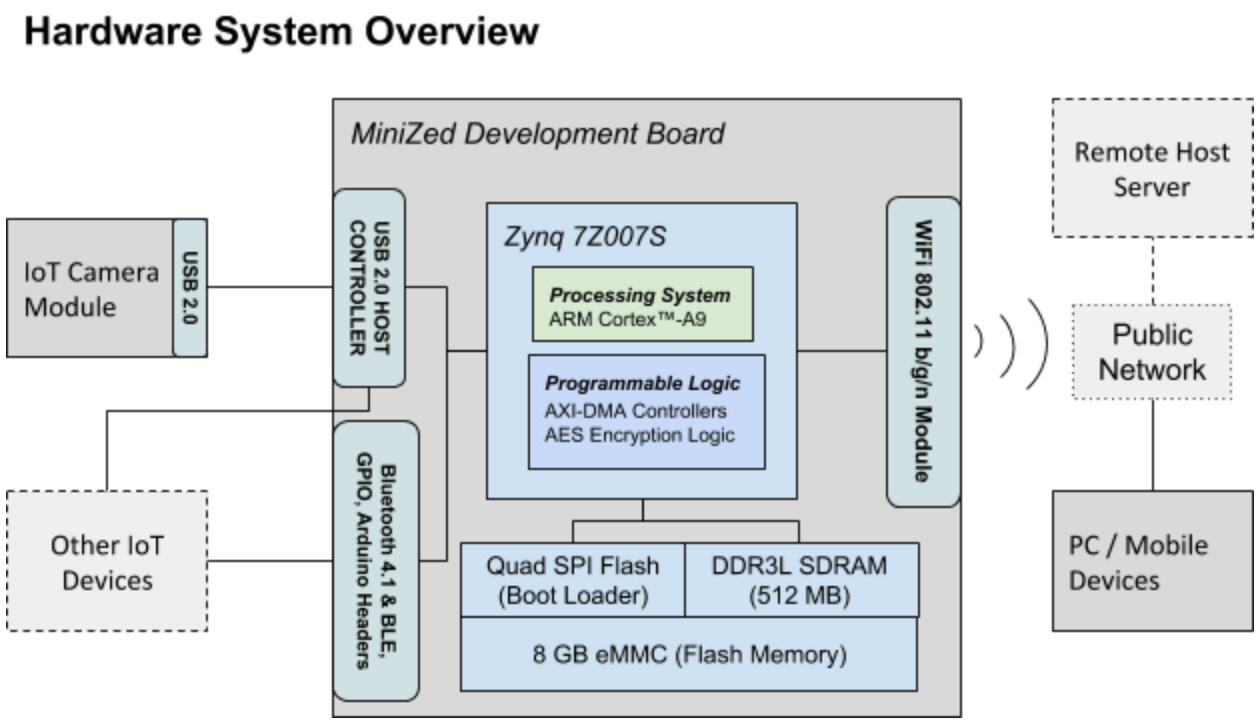

Figure 2: Hardware Overview of the System

Figure 2 illustrates the top-level hardware block diagram of the design. The IoT devices can be connected to the board with or without wire. The Zynq SoC has a processing system based on a single ARM Cortex A9 core. The crypto-accelerator is to be built on the programmable logic of the SoC. A DMA controller serves as the communication bridge of the crypto-accelerator and the embedded processor. The processor gets the serial data from the controller of physical ports or wireless communication modules, processes the data (decode, resize, filter, and encrypt), and sends the data to remote devices through the public network using the Wi-Fi module on the board.

Figure 3 illustrates the design from the software point of view. It clearly shows the relationship among drivers, kernel modules, libraries, and user space programs. All the modules marked in yellow are parts of the implementation of the system. The rest in green are libraries or drivers that come with the OS.

Software System Overview

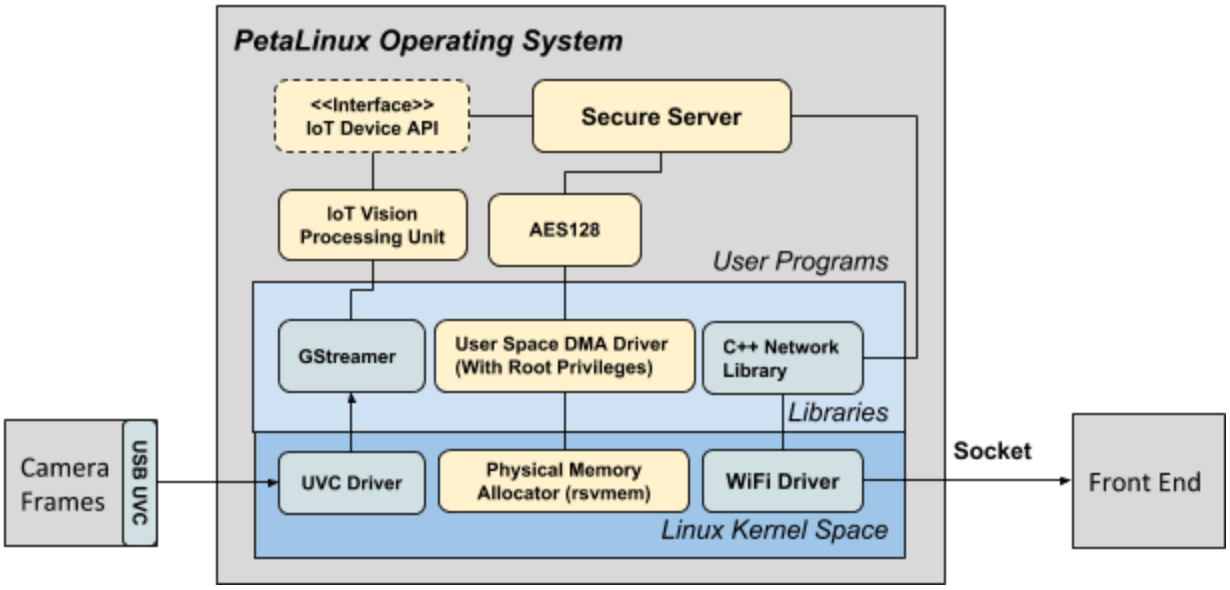

Figure 3: Software Overview of the System

The operating system controls all peripherals available on the board through driver programs. It communicates with the USB IoT camera module that is used as an example of an IoT node through the USB video device class (UVC) driver. The image frames are then processed and decoded with GStreamer library with additional plugins. The secure server program retrieves the data stream from IoT devices through the IoT Device API, encrypts the stream with a given key by using the AES128 functions, and then sends the ciphertext to the client through socket programming. The AES128 program, which must be run with root privilege, maps the plaintext into the DMA physical memory buffer, initiates the transfer to encrypt the data, write the ciphertext out to a file descriptor.

In order to work with IoT devices on different platforms, the system is implemented in four major and eight minor layers, as shown in Figure 4 below. The four layers are respectively the PL hardware logic, Linux kernel modules, libraries, user-space applications. The names of each module/program/entity are indicated inside the parentheses next to them.

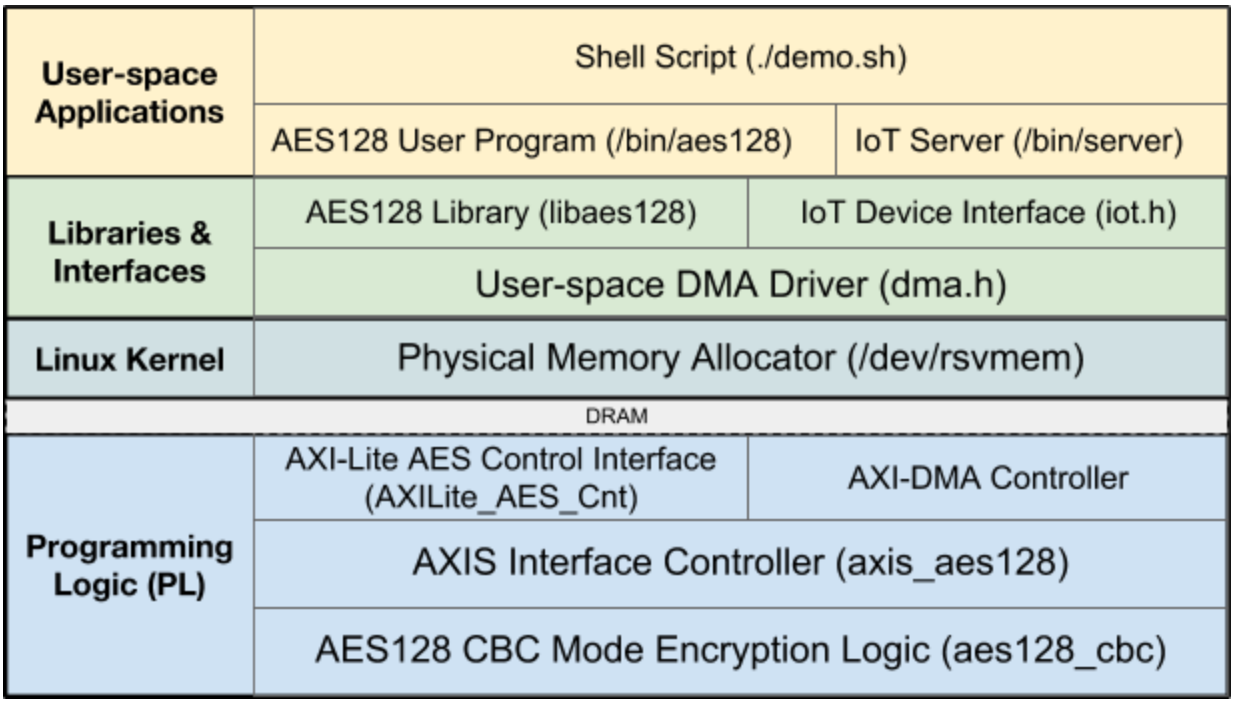

Figure 4: Hierarchical View of the Overall System

3.3.3 Network Communication Module

The network communication module is the layer of the project that implements the abstractions for the interface between the client and IoT device. Initially, the IoT device was either wirelessly connected to a local server or acted as the server itself. In this project the SoC module serves as the middleman as seen in Figure 5. With the server it provides additional computational power and features for the security process. The complete design of the secure network communication module is detailed in this section. The design is divided into two sub-modules, the communication protocol between the server and clients and encryption/decryption implementation.

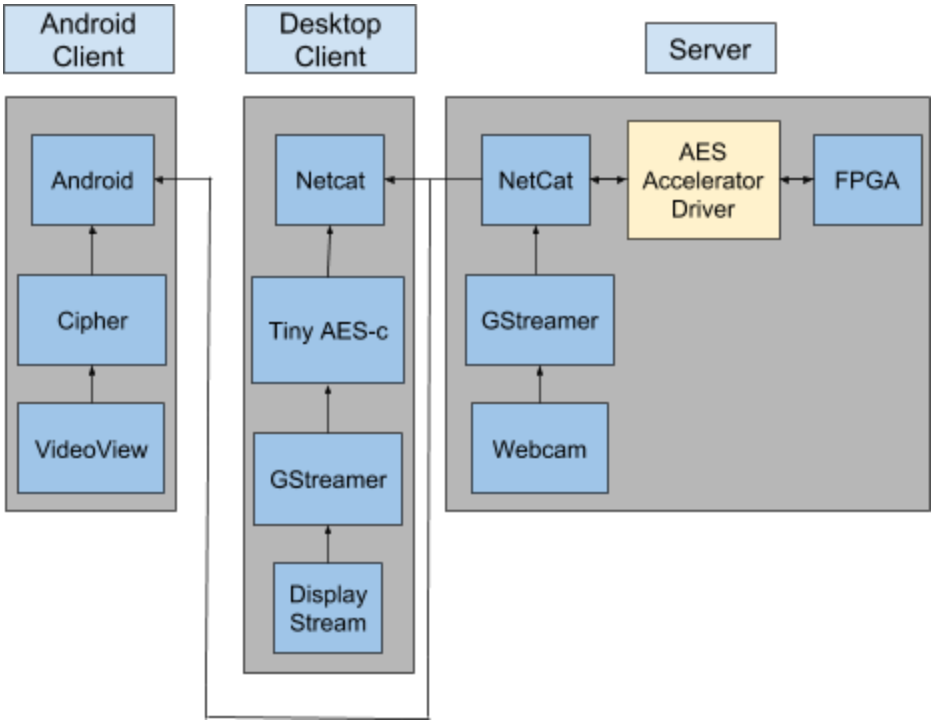

Figure 5: High Level Overview of Secure Network Communication Module

3.3.3.1 Communication Protocol

The original communication protocol involved implementing an HTTPS server to provide secure connections between the server and clients. However, this idea was swapped out for a socket to socket connection as shown in Figure 6. This is due to the fact that this project is focused on the hardware accelerator, not the implementation of an HTTPS server. Also, users of this system should be able to use any kind of network protocol and should not be limited to HTTPS.

The final implementation of the socket to socket connection is performed both by a server written in C++ code as well as the netcat library. The server written program uses the native C++ socket library and integrates the DMA driver header file and GStreamer library. The extent of this implementation allows for the MiniZed to output encrypted test data and images out to a client. The netcat implementation, on the other hand, not only has the same functionality as the C++ developed server but also handles streaming of data. With this the server is able to stream the encrypted webcam video rather than output a single frame at a time. The main difference is that the netcat implementation is performed by pipelining the standard outputs of the GStreamer webcam stream into the AES accelerator driver, which in turns pipelines its output into the netcat library to be sent to client devices.

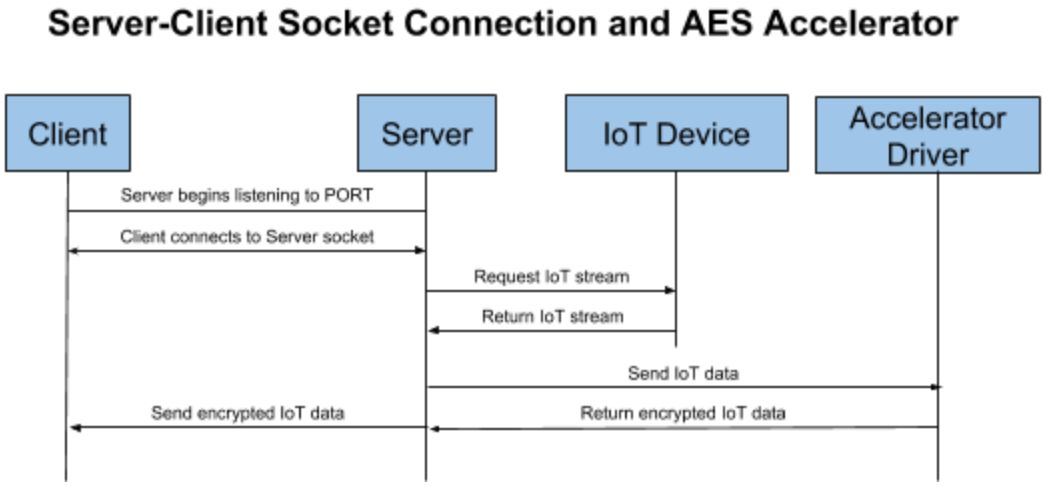

Figure 6: Server-Client HTTPS Handshake and AES Accelerator

3.3.3.3 Encryption/Decryption Implementation

The encryption and decryption implementation can be seen in the high level flowchart shown previously in Figure 5. Starting from the server, it begins by using the GStreamer library to grab the webcam stream and this serves as the input into the AES accelerator driver. For the netcat example, GStreamer is used to pipeline its input to “fdsink.” This is a GStreamer plugin used to sink data to a final destination, in this case it will write data to a unix file descriptor. The result of this is pipelined into the AES accelerator driver which will encrypt and output this to the netcat library. This process can be seen in Figure 7 shown below.

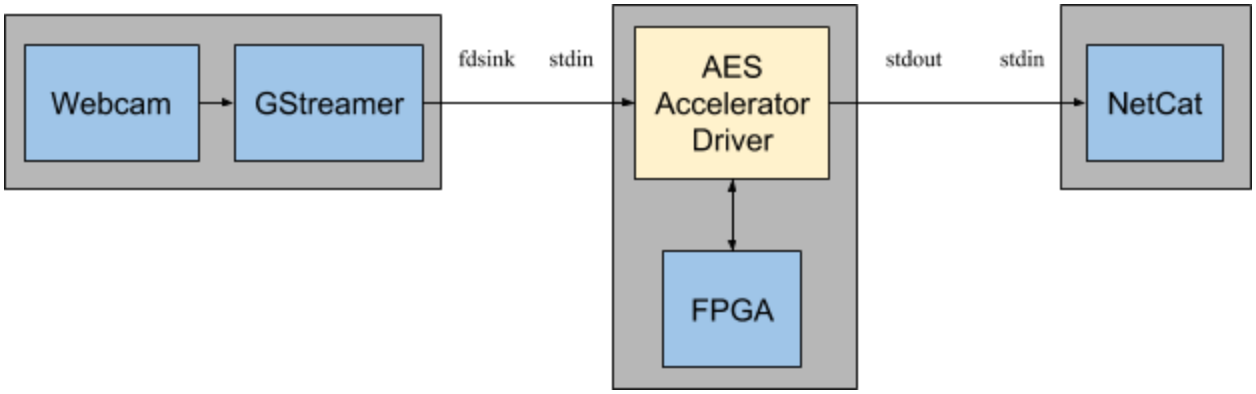

Figure 7: Netcat Server Pipeline

The client decryption for the desktop client in Figure 7 follows the same implementation as the server. Using netcat the stream is inputted and pipelined to the same modified tiny-AES-c [14] program used before to turn back into a readable stream that is displayed through gstreamer.

The Android client, on the other hand, uses the native java Cipher library to decode the encrypted stream and displays the result onto a VideoView object.

3.3.4 Cryptography Accelerator

The complete design and implementation of the hardware cryptography accelerator is detailed in this section. The section is divided into several sub-sections, including the communication between the processing system and the programmable logic (PS-PL Communication), the cryptographic algorithm (AES Encryption Algorithm), the AXIS interface (AES-DMA Communication), the kernel memory allocator, the DMA/AES driver program, and a complete shell program providing a simple abstraction. Figure 8 is the top-level block diagram of the entire PL design.

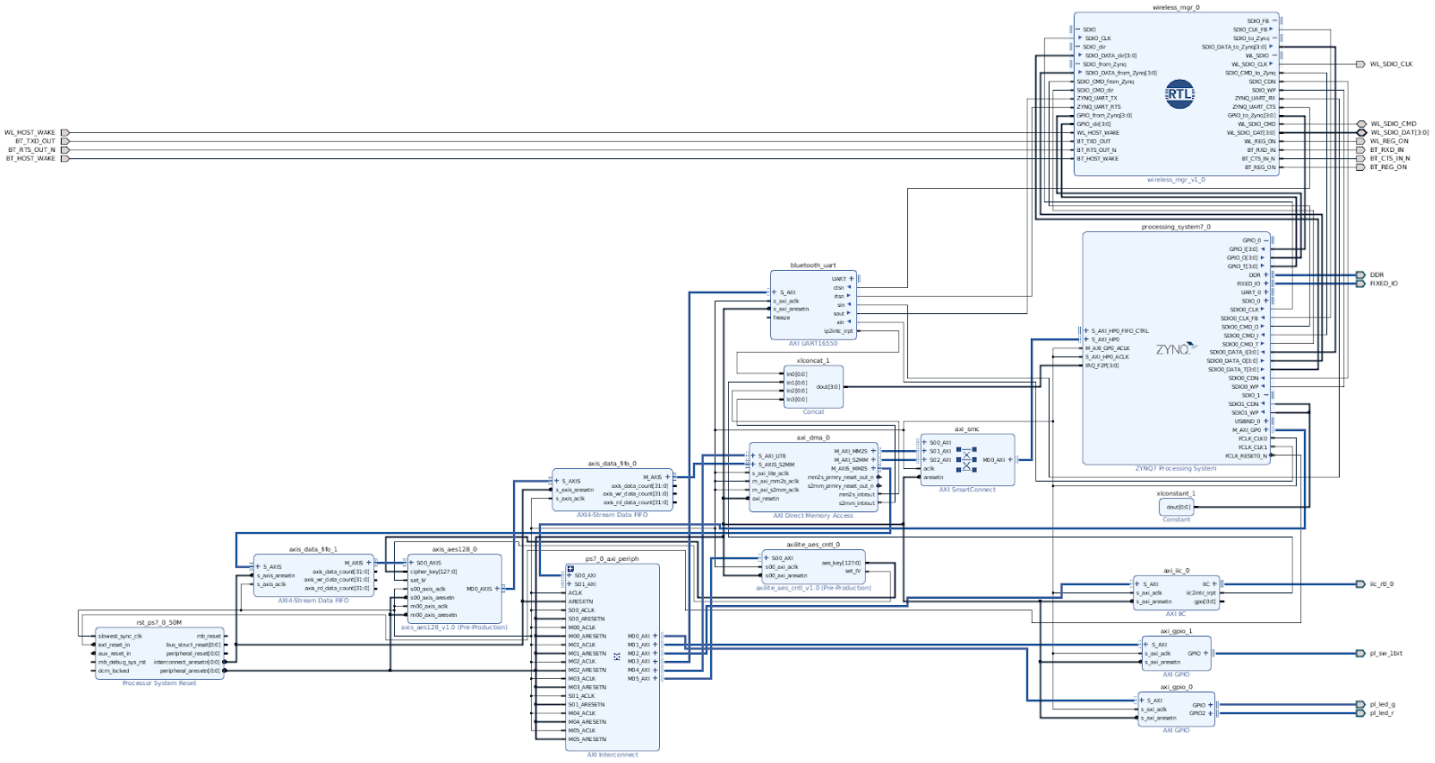

Figure 8: Top-level Block Diagram of the PL Design

3.3.4.1 PS-PL Communication

AXI DMA is widely used in many technology fields, such mainstream technique provides high-bandwidth direct memory access that suits our design the best. Since it's industry standard and very promising in the market, we chose this method for our ARM-core and FPGA handshake. Advanced eXtensible Interface (AXI) protocol established by ARM is often used for

data transfer and communications in SoC device. AXI protocols work as main communication between FPGA and ARM core in our Zynq-7000 AP SoC device, MiniZed board.

There are three types of AXI4 interfaces to be implemented in our system design. AXI4-Lite is for single transaction memory-mapped interface. AXI4 works the same way as AXI4 except it allows up to 256 data transfer per transaction. So far AXI4 and AXI4-Lite protocols are categorized as Memory-Mapped Protocols which involve target addresses for all data transaction. Both AXI4-Lite and AXI4 also include read/write address channels and read/write data channels to support data communication between masters and slaves.

AXI4-Stream is categorized as a stream protocol. Unlike AXI4 and AXI4-Lite, AXI4-Stream can enable high speed and unlimited data streaming while address is not used. Our project goal is to combine AXI4-Stream and Memory-Mapped Protocols to work together as communication between hardware blocks such as ARM-core processor, Memory and IP interface.

"The Xilinx® LogiCORE™ IP AXI Direct Memory Access (AXI DMA) core is a soft Xilinx IP core for use with the Xilinx Vivado® Design Suite. The AXI DMA provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals" [7]. Lastly, AXI-Lite allows the processor to communicate with the AXI DMA to initiate data transfers, detailed operation of our AXI DMA design will be explained below. There are few key components to introduce in the AXI DMA protocol. AXI DataMover works as a bridge between AXI4 memory-mapped and AXI4-Stream domains that enables high performance of data transfer. Memory Map-to-Stream (MM2S) and Stream-to-Memory Map (S2MM) are the two AXI DataMover in AXI DMA module.

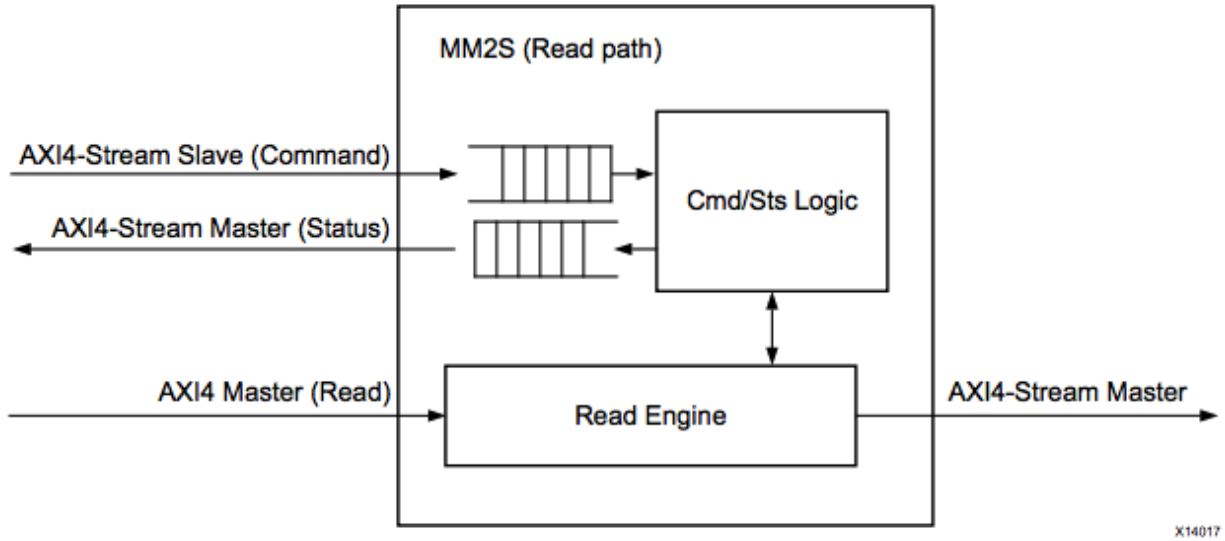

Figure 9: DataMover MM2S Read Path (diagram source: [8])

According to Figure 9, this is the block diagram of MM2S data mover which means memory map to stream. AXI4 protocol handles the data transfer between memory and DMA block and AXI4-Stream protocol handles the data transfer between DMA block and target IP source. AXI4 Read Master reads in the data from memory and transfer the data into Read Engine which is the memory-mapped side of DMA, then AXI-Stream Master is responsible to transfer the data obtained from the AXI4 Read Master interface to the corresponding receiver IP using the AXI4-Stream protocol.

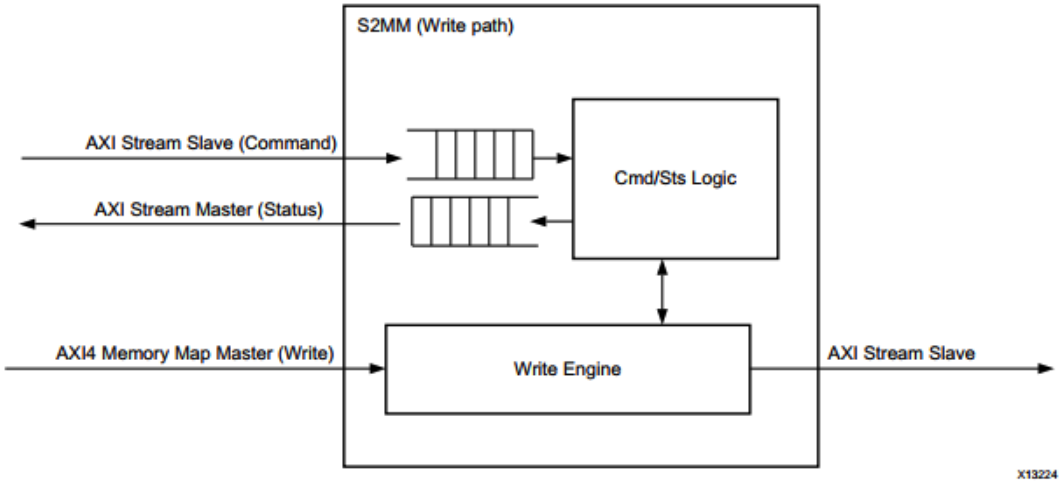

Figure 10: DataMover S2MM Write Path (diagram source: [8])

While MM2S issues a read request on the AXI memory-map interface, the S2MM issues a write request on the AXI memory-map interface as shown in Figure 10. This is the block diagram of S2MM data mover which means stream to memory-map. In our design, AXI Stream Slave is responsible to receive data from our source IP using AXI4-Stream protocol to the DMA Block, then transfer the received data from DMA block to the memory by AXI Write Master.

The block diagram in Figure 11 illustrates the hardware FPGA design that we will create for PS-PL interface. The design goal is to accomplish the mechanism for communicating data between ARM core and FPGA using combinations of AXI protocols in our Zynq-7000 AP SoC device, MiniZed board. The ARM-core processor is going to work as a commander using AXI-Lite protocol. As mentioned in the beginning of paragraph, the AXI-Lite provide single transaction at a time which is perfect for the processor to set up and initiate the data transfer.

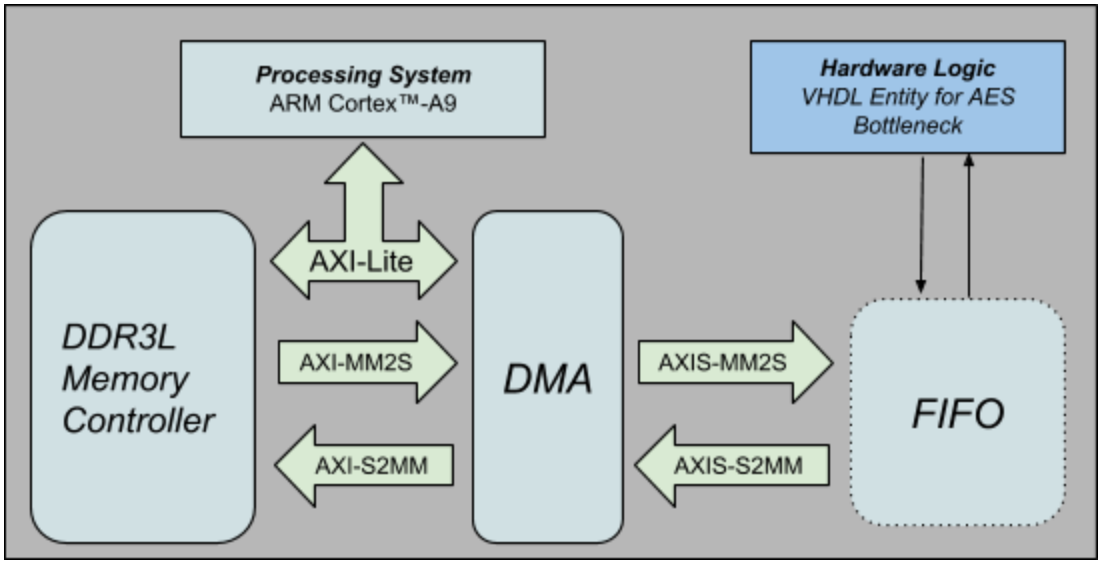

Figure 11: AXI-DMA Block Diagram (diagram source: [9])

After receiving the command from processor, DMA has to claim the data address from the DDR memory. AXI Datamover will work as bridge of data transfer for our hardware design between our memory and custom IP. We are using AXI-MM2S and AXI-S2MM as memory-mapped protocols to provide the DMA access to DDR memory. Memory-mapped protocols involve concept of transferring target address in all transactions therefore AXI buses allow DMA to obtain data from DDR memory and ready to stream out data to our hardware logic for AES encryption.

After the DMA controller obtained data from memory, we will be using AXIS-MM2S and AXIS-S2MM as streaming protocols to stream data without address into our custom IP block. We will be using FIFO as a working queue just like a waiting line. Let's say we stream 10,000 bytes data at a time which is the maximum that the data mover can handle, but only 32 bytes can be processed at a time, then the we will need a waiting queue for all these data to standby which is the main function of FIFO. Lastly, these data will go through our VHDL Entity for AES bottleneck in order to complete our encryption acceleration. After the work is done, the encrypted data will stream back to our DMA then back to memory using AXI memory-mapped protocol.

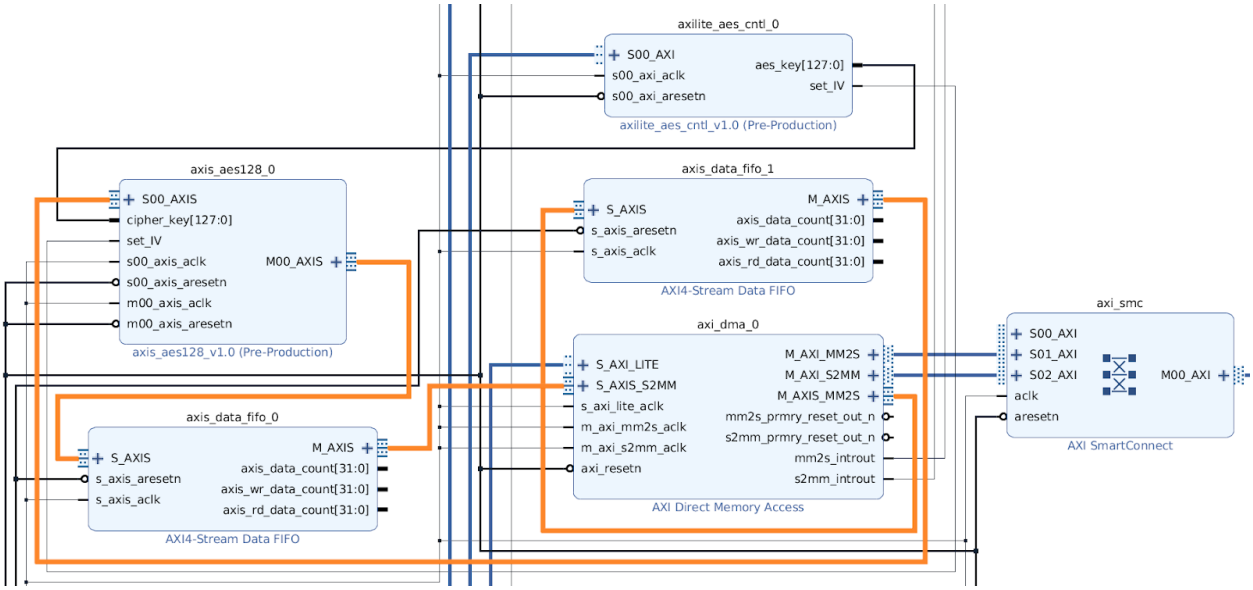

Figure 12: AES-DMA Communication Block Diagram

The block diagram illustrates the actual implementation of PS-PL communication. Our memory and the DMA are connected with AXI SmartConnect. One FIFO handles the data in and the other one handles the data out during encryption process. The orange line highlights the data path in our design. DMA obtained data from memory through AXI SmartConnect into the M_AXI_S2MM port then sends out the data to AXI4-Stream Data FIFO from S_AXIS_S2MM port. Then the data goes through our AES block for Encryption. The encrypted data then streams to our second FIFO then back to DMA through M_AXIS_MM2S port. Finally, the encrypted data is sent back to AXI SmartConnect through M_AXI_MMS2 port then back to memory.

3.3.4.2 AES Encryption Process and Logic

As discussed in Section 3.2.2.2, the two proposed approaches to accelerate the cipher have their own pros and cons. Considering the fact that our application performs much more encryption than decryption, the first approach that only implements MixColumn step on FPGA would not be feasible since the bottleneck is not obvious in encryption process. Therefore, we decide to take the second approach, realizing the entire AES algorithm in hardware. The MiniZed development board that we are using has enough programmable logic and on board memory for the hardware AES logic, which requires a 4 KB table for lookups. The table will be stored in the on-chip memory that has lower latency and higher bandwidth than the DRAM. To reduce the complexity of the logic design, only AES128 of CBC mode will be implemented. AES128 CBC is marked as a secure algorithm by TLS 1.2 standards and is widely supported. Although it is more complicated than EBC mode, it is proved to be much more secure for image encryption [10]. Hence, we believe that it is a reasonable choice of cryptography algorithm.

The design overview is illustrated in Figure 13. In the design, we will use the control unit to start the AES encryption process by sending out the control signal to initiate the Encryption Datapath and Key generation. 4-word (128 bit) round key will be generated as a multiplication with the previous round key. After the keys are generated, they will be sent to Encryption Datapath which process through SubBytes, ShiftRows, MixColumns and AddRoundKey except initial and final round. Since our key size is 128-bit key, the cycle will repeat 10 rounds that convert the plaintext (input) into ciphertext (output).

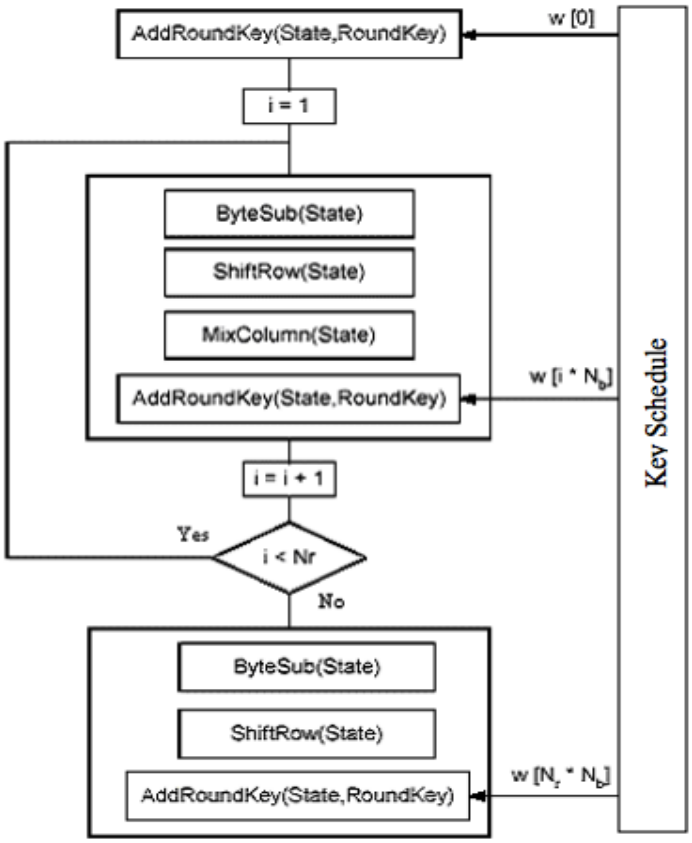

Figure 13: AES Encryption Process Flowchart (diagram source: [4])

High-level Description of AES Cryptography

Step 1: Initial Round

Initial Roundkey Transformation:

Plaintext ⊕ Cipher key

50	6e	74	41	⊕	54	20	74	6b	=	04	4e	00	2a
6c	74	20	45		68	69	68	65		04	1d	48	20
61	65	34	53		69	73	65	79		18	16	51	2a
69	78	20	2e		73	20	20	2e		1a	50	00	00

Figure 14: Visualization of Initial Roundkey Transformation

The table is an example of how the Initial roundkey is created by using XOR operations. The first box represents 16 bytes of our plaintext input, written as hex numbers; then the second box also represents 16 bytes of our cipher key inpet, written as hex numbers. Since we are in CBC mode, the result then XOR with Initialization Vector (IV). We would like to set it as 16 bytes of [0,0], so the result stays the same after XOR. Each byte of the Plaintext box is XORed with the corresponding byte in the Cipher text box. Round 1 to round 9 keys are derived from four steps of transformation repeatedly which are Subbytes, Shiftrows, Mixcolumns and Addroundkey after the Initial roundkey is created.

Step 2: Rounds (Round 1 to Round 9):

a) *Subbytes*
b) *Shiftrows*
c) *MixColumns*
d) *AddRoundKey [Key Expansion]*

a) Subbytes:

This is a nonlinear byte substitution. In this step, each byte in the state is replaced independently according to a fixed 8 bit lookup table, the Rijndael S-Box (substitution-box).

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 2 : Rijndael S-box (diagram source: [11])

The AES cryptography in our design is 128-bit key which is 16 bytes in a 4x4 square box. The S-box lookup table is represented in Hexadecimal numbers. For example, if the 65 is the hex number in our AES key box. The first digit refers to the row of the Rijndael S-box and the second digit refers to the column. Then 65 will be substituted to 4D after S-box function according to the table 2.

b) Shiftrows:

In this operation, each row of the state is cyclically shifted to the left, depending on the row index. The 1st row doesn't shift. The 2nd row is shifted 1 position to the left. The 3rd row is shifted 2 positions to the left and the 4th row is shifted 3 positions to the left. And each byte that are shifted will be combined again from the right side to form a 4x4 square box again as the diagram shown below.

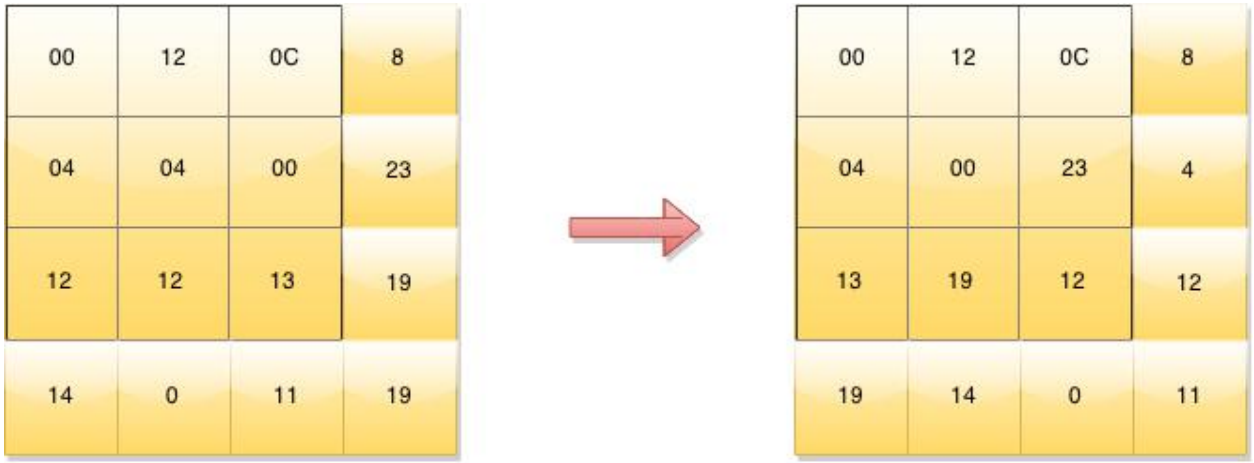

Figure 15: Visualization of Shiftrows operation [11]

c) MixColumns:

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes. This application can be understood as a column-by-column multiplication with a fixed matrix according to the figure 16. The mathematical way of performing calculations using table L and table E will present below.

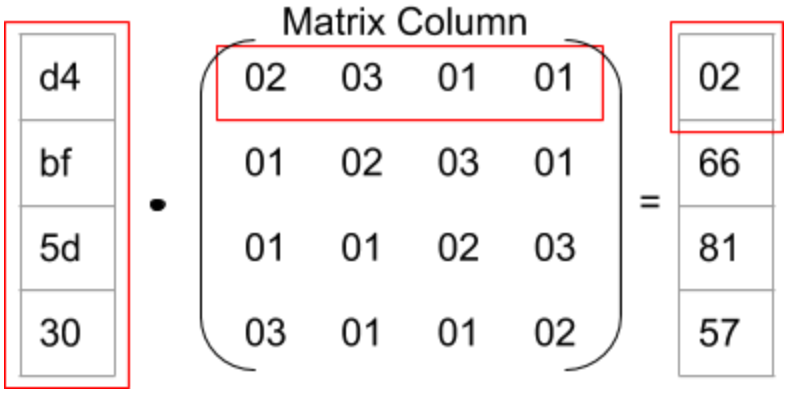

Figure 16: Visualization of Mix-Column calculation

E Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

L Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	8E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	C4	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	6C	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	BC	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2E	89	B4	7C	B8	26	77	99	E3	A5
F	67	4A	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

Table 3: AES mixcolumn Table E & L look up (diagram source: [12])

Mathematical Calculation

[d4,bf,5d,30] becomes [02,66,81,57].

First byte:

$02 \bullet D4 \oplus 03 \bullet BF \oplus 01 \bullet 5D \oplus 01 \bullet 30$

$= E(L(02) + L(D4)) \oplus E(L(03) + L(BF)) \oplus E(L(01) + L(5D)) \oplus E(L(01) + L(30))$

$= E(19 + 41) \oplus E(01 + 9D) \oplus E(00 + 88) \oplus E(00 + 65)$
$= E(60) \oplus E(9E) \oplus E(88) \oplus E(65)$
$= B5 \oplus DA \oplus 5D \oplus 30$
$= 02$

Second byte:
$01 \bullet D4 \oplus 02 \bullet BF \oplus 03 \bullet 5D \oplus 01 \bullet 30$
$= E(L(01) + L(D4)) \oplus E(L(02) + L(BF)) \oplus E(L(03) + L(5d)) \oplus E(L(01) + L(30))$
$= E(00 + 41) \oplus E(19 + 9D) \oplus E(01 + 88) \oplus E(00 + 65)$
$= E(41) \oplus E(B6) \oplus E(89) \oplus E(65)$
$= D4 \oplus 65 \oplus E7 \oplus 30$
$= 66$

Third byte:
$01 \bullet D4 \oplus 01 \bullet BF \oplus 02 \bullet 5D \oplus 03 \bullet 30$
$= E(L(01) + L(D4)) \oplus E(L(01) + L(BF)) \oplus E(L(02) + L(5D)) \oplus E(L(03) + L(30))$
$= E(00 + 41) \oplus E(00 + 9D) \oplus E(19 + 88) \oplus E(01 + 65)$
$= E(41) \oplus E(9D) \oplus E(A1) \oplus E(66)$
$= D4 \oplus BF \oplus BA \oplus 50$
$= 81$

Forth byte:
$03 \bullet d4 \oplus 01 \bullet bf \oplus 01 \bullet 5d \oplus 02 \bullet 30$
$= E(L(03) + L(D4)) \oplus E(L(01) + L(BF)) \oplus E(L(01) + L(5D)) \oplus E(L(02) + L(30))$
$= E(01 + 41) \oplus E(00 + 9d) \oplus E(00 + 88) \oplus E(19 + 65)$
$= E(42) \oplus E(9D) \oplus E(88) \oplus E(84)$
$= 67 \oplus BF \oplus 5D \oplus D2$
$= 57$

d) Key Expansion:
AES-128 bit key is made up with 10 rounds and each round key is transformed from the previous roundkey using a key expansion function. Expanding a cipher key to the next round key works in a column-by-column approach. Since AES-128 is 16 bytes in a 4x4 square box, each column contains 4 bytes. Therefore, four transformations have to be perform to generate a new roundkey. The completion of key expansion acquire three steps which are RodWord(RW), SubWord(SW) and Round Constant(Rcon) operations. These operations are defined as follows:

RotWord: Takes the last column of the cipher key and performs the rotation of shifting top byte to the bottom of the column. For example: [a1,a2,a3,a4] becomes [a2,a3,a4,a1].

SubWord: After RodWord operation, all four bytes of the column now get substituted with S-box. S-box table look up refers to table 2.

Round Constant: After SubWord operation, the entire column then operate XOR with the corresponding round constant column according to the table 4.

Initial	R2	R3	R4	R5	R6	R7	R8	R9	Final
01	02	03	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
00	00	00	00	00	00	00	00	00	00

Table 4: Round Constant (Rcon[i])

After XOR with round contant, outcome of the column then XOR again with the first column of the cipher key. The rest of the columns don't have to go through these steps, just simply XOR the previous Column in the next roundkey with the corresponding column in cipher key to get the new column. The virtualization of the 1st column computation is presented aside.

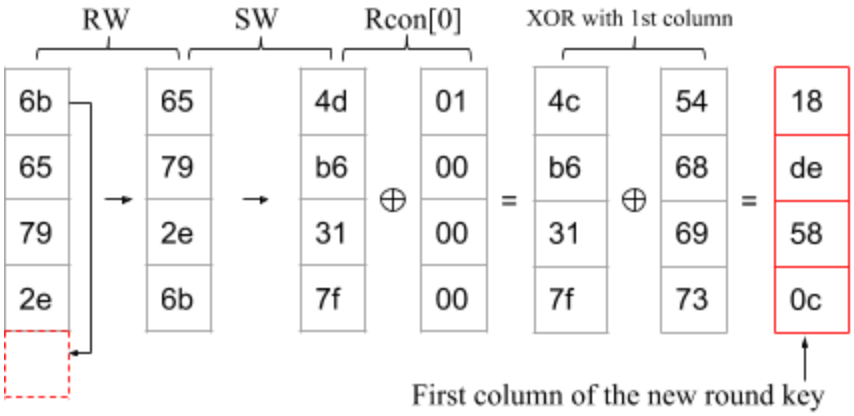

Figure 17: Visualization of Key Expansion Process

Step 3: Final round:

a) *Subbytes*
b) *Shiftrows*
c) *AddRoundKey [Key Expansion]*

In the final round step, all procedures are the same as Step 2 except there is no MixColumns. The last MixColumns of encryption has no corresponding InvMixColumns for decryption, so we simply omit the MixColumns step in final round.

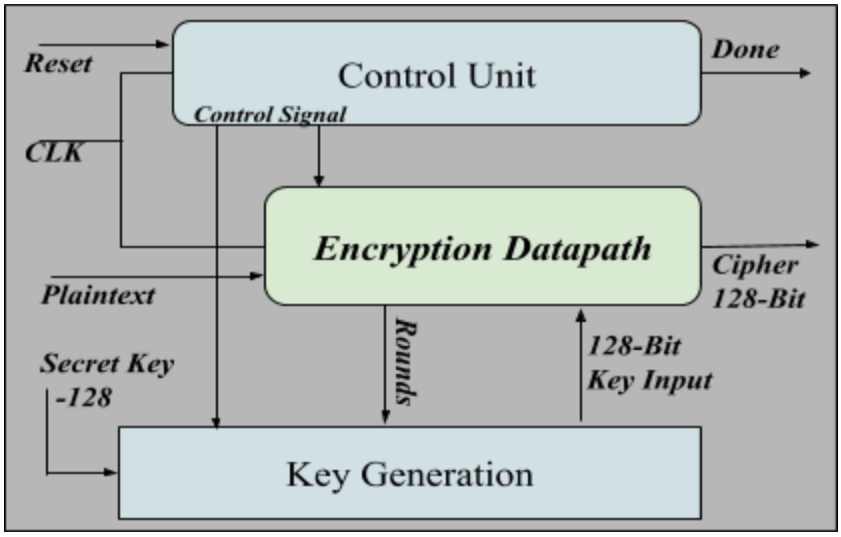

Figure 18: Block Diagram of AES Hardware Design

3.3.4.3 Top-level AES Module Implementation

The top-level of the AES module is implemented to control the dataflow during the encryption process. The module input and output ports are illustrated in Figure 19. The hardware description of the encryption substeps are based on an open source project available on Github named aes128-hdl [13]. The controller of this module is implemented as a simple Finite State Machine (FSM) consisting of 4 major states as shown in Figure 20 below. As discussed in the previous section, the AES128 encryption process has 10 rounds of repetition with a special last round. The FSM starts the encryption as soon as the "START" signal is asserted. It then perform an exclusive-or operation on the plaintext, the key, and the initialization vector (IV) and uses result as the input to the first round. The "BUSY" signal is asserted throughout the entire process of encryption. The output of each round is stored in a register which is connected back to the input of the next round except for the last round. The original key expansion unit from the aes128-hdl package generates and stores ten round keys at the same time in order to increase the encryption throughput with a pipeline architecture. Since our implementation in CBC mode cannot be realized with pipeline architecture, the ten round key expansion unit would become a waste of space and energy. Therefore, a different key generation unit is implemented to generate each round key one by one. The key generation entity takes the previous round key and one of the R constants which is explained in Section 3.3.4.2. It computes the next round key in a combinational approach.

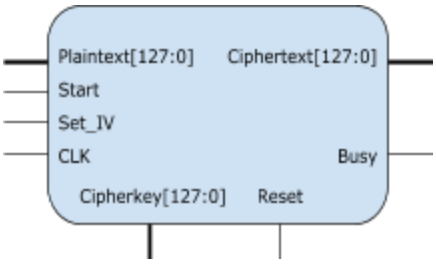

Figure 19: Top-level AES Module Ports

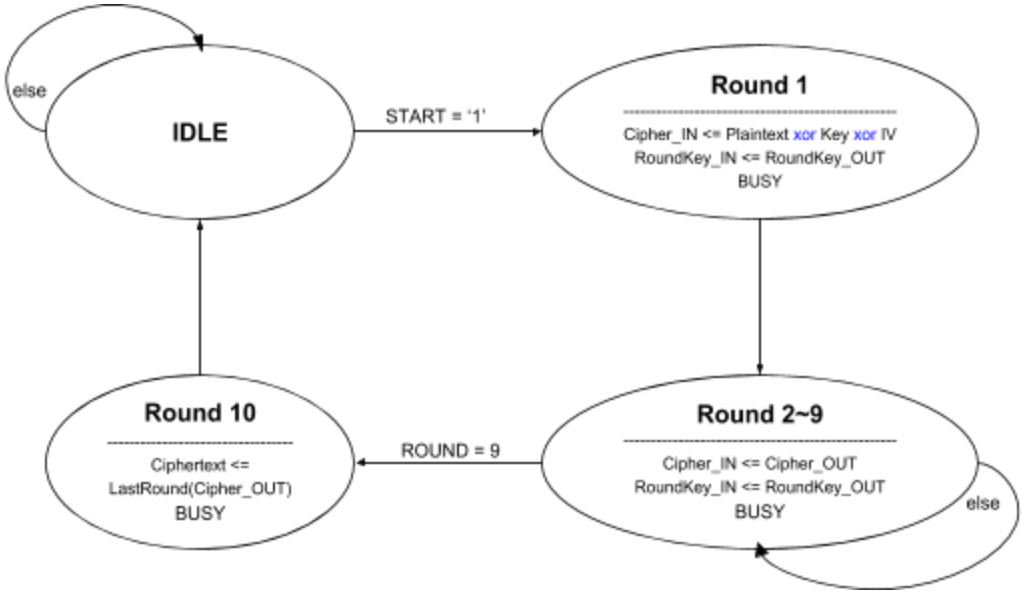

Figure 20: Top-level AES Module State Diagram

3.3.4.4 Advanced eXtensible Interface Stream (AXIS) Protocol

Since the Xilinx AXI-DMA IP Core has an AXIS interface at the PL side, the AES module must be able to comply with the interface in order to communicate with the DMA core. AXI4-Stream is a master-slave interface typically used by on-chip data flow communications. Each AXIS is a unidirectional channel which consists of hand-shaking controls signals and a data bus of some user-specified width. The master side of the channel outputs the data to the data bus, sets the "T_Valid" signal, and wait on the "T_Ready" signal from the slave. There is also a "T_Last" signal in the channel that should be asserted when it is the end of the stream. The detailed description of the interface can be found in the Xilinx AXI Documentation which is available online. Figure 21 is the port view of the axis_aes128 implementation.

Two AXIS interfaces are required in order to connect the top-level of the AES module and the DMA core. They are respectively a slave interface that can receive plaintext from the data FIFO connecting to DMA module and a master interface that sends out the ciphertext back to the DMA core. A simple FSM is implemented as the controller for both AXIS interfaces and serves as a wrapper for the AES module. Figure 22 shows the state diagram of the implemented FSM. The prefix "S_" indicates that the signal belongs to the slave interface (which receives the data). The prefix "M_" on the other hand means the signal is connected to the master interface. Note that the data width of the interfaces are configured to 128 bits, which is the same as the block size of the AES encryptor. Therefore, the data bus of both master and slave interfaces can be directly mapped to the AES plaintext input and ciphertext output respectively.

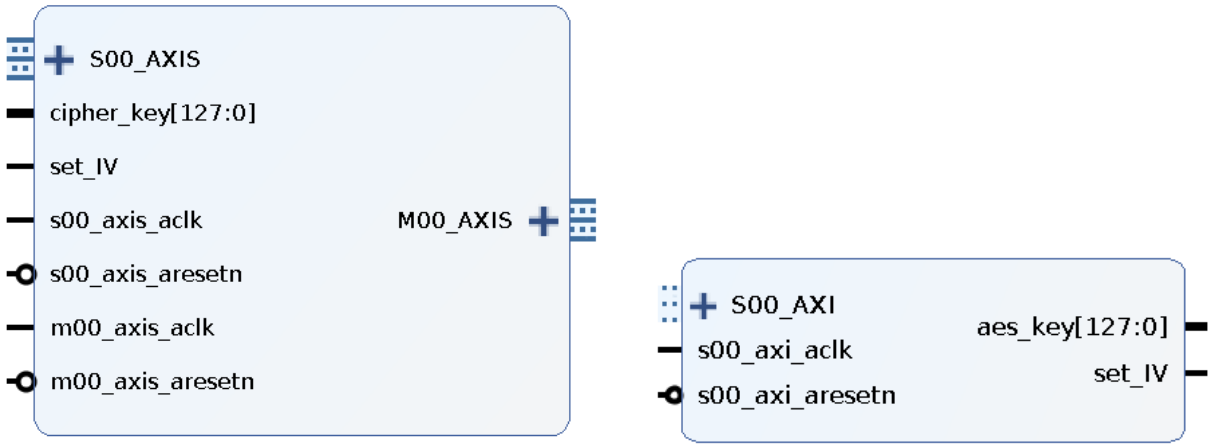

Figure 21: Port Interface of AXIS_AES128 and AXILite_AES_CNTL

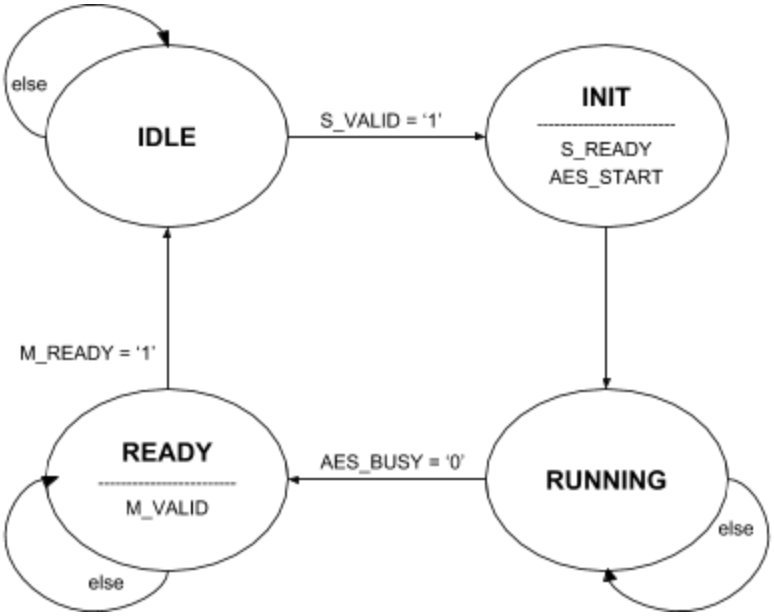

Figure 22: State Diagram of the AXIS Interface Controller

3.3.4.5 Kernel Space Memory Allocator

The DMA APIs and drivers in Linux are only available in kernel space because the structure of a Linux-based operating system does not allow any user space process to see the physical memory address, not to mention allocating a physical memory buffer. However, the DMA core must know the physical address of the source and destination in order to start a transfer. Therefore, a custom kernel module needs to be developed in order to allocate physical memory buffer for the DMA.

The implemented kernel module is the Reserved Memory (rsvmem) virtual device. The module implements the character device interface of the linux kernel, which allows the user

space program to communicate with the module with basic file operations such as open(), read(), write(), and close(). The module allocates a fixed size buffer using kmalloc() kernel function call with __GFP_DMA flag when it is inserted into the kernel, e.g. at the time of initialization. The kernel memory allocation with the specific flag guarantees a DMA suitable consecutive memory buffer when the allocation is done successfully. The module then translates the kernel virtual address of the allocated buffer into physical memory using virt_to_phys() function from asm/io.h address and remembers the address.

When a user process opens and reads the character device (/dev/rsvmem) of this module, it returns exactly 4 bytes of data indicating the physical address of the buffer. The user process can then map the given region of physical buffer into its virtual memory through mmap(2) system call on /dev/mem. Note that such operation requires root privileges to perform.

3.3.4.6 User Space DMA/AES Driver

A user space driver for the DMA/AES peripherals on the PL is developed. In fact, it is more like a user-space application programmable interface (API) for the DMA transfer rather than a actual driver, which is normally developed in the kernel space of Linux. The main reason why the DMA/AES API is implemented in the user space is the fact that doing it in the kernel space complicates the program a lot and might also require an extra copy of the data since data must be transferred to the kernel space memory from the user space memory.

The basic operations of the user-space DMA driver API are discussed here. The API heavily depends on the reserved memory module (/dev/rsvmem) and the physical memory (/dev/mem) to work. The physically memory address of the AXI-lite registers for DMA controls and AES controls must be set accordingly in the header file before the API is used. The API requires the user program to call dma_init() function before any other functions are called. The initialization function map the AXI-lite registers into the current virtual memory space and retrieves the physical memory buffer address from reserved memory module. The physical address is then used to map the buffer into the current virtual memory space and a global pointer to the buffer would be set accordingly. Note that the physical address is hidden in the API as a static variable to avoid user program's misuse of the physical address.

After the source buffer is filled up with the data to be transferred, the user program will call the dma_start() function which takes an unsigned integer specifying the length of the data to be transferred. The dma_start() function sets the corresponding AXI-lite control registers of the DMA to start the transfer and then returns immediately. The dma_quick_poll() and dma_sync() functions can then be used to check the completion of the transfer. The former returns the status of the DMA immediately, whereas the latter suspend the calling thread until the transfer is completed.

When the user process is done with all the DMA transfers, the dma_clean_up() must be called to release the resource held by the API. The functions unmap and unmount all the devices used by the API and force the driver to go back to the uninitialized state. Several AES-specific functions available in the API will be discussed in the next section.

3.3.4.7 AES128 User Program

A complete shell program for AES128 encryption in CBC mode is developed for demonstration and testing of the system. The program takes a list of arguments from the shell and performs the cryptography process based on the given parameters. It is designed to be very flexible in order to meet the needs of different IoT devices.

The basic operations of the program highly depend on the DMA/AES API discussed in the previous section. The user program reads the data from a source file descriptor, encrypts the data with a user-specified chunk size, and writes the ciphertext to an output file descriptor. The encryption key is read from a given file path containing the key in hexadecimal string format. The program also sets a user-specified polling interval for the DMA synchronization since interrupts are not used in this system. This is done to avoid busy waiting of the process which could waste lots of computing cycles. A table of the encryption time of different chunk size is presented in Table 6 in the Section 4.1 Result Analysis. It is suggested that the interval should be carefully set according to the estimated encryption time in order to maximize the processor utilization.

A pure software-based AES128 encryption/decryption function is also included in the program and is available as an option for results comparison purpose. The software encryption is based on a modified version of tiny-AES-c [14], which is an open source unlicensed project aiming to create small and portable software for AES encryption. Some additional options to the program are also supported, including a timing mode that shows the real time and the CPU time used by the encryption process.

3.3.5 IoT Device Interface

A software interface for IoT devices is discussed in detailed in this section. Considering the flexibility and extensibility of the system, an abstraction layer of the different types of IoT nodes is developed to generalize the communication between the devices and our system. An example program, the IoT Vision Processing Unit, is also designed and implements the interface to demonstrate how the system works with this interface.

3.3.5.1 Detail Description of the Interface

The interface defines the way the data can be retrieved from all IoT devices connecting to the system through a variety of ports and protocols. All programs that implement this interface are required to provide an ID and a name of the device, as well as a method that returns a specific length of the data stream coming out of the associated IoT device. More specifically, the method should retrieve a specific size of data from the device, map the data to a given address of the memory, and return a pointer to the memory location. The interface also defines an optional method that can send data to the IoT device for control purpose.

The interface is defined in a way that abstracts the drivers of different physical ports of the system. To put it differently, it creates an abstraction layer for different data format and communication protocols between the system and the IoT devices. The server module which uses the interface does not need to know the actual type of connection between the system and the IoT device. This approach not only makes it more extensible but also increases the security of the system by limiting the server program's control of the IoT devices.

Figure 23 below shows how the interface acts as a middleman between the server module and the IoT devices. The IoT devices can be connected to the system using different protocols and drivers. In addition, different types of IoT device can have their own data processing programs that implement the IoT Device Interface by providing the three key methods, identify(), read(), and send().

The identify method checks the connection of the device and returns the ID and the name of the device. The method also returns an 8-bit status code indicating the status of the device. The purpose of this method is to identify a device and make sure that the device is ready to be read. The read method, which takes address, length, offset, and mode as parameters, maps the data from the device to the given location and returns a boolean indicating the success of the operation. The send method, which is an optional method of this interface, sends data, mainly for control purpose, to the IoT device. The caller of this function passes the address and length of the data along with a control code to the function. The method also returns a boolean value indicating the success of the operation.

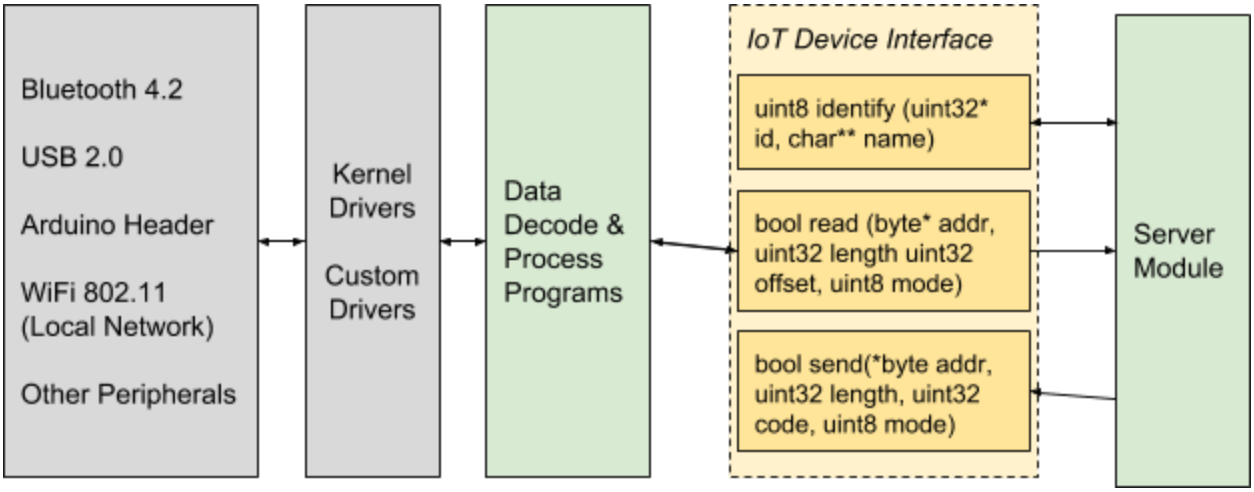

Figure 23: IoT Device Interface

Note that even though the IoT device interface has been proposed and developed in the design, it it not a requirement. In fact, the IoT camera streaming demonstration presented in this paper does not use this interface. Instead, it invokes the AES128 user program and netcat tools in a Bourne Shell script. This change is done because we realized that the interface and the system data flow should not be limited. The most flexible way to do this would be to create a set of tools/libraries/APIs available in different layer of the system. Therefore, the IoT device interface here is only a suggestion of the system data flow.

3.3.5.2 IoT Vision Processing Unit

The IoT vision processing unit uses the GStreamer library. GStreamer is a collection of plugins that can be used for inputting, processing, editing and outputting video and audio files. Shown in Figure 24 is the GStreamer pipeline used to access the webcam, convert it into mjpeg format and output it into the AES accelerator hardware.

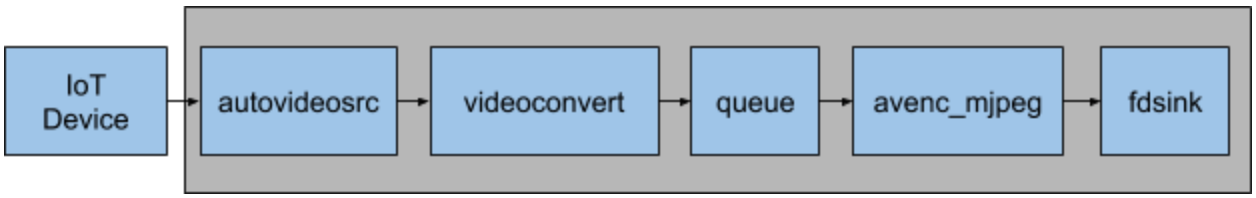

Figure 24: GStreamer Processing Pipeline

Starting from the beginning of the pipeline, “autovideosrc” is used to scan for video or audio sources. This scan isn’t interfered with by any other inputs because the only inputting media device is the IoT webcam, but the device can be specified. After the source is found “videoconvert” converts the input video source format such that it can be placed in a queue. The “queue” is used in the pipeline because it is necessary to have both the server and client buffer be equal else it can result in a bufferpool error. The bufferpool is necessary to manage a list of

buffers of the same type and length. Once the queue is set then the video input is encoded to MJPEG using “avenc_mjpeg.” GStreamer also provides a large number of different formats such as h264. Once the video input is inputted, queued and formatted, “fdsink” or file descriptor sink is called to output the result as a standard input for the AES accelerator hardware.

3.4 Testing

This section will go over how the implemented section is used to accomplish the goals set and confirm the results of the hardware cryptographic accelerator on client devices.

3.4.1 Functional Simulation

Separate VHDL testbenches are developed for the AES_CBC encryptor as well as the top-level AXIS_AES128 module for functional (pre-implementation) verification of the design. The testbenches are non-exhaustive. Several combinations of plaintext and keys as well as IVs are tested and the results are compared with the output from the tiny-AES-c [14] software package with the exact same set of input. Figure 25 is a simulation waveform for the top-level axis_aes128 design. The testbench feeds input data and control signals into the unit under test (UUT) by following the AXI stream protocol and expects the outputs also comply with the interface.

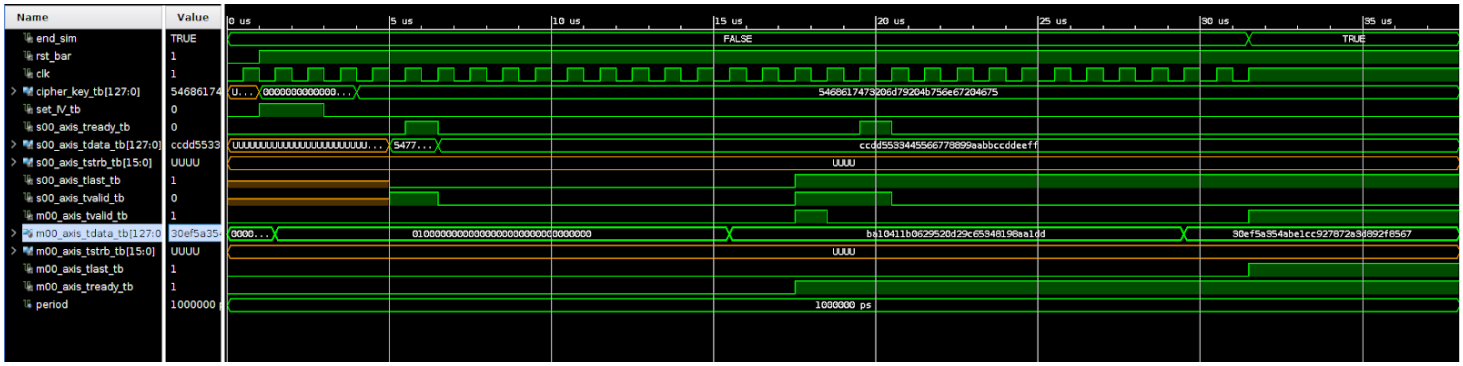

Figure 25: Functional Simulation Waveform of AXIS_AES128

The reset_bar signal is first asserted for one cycle to initialize the FSMs in the UUT. Following the reset, the set_IV control signal is set and the UUT is expected to load the IV from the 128 bits input cipher_key[127:0]. Note that the cipher_key input is shared by the IV registers and the internal cipher key registers. The set_IV signal selects the input to them. In fact, the IV only takes one full cycle to set, but in this testbench two cycles are assumed since the driver code that has been developed on the PS sets this control signal for several cycles. After the IV is properly set, the input plaintext is placed to the s00_axis_tdata bus and the s00_tready signal is set to signal the UUT that a valid data is available on the bus. The cipher_key input is also set to the desired encryption key at the same time. As shown in the waveform, the ciphertext becomes

available on m00_axis_tdata bus exactly 10 cycles after the s00_tready. Note that the current ciphertext is also the IV for the next round. After the testbench issues the m00_axis_tready signal, the UUT goes back to the idle state, and the next block of plaintext is then available on the bus. The key and IV do not need to be reset this time because the assumption is the two blocks are parts of one DMA transfer. Therefore, only 10 cycles are required for the following blocks of encryption.

Table 5 below shows the corresponding input and output of this waveform. Note that the presentation of bytes is LSB first for the simulation (e.g. plaintext[7:0] will be the first byte). The software byte ordering is higher address first (e.g. buf[15] comes before buf[14]).

IV	00000000000000000000000000000001	
Cipher Key	5468617473206D79204B756E67204675	
	Plaintext	Ciphertext
Simulation	6F775420656E694E20656E4F206F7754 FFEEDDCCBBAA9988776655443355DDCC	DDA18A194853C6290D5229061B4110BA 67852F893D2A8727C91CBE4A355AEF30
tiny-AES-c	6F775420656E694E20656E4F206F7754 FFEEDDCCBBAA9988776655443355DDCC	DDA18A194853C6290D5229061B4110BA 67852F893D2A8727C91CBE4A355AEF30

Table 5: Comparison of the Results from Simulation and tiny-AES-c

3.4.2 System Verification

Since the block design of the system is mostly composed of reliable IP cores from Xilinx and the axis_aes128 entity has been functionally verified as discussed in the previous section, a block design simulation is not urgently required and is not very feasible for a project of this scale. The axilite_aes_cntl block is based on the example AXI-lite design from Vivado with minimum modification, so it is very unlikely to have an error in this block. Therefore, the next step of system testing is a simple AXI-DMA program running on the baremetal mode because the baremetal mode has less indefinite factors than the non-real time operating system.

The baremetal program is obtained from modifying the Xilinx AXI-DMA Simple Polling example which comes with the Xilinx SDK. The bitstream of the hardware platform is programmed onto the FPGA prior to the execution of the program. The program starts ten DMA transfers of 32 bytes without resetting the IV for each round. Therefore, it is equivalent to encrypting 320 bytes stream in chunk of 32 bytes. Again, the result is compared with the output of the software package and they match exactly.

After verifying the hardware platform with the baremetal program, the bitstream is then exported to the PetaLinux project and is packed with the first stage boot loader (FSBL) binary. The kernel modules, software packages, and user programs for the AES crypto accelerator are all

included in the U-Boot image file, which is stored in root of the on board eMMC. The MiniZed board is then configured to boot from the Quad SPI Flash that contains the binary with the FSBL and the bitstream. U-Boot programs the FPGA with the bitstream and then unpack and load the U-Boot image in the eMMC to start the OS. The kernel module is inserted into the OS by issuing the insmod and mknod commands on the shell. The hardware encryption program is first tested with some hardcoded values in a test program. Figure 26 demonstrates one sample run of the test program. The output of the hardware AES encryption is checked with that of the tiny-AES-c [14] software package to ensure the correctness of the encryption. Following the test of hardcoded data, the complete AES128 program is tested with a short plaintext file. The encrypted file is sent to another computer through the netcat tool. The client side decrypts the file with the previously negotiated key and verify the result.

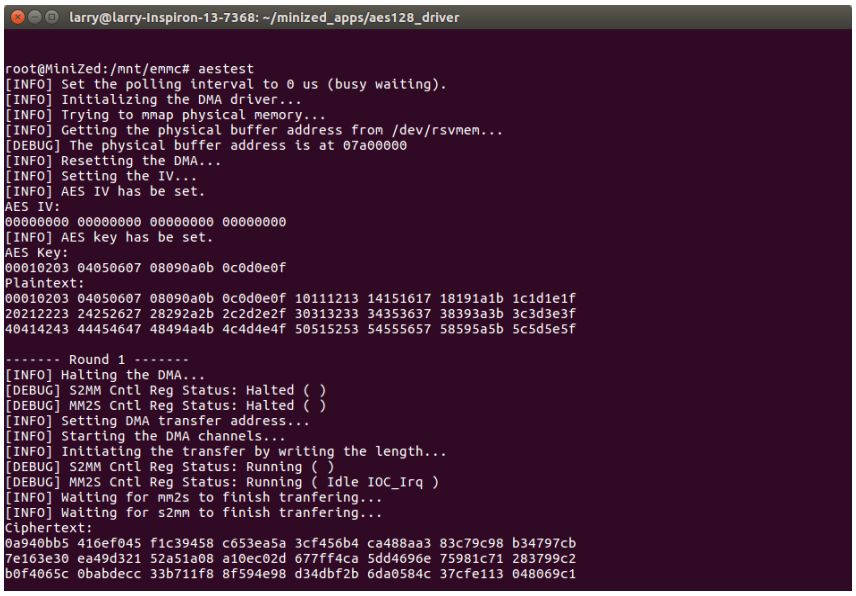

Figure 26: Screenshot of Hardware AES Encryption Test Run

The IoT camera and a extra power cable are connected to the board. Since the UVC driver in the kernel has been enabled and properly configured, the camera is expected to be recognized as a UVC node in the OS at /dev/video0. Gstreamer Tool with a variety of plugins is used to retrieve and decode the video stream. The test of the entire system is done using the UNIX shell filters techniques. The video stream is sent out to a file descriptor which is fed to the standard input of the AES128 program, which encrypts the input stream in chunks of a user-specified size and then writes the ciphertext to the standard output. The standard output is assigned to the standard input of the netcat tool, which sends the data to the client side. On the client side, the netcat tool and the same AES128 program running in software encryption mode are used to receive and decrypt the stream. The decrypted data is then stored in a video sink and displays the video on the window.

3.4.3 Client Test Verification

This section will explain the how the client devices will perform AES 128 decryption and confirm the validity of the encrypted stream.

3.4.3.1 Android Client Verification

The Android client is a Kotlin language developed application that streams in the encrypted message from the server through a socket connection and implements the native Java Cipher library for decryption. Since the encryption algorithm used is AES 128, the inputted streams are decrypted at 16 byte blocks at a time as seen in Figure 27 and displayed onto a text view or converted into a bitmap to confirm if the decrypted result matches the original file.

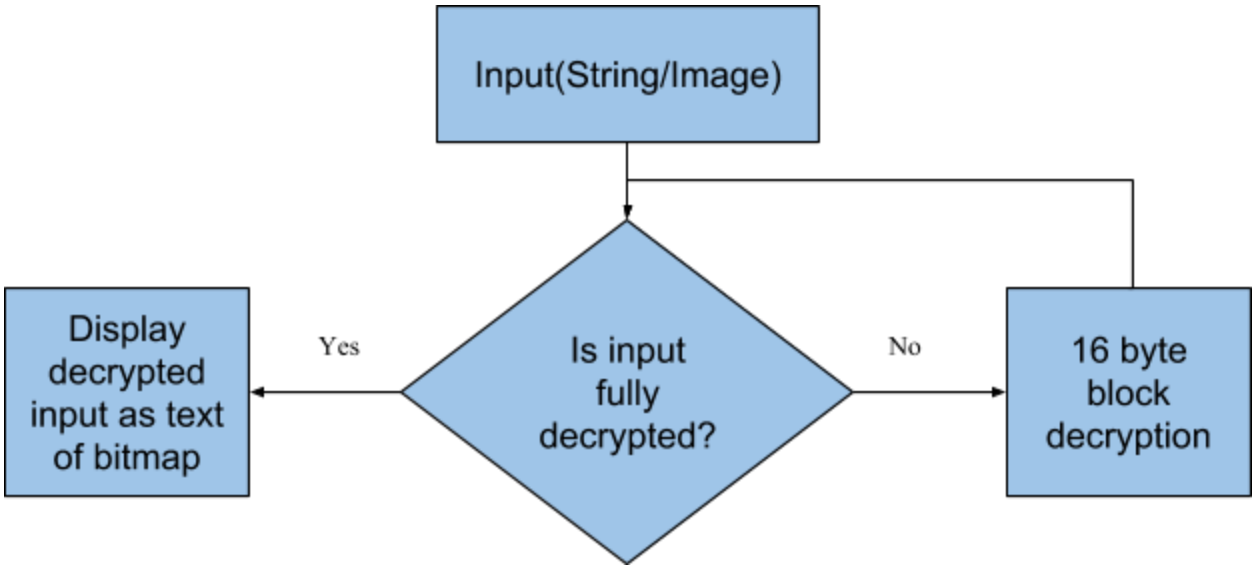

Figure 27. Android Client Verification Flowchart

3.4.3.2 Computer Client Verification

The computer client uses netcat to listen to a given port and pipelines the input stream into the aes decryption library. To display the decrypted stream, the computer client implements the same libraries used as the server vision processing unit. The verification process and pipeline can be seen in Figure 28.

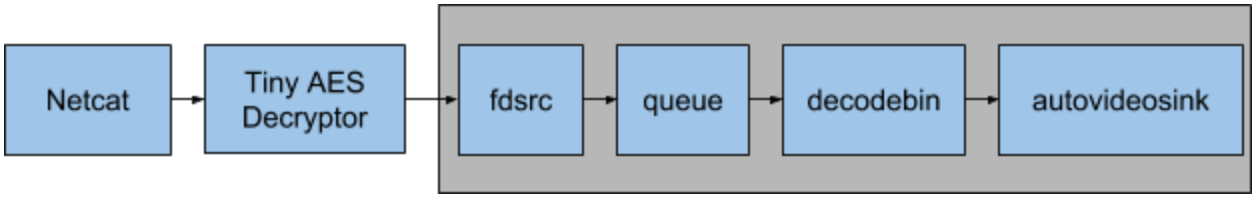

Figure 28. Computer Client Input and Vision Processing Pipeline

As the client streams in the encrypted video, it is being sent to the Tiny AES decryptor to turn it into a decrypted format. This is pipelined as a standard input to “fdsrc,” or file descriptor source. Once again “queue” is used here as previously mentioned in the IoT vision processing unit because it is necessary to make sure that the stream being outputted and inputted have the same type and buffer size, else there would be a buffer pool non-negotiated related error. After this is the “decodebin,” which constructs a decoding pipeline to convert the decrypted input into a format that can be displayed through “autovideosink.” Just like “autovideosrc,” the video sink version scans for an available video player that can play the given format.

Section 4. Results and Discussions

4.1 Result Analysis

The results and evaluations of the system from different perspectives are presented in this section. In general, the implemented system meets the major goals of the design.

4.1.1 Timing Analysis

Latency is one of the major criteria of the overall performance of a system. Especially for systems that interact with sensors or other human related I/O, latency has direct impacts on the user experience. Therefore, timing analysis of the crypto-accelerator is performed in comparison with the a regular software-based cryptography process on the same system. The overall network streaming latency is not directly measured because the latency can vary a lot under different network systems, network traffic condition, client system, etc., More importantly, the main purpose of the design is to reduce the security overheads for different IoT devices, not just a network video streaming device.

The software encryption program used for comparison is a modified version of tiny-AES-c [14], which is an open-source project that aims to provide small and portable AES encryption software. Table 6 below presents the result of the timing experiment. The testing chunk size starts ranges from 16 bytes to 512KB, in steps of power of 2. The time is measured in real time for both hardware and software AES128 encryption in CBC mode. Both hardware and software have the same input plaintext and key, so they are expected to perform the exact same encryption process on the same set of data. The latency is shown in the second and third column of Table 6. The speed up in the fourth column is the speed up in terms of the latency for the corresponding chunk size.

Size (bytes)	Software (s)	Hardware (s)	Speed Up	Software Avg. (us/byte)	Hardware Avg. (us/byte)
16	0.000076	0.000165	0.4606	4.750000	10.31250000
32	0.000145	0.000165	0.8788	4.531250	5.156250000
64	0.000283	0.000165	1.7152	4.421875	2.578125000
128	0.000562	0.000165	3.4061	4.390625	1.289062500
256	0.001119	0.000167	6.7006	4.371094	0.652343750
512	0.002284	0.000167	13.677	4.460938	0.326171875
1024	0.004538	0.000168	27.012	4.431641	0.164062500
2048	0.009016	0.000168	53.667	4.402343	0.082031250
4096	0.017983	0.000175	102.76	4.390381	0.042724609
8192	0.035949	0.000297	121.04	4.388306	0.036254883
16384	0.071681	0.000333	215.25	4.375061	0.020324707
32768	0.143042	0.000520	275.08	4.365295	0.015869141
65536	0.285893	0.000945	302.53	4.362381	0.014419556
131072	0.571544	0.001779	321.27	4.360535	0.013572693
262144	1.142762	0.003389	337.20	4.359291	0.012928009
524288	2.285511	0.006695	341.38	4.359267	0.012769699

Table 6: Software and Hardware Encryption Time

Figure 29 below shows the time the hardware and software encryption takes to encrypt the given chunk size of data. Note that both X and Y axis are in logarithm of two scales. The software encryption time is very close to a linear function with a slope of approximately 4.4 us/byte. The hardware encryption time, on the other hand, remains constant when the chunk size is smaller than 4KB. It then gradually goes up and eventually resembles a linear function with a of approximately 0.012 us/byte. Note that the hardware is only slower than the software when the chunk size is 32 bytes or below. The hardware outperforms the software as the chunk size increases. The possible reason why the hardware is slower than the software when the chunk size is small is constant time the DMA takes to start a transfer, which is also measured and it happens to be around 111 us. Note that this is the actual processor time it takes per chunk of encryption, no matter how large the chunk is. To put it differently, the processor is free to perform other computation or controls during the all the hardware encryption time except for this 111 us.

The hardware speed up is plotted in Figure 30. We can see that the speedup goes up dramatically when the chuck size is less than 4KB. For chunk size larger than 256KB, the increment of speed up is not very obvious, but they are still slowly increasing at a rate of about 0.016 per kilobyte.

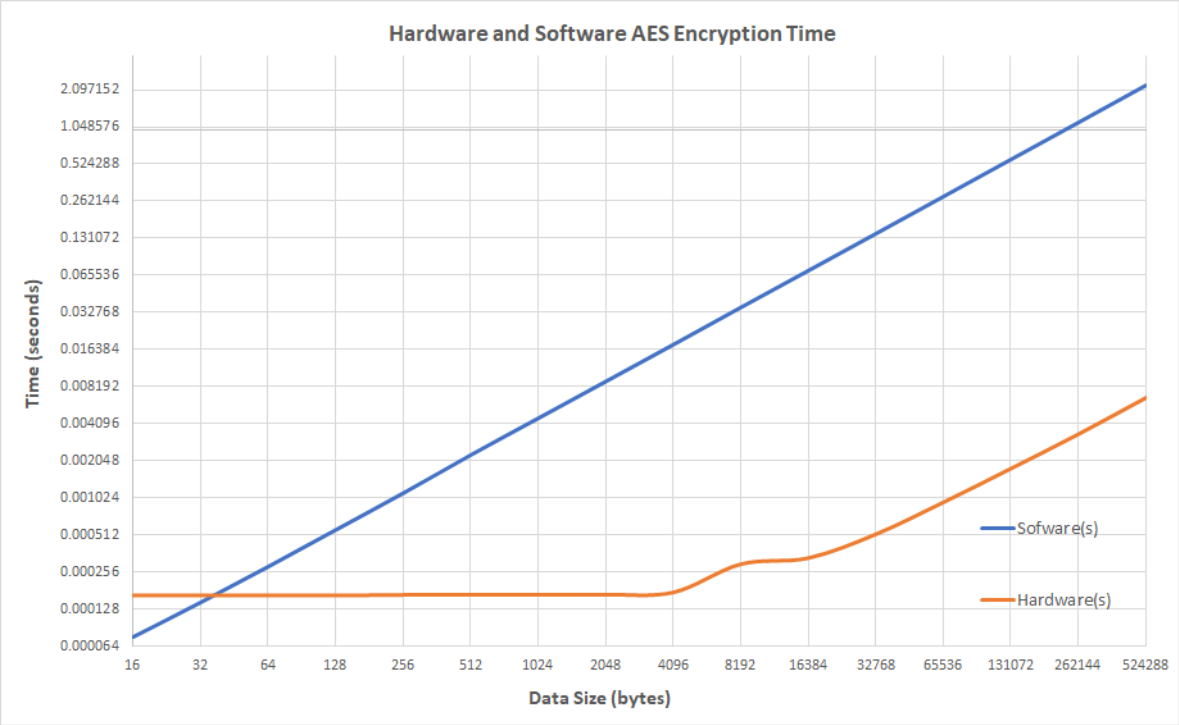

Figure 29: Hardware v.s. Software AES Encryption Time

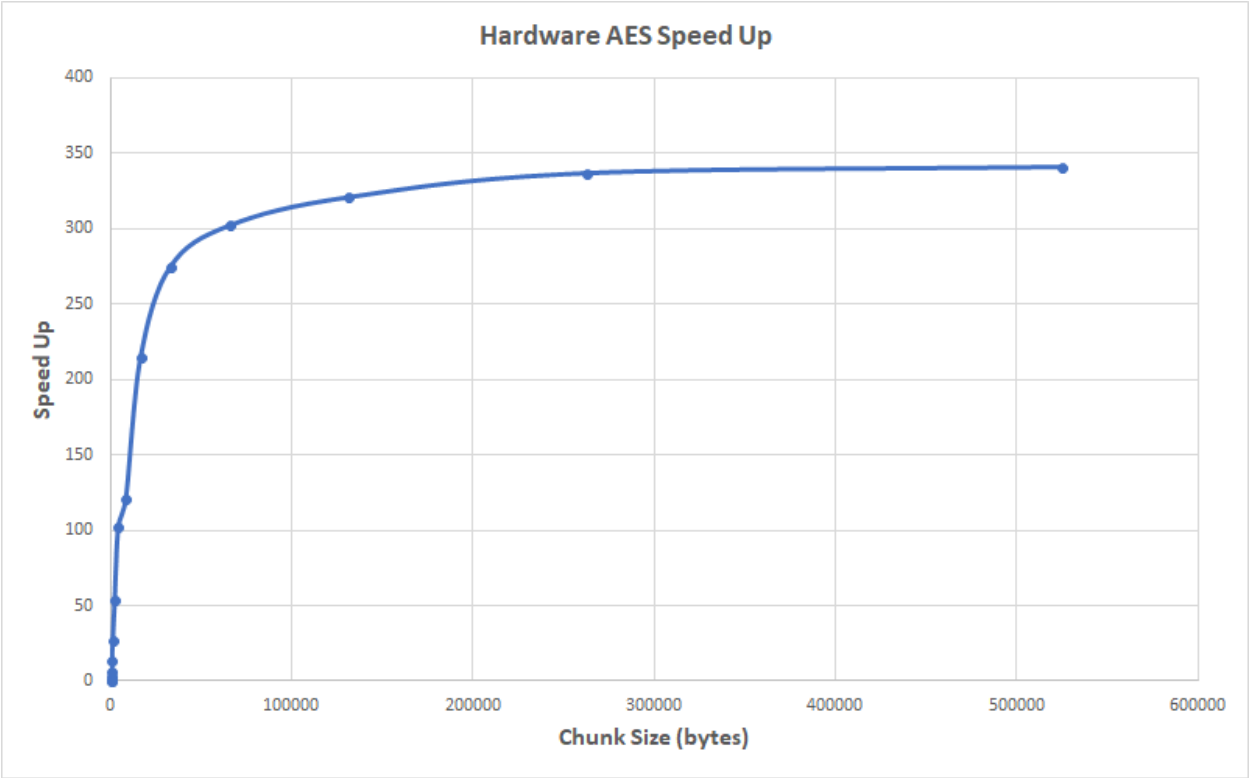

Figure 30: Hardware AES Speed Up

4.1.1 System Throughput

Throughput is another key factor when evaluating the system's performance. The throughputs of the AES crypto-accelerator and the overall system with the demo IoT camera are measured. Similar to the latency measurement, the throughput of crypto-accelerator with different chunk size ranging from 16 bytes to 512KB is presented in Table 7 below.

Size (bytes)	Software Throughput (KB/s)	Hardware Throughput (KB/s)
16	205.5921053	94.69696970
32	215.5172414	189.3939394
64	220.8480565	378.7878788
128	222.4199288	757.5757576
256	223.4137623	1497.005988
512	218.9141856	2994.011976
1024	220.3613927	5952.380952
2048	221.8278616	11904.76190
4096	222.4322972	22857.14286
8192	222.5374837	26936.02694
16384	223.2111717	48048.04805
32768	223.7105186	61538.46154
65536	223.8599756	67724.86772
131072	223.9547611	71950.53401
262144	224.0186496	75538.50693
524288	224.0199238	76474.98133

Table 7: Software and Hardware AES Encryption Throughput

Figure 31 plots the hardware and software throughout against the data chunk size. The software throughput remains roughly constant at approximately 220 KB/s, whereas the hardware throughput goes up rapidly with increasing chunk size. The hardware throughput eventually saturates when chunk size reaches 256KB, at around 74.5 MB/s. Note that the hardware has around 350 times larger throughput than that of the software with chunk size of 512KB.

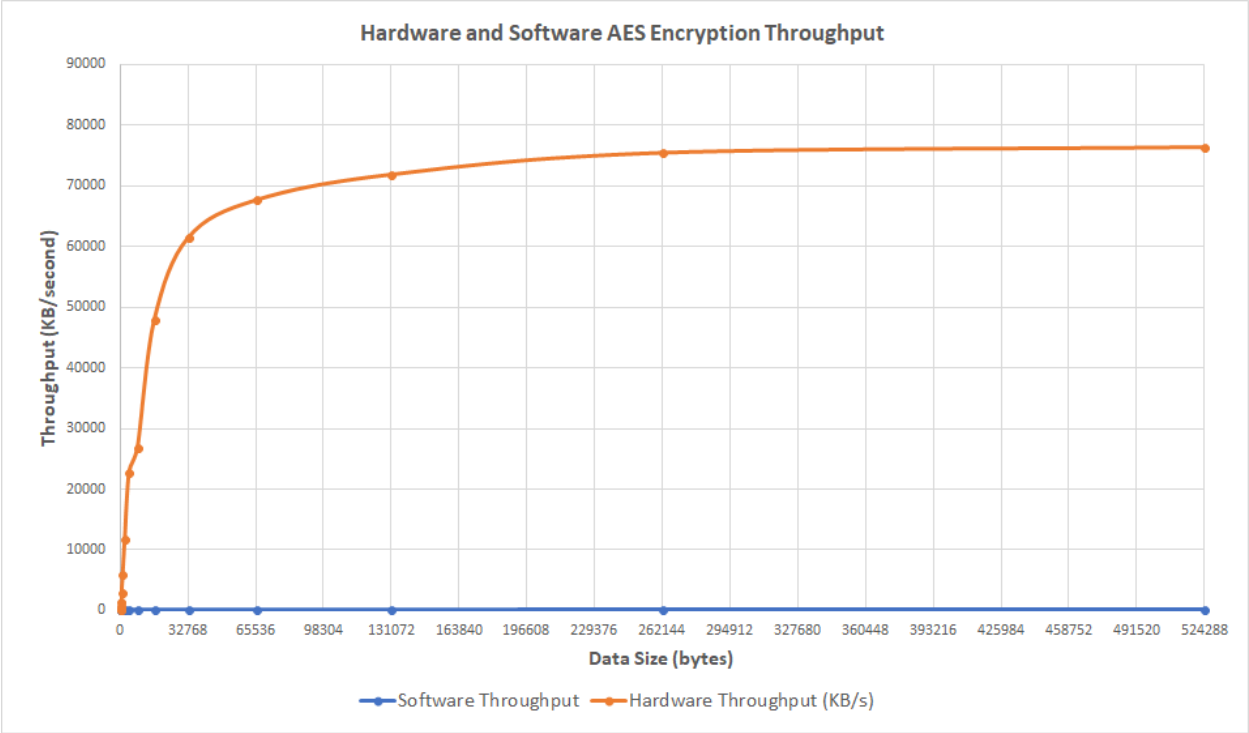

Figure 31: Hardware v.s. Software AES Encryption Throughput

With Figure 29 in the timing analysis section and Figure 31 above, optimized range of chunk size for high throughput and low latency can also be determined. Figure 32 plots the relationship between the throughput and the latency. For the hardware encryptor, it is notable that the higher throughput is, the longer the latency tends to be. This is because larger chunk size is required for higher throughput, but the higher the chunk size is, the longer the latency would be. The latency remains roughly constant when the chunk size is less than 4KB, and the throughput increases relatively slower when the chunk size is larger than 64KB. Therefore, assuming that the input source (the plaintext) is an infinite stream, the best chunk size to use would fall between 4KB and 64KB depends on the system performance criteria. In the IoT camera example presented in this paper, a 16KB hardware encryption buffer is used. Note that for IoT devices with small data rate, smaller chunk size would probably leads to better performance since it takes time to fill the buffer and this introduces fairly large latency.

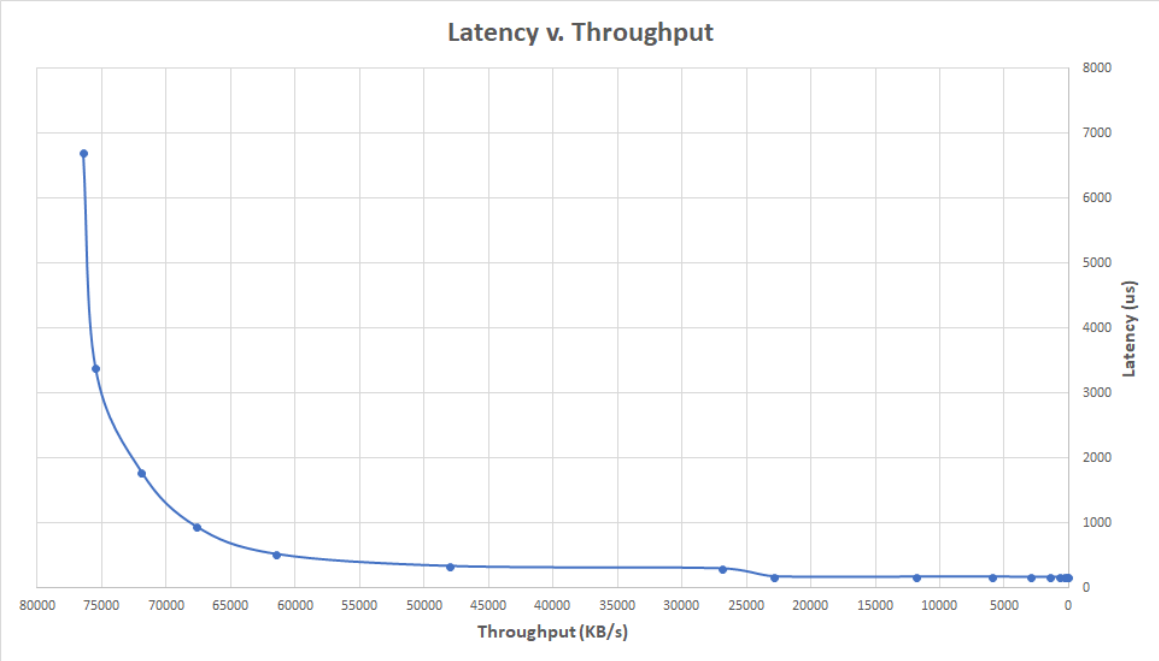

Figure 32: Throughput v.s. Latency of the Hardware AES Encryptor

As for the overall streaming quality, the demo that runs with Gstreamer for IoT camera video streaming in h263 format has its peak throughput at around 6 FPS. The MJPEG encoding format with highest resolution has approximately 3 FPS. The streaming throughput does not really meet the original expectations, so further analysis is performed to locate the bottleneck.

First, the size of a MJPEG frame obtained from Gstreamer is measured. The frame generally ranges from 38 KB to 42 KB. Since the AES encryption throughput has been measured to be more than 40 MB/s with chunks of 16 KB, it is a reasonable prediction that the bottleneck is at other steps. The throughput of the AES encryption and the network streaming using netcat is then measured. The assumption of this measurement is an ideal decoded video stream that has been located in the memory. The throughput turns out to be around 2.64 MB/s under an average Wi-Fi router in normal household. From the measurements above, it can be estimated that if the video source is ideal, the system should be able to reach 67 FPS with MJPEG encoding frames.

The throughput of the video streaming without the encryption step is also measured. The peak throughput is also around 3 FPS, which is almost the same as that of the one with encryption. Note that the FPS is an only estimation since it varies all the time. Based on these results, it can be concluded that the bottleneck of this system is in the Gstreamer, which reads from the camera and decodes the raw data. It is suspected that the PS does not have enough computing power to carry out the image decoding smoothly given that it has a relatively cost-efficient and power-efficient single core Cortex A9 process without any other dedicated graphic chips. A potential solution to this bottleneck is to have the PL or another dedicated SoC

carry out the video decoding process. Again, the throughput with ideal video source is estimated to be about 67 FPS.

3.4.4 Energy Efficiency

Another important criterion for IoT devices is the power consumption since IoT devices may run on a battery and do not have a large heat sink. It is relatively difficult to reduce the static power consumption, so in this section we focus more on the estimately dynamic power consumption. Table 8 shows the estimated power consumption of the system generated by the implementation tool in the Vivado Design Suite. The PS power consumption cannot be avoided since the operating system is constantly running on the chip to provide services. The AES accelerator accounts for about 0.219W of the total 1.548W dynamic consumption, which means that the chip would consume roughly 1.548W - 0.129W = 1.419W without the AES accelerator.

Category	Power (W)
Total On-Chip Power	1.548
➔ Dynamic	1.425
➔ Static	0.124
Processing System	1.177
Total AES Accelerator	0.219
➔ AXI-DMA	0.009
➔ AXILite	0.001
➔ AXIS_AES128_CBC	0.203
➔ AXIS_Data_FIFO_0	0.003
➔ AXIS_Data_FIFO_1	0.003
PS + AES Accelerator	1.396

Table 8: Estimated Power Consumption from the Implementation Report

The energy efficiency can be obtained by multiplying the encryption throughput with the estimated power consumption. Table 9 shows the energy efficiency for both hardware and software encryption in kilobytes of ciphertext per joule.

Size (bytes)	Software Ener. Efficiency (KB/J)	Hardware Ener. Efficiency (KB/J)
16	174.6746859	67.83450551
32	183.1072569	135.6690110
64	187.6364117	271.3380221
128	188.9719021	542.6760441
256	189.8162806	1072.353860

512	185.9933608	2144.707719
1024	187.2229335	4263.883204
2048	188.4688714	8527.766407
4096	188.9824105	16373.31150
8192	189.0717788	19295.14823
16384	189.6441561	34418.37253
32768	190.0684100	44081.99251
65536	190.1953913	48513.51556
131072	190.2759228	51540.49714
262144	190.3302035	54110.67832
524288	190.3312861	54781.50525

Table 9: Hardware and Software AES Energy Efficiency

Although adding the AES accelerator increases the power consumption per unit of time, the hardware encryption is much faster than the software encryption when the chunk size is large, which leads to a much better energy efficiency than the software. While the hardware is less energy efficient when chunk size is smaller than 32 bytes, it is about 288 times more energy efficient than the software with chunk size of 512KB. Figure 33 on the next page shows the plot of hardware and software energy efficiency with different chunk sizes. From the graph, we can see that the software has a constant energy performance while the hardware has better bytes per joule when chunk size is larger.

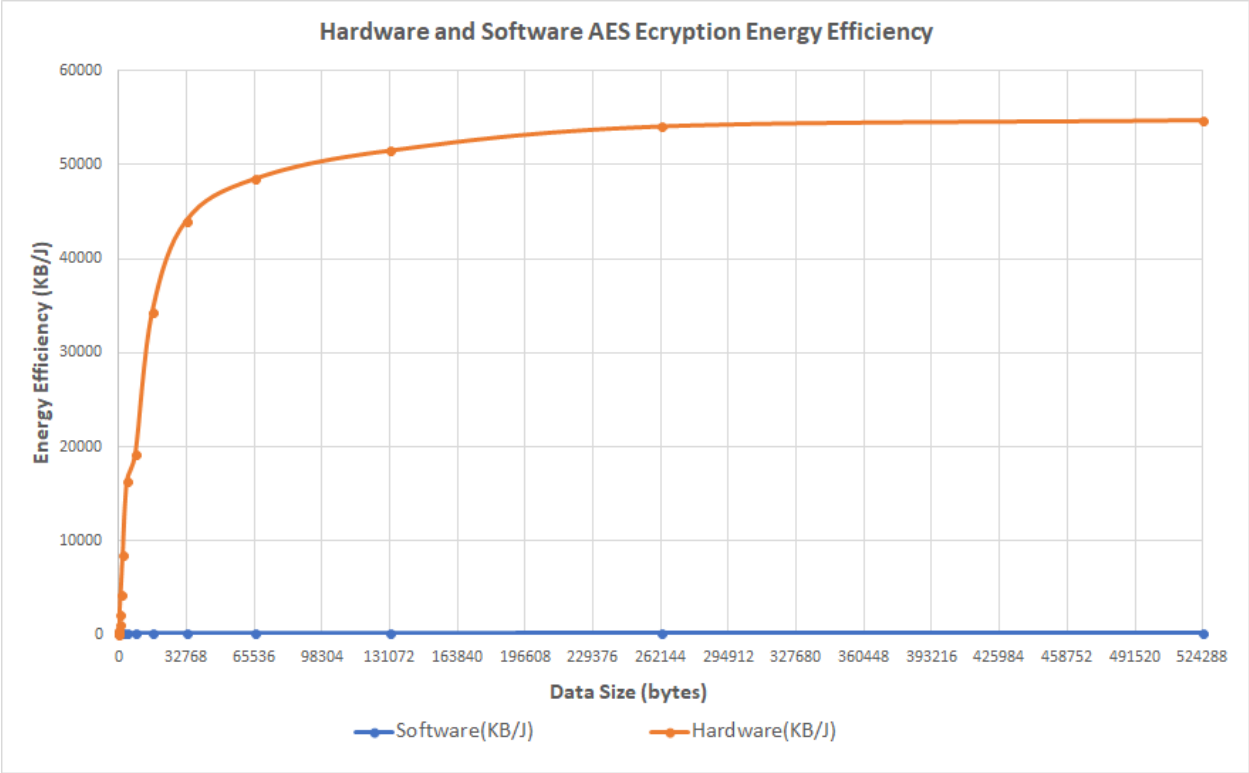

Figure 33: Hardware and Software AES Energy Efficiency Plot

Overall, the implemented system meets the major goal of increasing network security without introducing too much overhead. While the IoT camera streaming demo does not meet the expectations, the bottleneck has been found to be the video decoding process, which is not listed in the main goals of the design. In fact, the latency, throughput, and energy efficiency of the hardware encryption is better than the expectations. In addition, both vertical scalability and horizontal extensibility are taken into account when designing and implementing the system. Therefore, we believe a variety of future variants of the system can be implemented in order to meet other requirements.

4.2 Multi-Disciplinary Issue

The current team dynamic consists of two CE and one EE major related students. The design of the hardware crypto-accelerator on FPGA using hardware description language is done by the EE major who specializes in microelectronics. One of the CE students works alongside the EE major to develop a software abstraction for the hardware accelerator while developing drivers for use in the embedded operating system. The other CE student is responsible for the design and implementation of the software program and libraries on the operating system. This CE student also develops the secure network communication module and test client across multiple platforms. The operating system configuration is done by both CE students, given that both of their parts rely on and have to be compatible with the operating system. The

implementation of the AES encryption algorithm will be done by the collaboration of all three team members.

The project is monitored weekly by meetings of the two CE and one EE student with the respective professor. The time is used to address any concerns involving difficulties, roadblocks and new ideas regarding this project. Referencing the section 2.2 project planning, the students and professor define a project schedule. This provides students with goals that they aim to achieve and allows the professor to monitor the progress of the project. The students also have access to computer within the computer lab with all necessary programming software for the project.

This project can be applied to non EE/CE areas by introducing a trending and highly influential topic to students. Highly influential being that this technology is becoming more widely known and impacting more areas than previously operated in before. For example, the study of block chaining in cryptography is applicable to the understanding and development of cryptocurrencies such as Bitcoin. The knowledge of cryptography is also applicable towards students in areas such as CS or mathematics. As for the knowledge of FPGA programming, this can be valuable towards projects in fields that will require large amounts of computation and require such an example involving hardware acceleration.

4.3 Professional/Ethical Considerations

This project develops a new tool that can accelerate the encryption and decryption of data. This data will come from IoT devices that handle a variety of tasks. This type of data can be video from a device such as a baby monitor or voice recordings from devices such as Google's Nest. It is necessary that this project performs professionally as described to preserve these datas from fraudulent hackers and attacks. As for ethics, this type of device can also introduce a faster method for hackers to use against encrypted data. Since it is also an accelerated decrypter, a malevolent source can implement this to increase the speed of his/her attack when performing attacks such as man in the middle.

4.4 Impact of Project on Society/Environment

With a growing interest and implementation of IoT devices the concerns for security is growing as well. The multitude of IoT devices perform different tasks and collect a variety of personal data through sensors such as cameras or microphones. Popular devices such as Amazon's Nest are constantly listening and collecting surrounding information about the house either through its sensors or other connect IoT devices, making it very important to make sure that this data is secure and not handed out freely. With an increase in these devices there is a larger risk of leaking this data because not all IoT devices are kept to a standard of security or

users may not properly update software to prevent attacks. There are many cases where users don't properly update the firmware of their devices and companies may not update older devices, which leaves these devices vulnerable to attacks. By introducing this project, it can properly add that integral layer of security for these loopholes.

Section 5. Summary and Conclusions

With the increase of IoT devices, it is necessary to ensure the quality of the end-to-end security between the server and embedded devices. This maintenance is of utmost importance to the integrity of a company and its consumers since IoT devices constantly collect a variety of personal data. Security is the key to the wide adoption of IoT devices, and we believe that the proposed design can be helpful in solving such problems.

The proposed design is implemented on the MiniZed development board which comes with an energy efficient single core ARM processor and supports a variety of physical ports and logical interfaces for different IoT peripherals. Eliminating the burden of enhancing the level of security, the implemented system effectively provides an extra layer of security for the IoT devices without introducing too much overheads. The demonstration of the system with an UVC IoT camera proves that the implemented system functions correctly and meets the project goals. While the FPS of the demo is lower than the expectations, the analysis indicates that the bottleneck is in the image processing library Gstreamer. The hardware encryption with secure network streaming can easily reach 2.5 MB/s throughput provided that the network is stable. A variety of software packages, interfaces, libraries, and user programs are developed or installed on the system in order to provide a flexible, scalable, and extensible abstraction for currently running IoT management systems to integrate with.

For the client side, both a computer and Android test client programs are setup to stream input encrypted text, images and video to confirm that server is properly encrypting files as well as to confirm the improvements of using hardware accelerated encryption over software encryption.

Acknowledgements

We would like to acknowledge Professor Milder for his insight and help during this project. The weekly meetings allowed us to bring questions about the theory of the project and each hourly meeting allowed us to figure out more and more on how to design the project, choose the development board, decide on the program to use with the board, specify what encryption algorithm to use, and how to handle our open source libraries.

We would also like to thank the EE department for their support. Tony's help by setting up the lab room greatly facilitated the process of the board development and programming. Professor Tang for her advice and help towards searching for a proper project, introduction to professionals in the working field and providing a rubric for writing this paper along with the proper citations. Also, for the EE department's reimbursement towards academic research for this project.

References

[1] A. Hodjat and I. Verbauwhede., “Interfacing a high speed crypto accelerator to an embedded cpu”. In Proc. 38th Asilomar Conference on Signals, Systems, and Computers, volume 1, pages 488–492, November 2004.

[2] M. Katagi and S. Moriai., “Lightweight Cryptography for the Internet of Things”. Sony Corporation, 2008

[3] M. Usman, I. Ahmed, M. I. Aslam, S. Khan, and U. A. Shah, “SIT : A Lightweight Encryption Algorithm for Secure Internet of Things”. vol. 8, no. 1, 2017.

[4] Rajender Manteena, “A VHDL Implementation of the Advanced Encryption Standard-Rijndael Algorithm”, University of South Florida, March 2004. pp. 13

[5] Michael H. Behringer, “End-to-End Security,” The Internet Protocol Journal, vol 12, September 2009

[6] Hamdan o. Alanazi, B. B. Zaidan, A. A. Zaidan, Hamid A. Jalab, M. Shabbir, Y. Al-Nabhani, “New Comparative Study Between DES, 3DES, and AES within Nine Factors,” Journal of Computing, vol. 2, March 2010. pp. 152--157

[7] Xilinx, “LogiCORE IP AXI DMA v7.1”, Vivado Design Suite, PG021, October 2017.

[8] Xilinx, “AXI Reference Guide”, Vivado Design Suite, UG1037, v4.0, July 2017. pp. 47--50

[9] Jeff Johnson, “Using the AXI DMA in Vivado”, FPGA developer, Vivado, August 2014.

[10] Salleh M, Ibrahim S., “Image encryption algorithm based on chaotic mapping”, Universiti Teknologi Malaysia, JTMKK39. 2003. pp. 8--10

[11] Vazhayil A, “AES-Advanced Encryption Standard”, wordpress, April 2015. <https://captanu.wordpress.com/tag/aes/>

[12] “AES mix column stage”, Cryptography Stack Exchange, May 2018. <https://crypto.stackexchange.com/questions/19986/aes-mix-column-stage>

[13] Muehlberghuber, Michael, “aes128-hdl”, Github, Open Source Under GPL-2.0, <https://github.com/mbgh/aes128-hdl>

[14] “tiny-AES-c”, Github, Open Source Under Unlicense, <https://github.com/kokke/tiny-AES-c>