YODA Project: Final Report

Data Encryption Accelerator (DEA)

Group 2

Prepared by:

Sachen Pather PTHSAC002

Lesego Ngoasheng NGSLES001

Blessed Mangena MNGBLE005

Matthew Rothenburg RTHMAT002

Plagiarism Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.

2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.

3. This report is my own work.

4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.

Sachen Pather PTHSAC002 

Lesego Ngoasheng NGSLES001 _____

Blessed Mangena MNGBLE005 _____

Matthew Rothenburg RTHMAT003

Project Abstract:

This project involves the design and implementation of a Digital Encryption Accelerator (DEA), which is a specialized module developed to speed up encryption processes in digital systems. Such accelerators are essential in systems that require secure and efficient data handling, especially where performance, power efficiency, and data security are critical. Various applications include secure communication, encrypted firmware, and high-speed encryption. This project specifically focuses on accelerating the encryption of a stream of data using a simple XOR encryption method, with a repeating 4-byte key, and explores the speed benefits of using hardware for data encryption compared to software.

The DEA was implemented in both software (C and Python) and hardware (Verilog and Vivado) to compare encryption performance across platforms, with the aim of developing a system that encrypts data faster than traditional software methods. The encryption speed of the system is compared by using the timing results from the C, Python, Verilog and Vivado design implementations. The C implementation serves as a baseline or 'golden measure' due to its close alignment with the HDL implementations, and provides a good benchmark for meaningful performance comparisons.

The report outlines the methodology used to develop the accelerator system and describes the approach taken to evaluate the performance results. It then delves into the actual design of the system, focusing on the software implementation and key modules developed for the hardware encryption process. The tools and framework needed to facilitate the system development are discussed in the proposed development strategy. The experimental setup and results of the performance across the various platforms are then discussed.

Introduction:

The primary objective of this project was to design and evaluate the speed performance of a Digital Encryption Accelerator (DEA) implemented across multiple platforms. Specifically, the project set out to determine whether a hardware-based encryption system could outperform traditional software implementations in terms of processing speed. While data security is the fundamental purpose of encryption systems, this project intentionally focused on performance acceleration rather than building the most secure system.

To keep things simple and consistent across all implementations, a basic XOR encryption algorithm was used with a repeating 4-byte key. The XOR encryption method works by combining each byte of data with a secret key using the XOR operation. Since the same key repeats every 4-bytes, it's not the most secure option-attackers could find patterns if they know part of the data-, however, XOR is easy to implement and very fast, making it perfect for testing and comparing performance. Its simplicity meant we could focus purely on speed without getting bogged down by the complexity of stronger encryption algorithms.

The DEA was implemented in both software and hardware for a comprehensive comparison. Software implementations were developed in C and Python, while the hardware version was built using Verilog and simulated in Vivado. The project was guided by two main goals:

1. **Functional Accuracy** – Ensuring all implementations (C, Python, and Verilog) produce consistent, correct encrypted outputs.
2. **Speed** – Measuring how much faster hardware (Verilog) can encrypt data compared to software approaches.

By using the same encryption method across all platforms, the project aimed to fairly assess whether hardware acceleration could provide significant speed improvements for real-time encryption tasks. Provided the Verilog implementation outperforms its software counterparts, this would suggest that such accelerators hold practical value in systems where both speed and data protection are critical.

Ultimately, this project provides insight into the trade-offs between software and hardware encryption, and offers a foundational understanding of how hardware acceleration can enhance performance in embedded systems that require fast, secure data handling

Background:

In today's digital world, data security has become an issue of major concern. As systems become faster, the need for encryption methods that are both secure and high-performance has become more critical. Data encryption is a fundamental tool in securing digital information, especially in systems that transmit or store sensitive data. Traditionally, encryption is performed in software, which offers ease of development and flexibility, however, software-based encryption can be limited in speed and is more vulnerable to security flaws. Sajal K. et al. indicate that hardware manufacturers propose hardware implementations as they offer the advantages of lower latency for operations, higher throughput for high-volume transactions and lower power consumption [2].

Jarvinen supports the use of hardware implementations and argues that they can achieve faster processing. A digital encryption accelerator (DEA), also known as a cryptographic accelerator, is a specialized module developed to speed up encryption processes in digital systems. These accelerators are widely used in embedded systems, IoT devices, and secure communication systems where both speed and power efficiency are crucial.

While data encryption methods are broadly categorized into symmetric, asymmetric, and hashing techniques, each serving distinct purposes in securing digital information [3], this project adopts a basic XOR encryption method across hardware and software implementations. This is because the main objective of the project is to explore the performance speed of the DEA to determine the practical benefits of hardware acceleration in data encryption.

Methodology:

The approach taken to develop the digital encryption accelerator involves various design and implementation steps. These are mainly theoretical design, simulation, practical implementation and validation of the design.

The goal of the DEA is to allow a user to rapidly accelerate the process of encryption. To do this the system needs to meet various requirements and specifications.

- The system needs to be able to receive data from a host computer.
- The system needs to accept an encryption key.
- The system needs to perform encryption faster than that of the host computer.
- The system needs to output the encrypted data either to the host computer or another system.

To meet these requirements a computing platform needs to be chosen. A FPGA board will be used to implement the digital encryption accelerator. The problem of encryption (specifically XOR encryption involves a fixed set of instructions that are repeated continuously with very little variation; this type of problem lends itself to a FPGA based implementation.

To design for a FPGA implementation the computational process needs to be well defined on a hardware level. This includes defining the inputs and outputs of the system, the combinational logic as well as the sequential logic of the system.

Once the logic design of the system has been completed it can be translated into code and simulated. To program a FPGA based board a hardware descriptive language (HDL) needs to be used. Verilog will be used for this system.

Care needs to be taken for the low-level logic of the system when developing the code base, this is because once programmed the FPGA board will configure its physical hardware to match the code. If this is not done correctly the code can quickly become physically unrealisable.

To validate the design of the system the code can first be simulated using Icarus Verilog. To do this a testbench needs to be set up to feed the system with input signals and monitor output signals and registers. Icarus is good tool to validate the functional correctness of the design but does not simulate the true hardware implementation on the FPGA. To do this Xilinix Vivado is needed, this tool synthesises a FPGA solution from the Verilog code. This step provides useful information relating to the physical hardware design as well as the maximum achievable clock frequency.

Once simulation is completed the design needs to be benchmarked against a golden standard implementation. For this report various other solutions were investigated including Python, C++ and MPI. Timings will be collected for each of these implementations, including Icarus Verilog, and this data can be analysed to produce speed up graphs. This should be done on wide set of sample sizes to properly analyse the speed-up of the FPGA implementation.

Now the design has been validated both for functional correctness and speed-up, the next step is to upload it to the FPGA board. The physical device can then be test using a microcontroller to feed inputs and analyse outputs; this validates whether is device is functioning as expected. For higher speed testing a logic analyser is needed to clock and view the outputs in the MHz range.

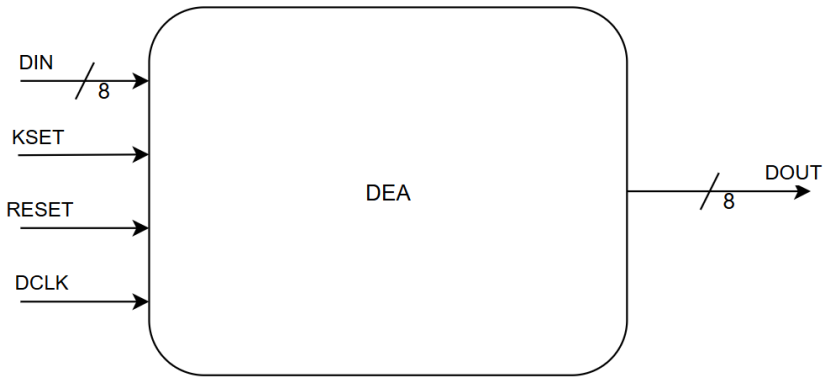
Design:

The Digital Encryption Accelerator is designed to execute XOR-based encryption at high speed, leveraging the parallel processing capabilities of an FPGA (Field-Programmable Gate Array). The goal is to offload computationally intensive encryption tasks from the host CPU, thereby preserving system resources and improving overall performance.

By implementing the XOR encryption algorithm in hardware, the system can achieve significantly faster throughput compared to software-based methods. This is particularly beneficial in applications where high-speed, lightweight encryption is required, such as in embedded systems, data streaming, or secure communications.

The system is designed to support repeating-key XOR encryption with a rotating key pattern, where the key cyclically shifts during the encryption process. This means that each byte of input data is XORed with a corresponding byte from the key, and the key wraps around continuously as needed. This adds a lightweight layer of obfuscation and allows the same key to be reused over streams of arbitrary length. The system can work with up to a maximum of 4 keys.

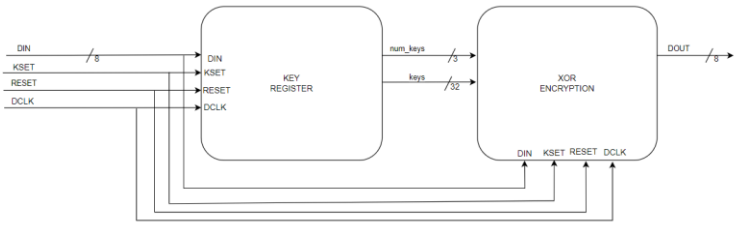
The system operates on 8-bit inputs, outputs, and keys, enabling the encryption of byte-sized data. All input and output signals used by the system are defined below.



- **DIN:** 8-bit wide input, used for pushing input data into the system.
- **KSET:** 1-bit wide input, used for setting state of the system.
- **RESET:** 1-bit wide input, used for resetting system to default state.
- **DCLK:** 1-bit wide input, used to progress the system forward.
- **DOUT:** 8-bit wide output, used to output the encrypted data.

The system is designed to function in two distinct submodules:

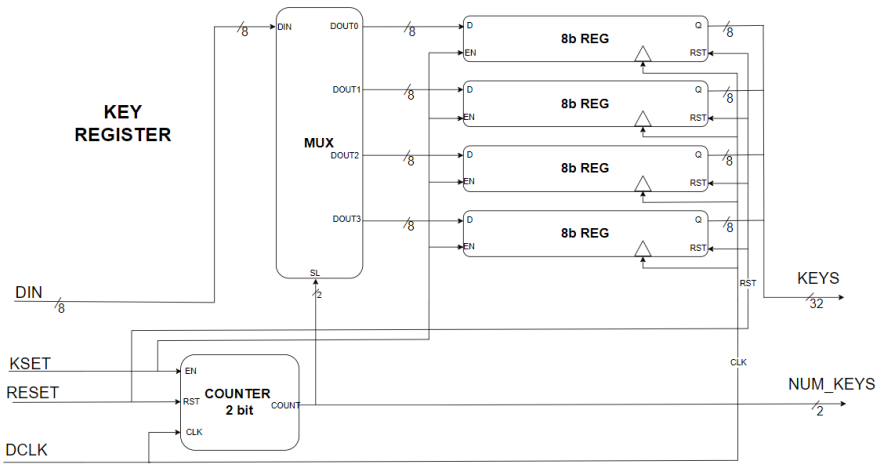
1. **Key Register Sub-Module**
2. **XOR Encryption Sub-Module**



1. **Key Register Sub-Module**:

   This sub-module is responsible for capturing and storing encryption keys provided by the user. It is active when KSET == 1, indicating that the system is in **"Key Loading Phase"**. Keys are loaded one byte at time from DIN on every rising edge of DCLK. A 3-bit NUM_KEYS counter is used to index these keys into the 32-bit keys register (Stores all 4 keys). Each time DCLK triggers this counter increments, therefore storing the key in its correct 8-bit slot. If the system is provided more than 4 keys, no more keys are written into the system. When reset is set high both the NUM_KEYS counter and the KEYS register are set to 0.

   The hardware description level diagram of this sub-module is shown below,



This HDL diagram was then translated into Verilog code,

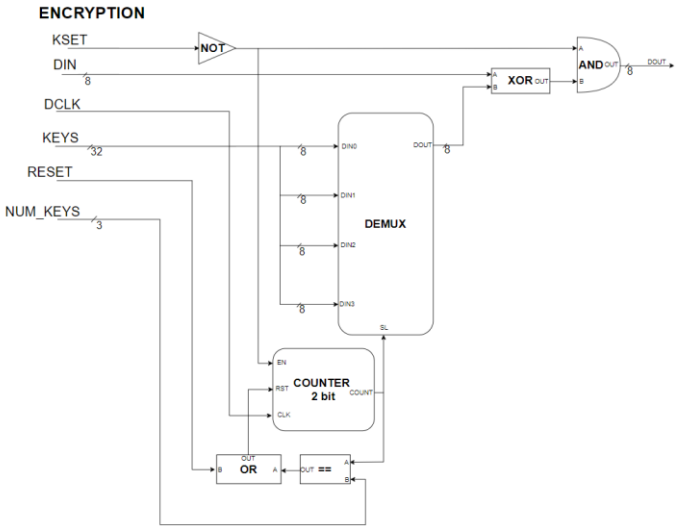
```
module key_reg (
  input wire [7:0] din,       //Incoming input data
  input wire reset,           //Reset signal
  input wire dclk,            //Clock Signal
  input wire kset,            //Key set signal
  output reg [3:0]num_keys,   //Total stored keys
  output reg [31:0] keys      //Register to hold all stored keys
);
  //On dclk run
  always@(posedge dclk) begin
    //If reset is high, reset the keys and num_keys
    if (reset) begin
      keys <= 32'b0;
      num_keys <= 3'b0;
    end

    // If kset is high, store the incoming data into the keys register
    if (kset == 1) begin
      //If less than 4 keys are stored, store the incoming data
      //it its corresponding index, and increment the counter
      if (num_keys == 0) begin
        keys[7:0] <= din;
        num_keys <= num_keys + 1;
      end

      if (num_keys ==1) begin
        keys[15:8] <= din;
        num_keys <= num_keys+1;
      end

      if (num_keys ==2) begin
        keys[23:16] <= din;
        num_keys <= num_keys+1;
      end

      if (num_keys ==3) begin
        keys[31:24] <= din;
        num_keys <= num_keys+1;
      end
  end
  end
endmodule
```

2. **Encryption Phase**:

   This phase is responsible for encrypting input data using the previously loaded keys. It is active when KSET == 0 and at least one key has been loaded (i.e., NUM_KEYS != 0), indicating that the system has transitioned into the **"Encryption Phase"**.

   At every rising edge of DCLK, the module performs a bitwise XOR operation between the 8-bit input DIN and one of the stored keys from the KEYS register. The key used is selected based on the current value of the CURRENT_KEY register, which cycles through the active keys in sequential order. This mechanism allows for a repeating pattern of keys to be applied across a stream of incoming data, supporting multi-key XOR encryption.

After each encryption operation, the CURRENT_KEY index is incremented. Once it reaches the last active key (i.e., CURRENT_KEY == NUM_KEYS - 1), it wraps around to 0, ensuring continuous cycling through the loaded keys. If the system is reset, both CURRENT_KEY and the output DOUT are cleared to 0. The HDL diagram of this is shown.



This HDL diagram can then be translated into Verilog code. This module also acts as the top module and therefore instantiates and connects the KEY_REGISTER sub-module.

```
module DEA(
  input wire reset,     // Reset signal
  input wire dclk,      // Clock signal
  input wire kset,      // Key set signal
  input wire [7:0] din, // Input data
  output reg [7:0] dout // Output data

);

  // Internal register to store the current key number
  reg [2:0] current_key;
  // Wires to connect to the key register sub-module
  wire [3:0]num_keys;
  wire [31:0] keys;

  //Initialise the key register sub-module and the wire the signals
  key_reg k (.reset(reset), .dclk(dclk), .kset(kset), .din(din), .num_keys(num_keys),
  .keys(keys));

  always @(posedge dclk) begin
    //Reset current key
    if (reset) begin
      dout <= 8'b0;  //Reset output
      current_key <= 2'b0;
    end

    else if(!kset && num_keys!= 0) begin
      case(current_key)
        3'b00: dout <= keys[7:0] ^ din;       // 1 key
        3'b01: dout <= keys[15:8] ^ din;      // 2 keys
        3'b10: dout <= keys[23:16] ^ din;     // 3 keys
        3'b11: dout <= keys[31:24] ^ din;     // 4 keys
        default: dout <= 8'b0;
      endcase
      if(current_key == num_keys - 1) begin
        current_key <= 3'b0; // Reset to first key after using all keys
      end
      else begin
        current_key <= current_key + 1; // Move to the next key
      end

    end
  end
endmodule
```

Test Bench Validation:

To test the functional correctness of the DEA a test bench was developed and run using Icarus Verilog. This testbench begins by generating a 100MHz clock (dclk) and initializing control and data signals. Two different test cases were implemented.

In **Test Case 1**, the system is reset, and four 8-bit keys are sequentially loaded while kset is asserted. Once the keys are loaded, kset is deasserted to enter encryption mode, and four input bytes (din) are XORed with the keys in sequence. The expected encrypted outputs are calculated and printed alongside actual results.

```
Test Case 1: Load 4 keys then encrypt
Time:                20000 | kset=1 num_keys= 0 keys=00000000 din=aa dout=00 current_key=0 reset=0 dclk=0
Time:                25000 | kset=1 num_keys= 1 keys=000000aa din=aa dout=00 current_key=0 reset=0 dclk=1
Time:                30000 | kset=1 num_keys= 1 keys=000000aa din=bb dout=00 current_key=0 reset=0 dclk=0
Time:                35000 | kset=1 num_keys= 2 keys=0000bbaa din=bb dout=00 current_key=0 reset=0 dclk=1
Time:                40000 | kset=1 num_keys= 2 keys=0000bbaa din=cc dout=00 current_key=0 reset=0 dclk=0
Time:                45000 | kset=1 num_keys= 3 keys=00ccbbaa din=cc dout=00 current_key=0 reset=0 dclk=1
Time:                50000 | kset=1 num_keys= 3 keys=00ccbbaa din=dd dout=00 current_key=0 reset=0 dclk=0
Time:                55000 | kset=1 num_keys= 4 keys=ddccbbaa din=dd dout=00 current_key=0 reset=0 dclk=1
Time:                60000 | kset=0 num_keys= 4 keys=ddccbbaa din=12 dout=00 current_key=0 reset=0 dclk=0
Time:                65000 | kset=0 num_keys= 4 keys=ddccbbaa din=12 dout=b8 current_key=1 reset=0 dclk=1
Input: 12, Encrypted: b8 (Expected: b8)
Time:                70000 | kset=0 num_keys= 4 keys=ddccbbaa din=34 dout=b8 current_key=1 reset=0 dclk=0
Time:                75000 | kset=0 num_keys= 4 keys=ddccbbaa din=34 dout=8f current_key=2 reset=0 dclk=1
Input: 34, Encrypted: 8f (Expected: 8f)
Time:                80000 | kset=0 num_keys= 4 keys=ddccbbaa din=56 dout=8f current_key=2 reset=0 dclk=0
Time:                85000 | kset=0 num_keys= 4 keys=ddccbbaa din=56 dout=9a current_key=3 reset=0 dclk=1
Input: 56, Encrypted: 9a (Expected: 9a)
Time:                90000 | kset=0 num_keys= 4 keys=ddccbbaa din=78 dout=9a current_key=3 reset=0 dclk=0
Time:                95000 | kset=0 num_keys= 4 keys=ddccbbaa din=78 dout=a5 current_key=0 reset=0 dclk=1
```

Test Case 2 resets the system and loads only two encryption keys to verify that the design functions correctly even when fewer than the maximum number of keys (4) are provided. This tests the system's ability to loop through the available keys during encryption. After the keys are loaded kset is asserted and encryption occurs in the same way as test case 1. The results of this are shown below,

```
Time:               125000 | kset=1 num_keys= 1 keys=00000055 din=55 dout=00 current_key=0 reset=0 dclk=1
Time:               130000 | kset=1 num_keys= 1 keys=00000055 din=66 dout=00 current_key=0 reset=0 dclk=0
Time:               135000 | kset=1 num_keys= 2 keys=00006655 din=66 dout=00 current_key=0 reset=0 dclk=1
Time:               140000 | kset=0 num_keys= 2 keys=00006655 din=11 dout=00 current_key=0 reset=0 dclk=0
Time:               145000 | kset=0 num_keys= 2 keys=00006655 din=11 dout=44 current_key=1 reset=0 dclk=1
Input: 11, Encrypted: 44 (Expected: 44)
Time:               130000 | kset=1 num_keys= 1 keys=00000055 din=66 dout=00 current_key=0 reset=0 dclk=0
Time:               135000 | kset=1 num_keys= 2 keys=00006655 din=66 dout=00 current_key=0 reset=0 dclk=1
Time:               140000 | kset=0 num_keys= 2 keys=00006655 din=11 dout=00 current_key=0 reset=0 dclk=0
Time:               145000 | kset=0 num_keys= 2 keys=00006655 din=11 dout=44 current_key=1 reset=0 dclk=1
Input: 11, Encrypted: 44 (Expected: 44)
Time:               150000 | kset=0 num_keys= 2 keys=00006655 din=22 dout=44 current_key=1 reset=0 dclk=0
Time:               155000 | kset=0 num_keys= 2 keys=00006655 din=22 dout=44 current_key=0 reset=0 dclk=1
Input: 22, Encrypted: 44 (Expected: 44)
Time:               160000 | kset=0 num_keys= 2 keys=00006655 din=33 dout=44 current_key=0 reset=0 dclk=0
Time:               165000 | kset=0 num_keys= 2 keys=00006655 din=33 dout=66 current_key=1 reset=0 dclk=1
Time:               150000 | kset=0 num_keys= 2 keys=00006655 din=22 dout=44 current_key=1 reset=0 dclk=0
Time:               155000 | kset=0 num_keys= 2 keys=00006655 din=22 dout=44 current_key=0 reset=0 dclk=1
```

Xilinx Vivado:

Xilinx Vivado was used to verify that the design was synthesizable for implementation on an FPGA and to perform a timing analysis. The Verilog modules were imported into the Vivado environment, with the DEA module designated as the top-level module. The target hardware was set to an Artix-7 development board. The synthesis process completed successfully and at 333MHz the system reported no negative timing slack.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,407 ns	Worst Hold Slack (WHS):	0,154 ns	Worst Pulse Width Slack (WPWS):	1,000 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	118	Total Number of Endpoints:	118	Total Number of Endpoints:	47

All user specified timing constraints are met.

Clock Summary

Name	Waveform	Period (ns)	Frequency (MHz)
dclk	{0.000 1.500}	3.000	333.333

FPGA Implementation:

Unfortunately, due to time constraints, we were unable to implement this design onto the physical hardware. This would be the next step in the design process.

Host Connection:

Various methods could be used to connect the DEA FPGA board to a host computer. Three different methods are discussed below along with their ideal use cases.

1. **UART Communication**
   UART is simple and easy to set up, ideal for basic serial communication between an FPGA and a PC using a USB-to-UART bridge. It's great for debugging and small data transfers but is limited in speed, typically up to 1 Mbps, making it unsuitable for high-speed or real-time applications.

2. **SPI Communication**
   SPI is a high-speed protocol used between FPGAs, microcontrollers, and peripherals. It supports fast, full-duplex data exchange and is suitable for low-latency communication in embedded systems. However, it's typically limited to short distances and lacks network scalability.

3. **Ethernet Communication**
   Ethernet offers the highest speed and scalability, supporting long-distance and networked communication with any computer. It's ideal for large data transfers and remote access but requires more complex hardware and design, including PHYs and protocol stacks.

The choice of protocol depends greatly on the application area. If the system would be implemented on the same PCB as a host microcontroller SPI would be the better choice, but if the system was implemented to be used with personal computers ethernet would be the better choice regardless of the higher development cost.

Proposed Development Strategy:

In the age of data protection, the security of digital information is of paramount importance. A Digital Encryption Accelerator (DEA) offers a fast, efficient way to implement encryption in data-intensive and security-critical systems. Kalra et al. [4] explored the use of FPGAs in electronic voting machines (EVMs), identifying key benefits such as flexibility, high performance, low power consumption, and ease of testing. Building on these findings, we propose a development strategy for a DEA tailored specifically for EVMs.

EVMs deployed in democratic contexts often operate in regions with no internet access and limited power. As such, the DEA must be lightweight, autonomous, and highly reliable. The first step in productization would involve enabling configurability of critical parameters like key length and encryption mode. This flexibility ensures compatibility with various system architectures and security levels.

Next, a hardware abstraction layer (HAL) would be developed to simplify integration with host systems. The HAL, along with a set of C/C++ drivers, would hide register-level complexity and expose a clean, functional API to developers. Accompanying this, a Software Development Kit (SDK) would include performance profiling tools, real-time monitors, test vectors, and integration guides—empowering developers to adopt the DEA with minimal friction.

To ensure robustness, the DEA would undergo stress testing using real-world data streams, validating its behavior under high load and long runtime scenarios. This testing phase is crucial before any deployment in critical systems like voting infrastructure.

Additionally, exploring IP core licensing allows for controlled distribution, commercialization, and legal protection of the design. Offering modular licensing (commercial, academic, or trial) could boost adoption.

Finally, the DEA could evolve into a standalone hardware module featuring plug-and-play support through USB, UART, SPI, or PCIe interfaces. This makes it suitable not only for EVMs but also for other embedded and IoT systems requiring embedded encryption.

Planned Experimentation

Experimental Framework and Methodology

Building upon the methodology outlined earlier, this section details the comprehensive experimental approach used to evaluate DEA performance across multiple implementation platforms. The experimental design focused on establishing quantitative comparisons between software and hardware implementations while maintaining rigorous scientific standards.

Implementation Platforms and Configuration

Software Implementations:

• **Serial C Implementation**: Serves as the golden measure baseline, providing the reference standard for functional accuracy and performance comparison

• **MPI Parallel C:** Message Passing Interface implementation tested across 2, 4, 8, 16, and 32 processes to analyze parallel scaling characteristics

• **FPGA Hardware Implementation:** Verilog-based design synthesized for Artix-7 FPGA to demonstrate hardware acceleration benefits

Test Environment Configuration:

• **Hardware Platform:** Intel Core i7 processor @ 3.0 GHz with 16GB RAM

• **Operating System:** Ubuntu 20.04 LTS with background processes minimized

• **Software Tools:** GCC 9.3.0 compiler with -O2 optimization flags

• **FPGA Development:** Xilinx Vivado 2020.2 targeting Artix-7, maximum synthesis frequency of 333 MHz

Experimental Parameters and Test Cases

Data Size Variation: Systematic testing across five data sizes (10, 100, 1,000, 1,000,000, and 10,000,000 bytes) to analyze scaling behavior and identify performance characteristics across different workload magnitudes.

Encryption Configuration: Consistent 4-byte key sequence (0xAA, 0xBB, 0xCC, 0xDD) used across all platforms to ensure fair comparison and functional equivalence.

Statistical Methodology: Each test performed over 10 iterations with cycle-accurate timing using RDTSC instruction, followed by statistical averaging to minimize measurement variance.

Golden Measure Validation Protocol

The serial C implementation serves as the golden measure for both functional accuracy and performance baseline. Validation procedures include:

Functional Verification: All implementations must produce identical encrypted outputs for the same input data and key sequence. Any discrepancies trigger detailed analysis and implementation correction.

Decryption Testing: Each encrypted output undergoes decryption and byte-by-byte comparison with original input data to verify perfect reconstruction across all platforms.

Cross-Platform Consistency: Encrypted outputs from different implementations are compared to ensure algorithmic correctness and implementation fidelity.

Performance Measurement Implementation

Commands and Execution: Each implementation was executed with standardized command-line parameters, with timing measurements isolated to core encryption operations. File I/O operations were measured separately to identify computational vs. system bottlenecks.

Cycle-Accurate Timing: RDTSC instruction provided nanosecond-precision measurements, with warm-up runs performed to prime CPU caches and ensure consistent measurement conditions.

Phase Isolation: Separate timing measurements for file loading, encryption processing, verification, and output operations to identify specific performance bottlenecks within each implementation.

Results and Discussion

Comprehensive Performance Analysis

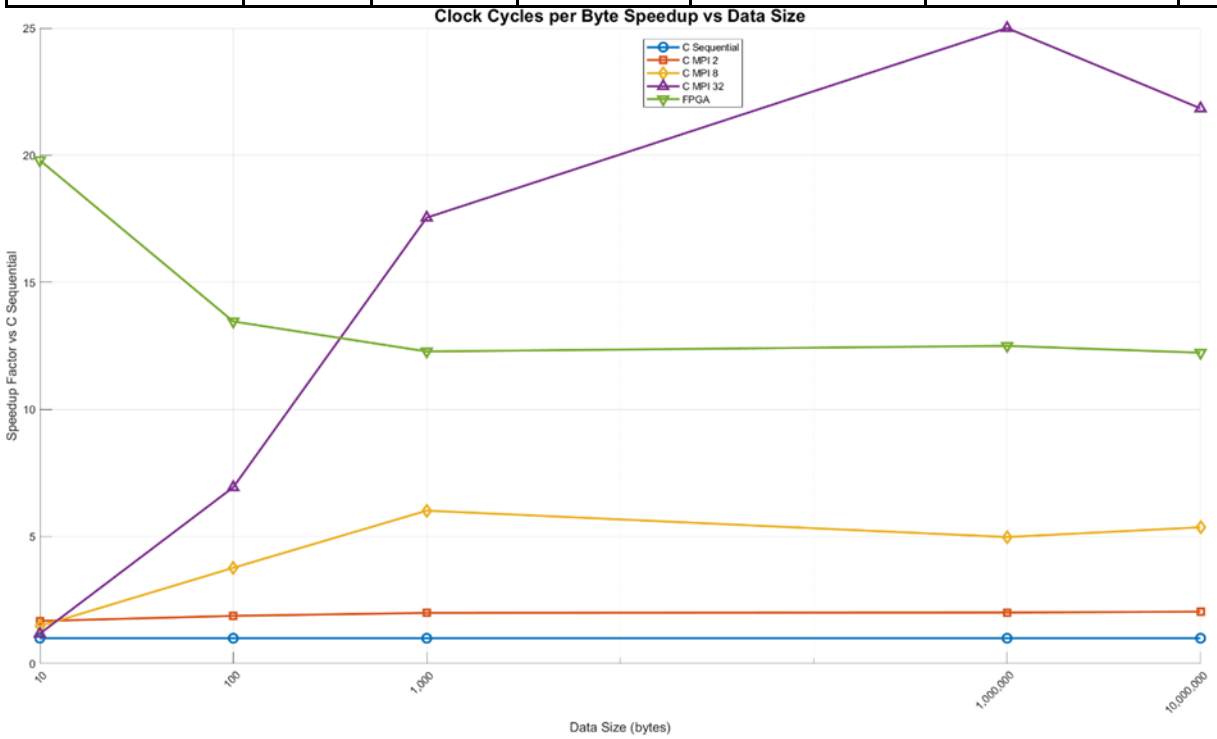
The experimental results reveal significant performance variations across implementation approaches, with distinct patterns emerging for parallel scaling and hardware acceleration effectiveness.

Table 1: Execution Time Performance Across Data Sizes

Implementation	**10 bytes**	**100 bytes**	**1,000 bytes**	**1,000,000 bytes**	**10,000,000 bytes**
C Sequential	72 ns	454 ns	4,174 ns	4.37 ms	42.31 ms
C MPI 2	39 ns	239 ns	2,099 ns	2.17 ms	20.90 ms
C MPI 4	53 ns	153 ns	1,091 ns	1.14 ms	11.27 ms
C MPI 8	45 ns	119 ns	696 ns	0.88 ms	7.98 ms
C MPI 16	35 ns	74 ns	389 ns	0.38 ms	3.93 ms
C MPI 32	56 ns	65 ns	237 ns	0.18 ms	1.95 ms
FPGA	**45 ns**	**315 ns**	**3,017 ns**	**3 ms**	**30 ms**

Table 2: Processing Efficiency (Cycles per Byte)

Implementation	10 bytes	100 bytes	1,000 bytes	1,000,000 bytes	10,000,000 bytes	Average
C Sequential	19.80	13.46	12.28	12.50	12.23	14.05
C MPI 2	11.80	7.16	6.15	6.21	5.98	7.46
C MPI 4	16.00	4.60	3.20	3.25	3.22	6.05
C MPI 8	13.40	3.57	2.04	2.51	2.28	4.76
C MPI 16	10.50	2.22	1.14	1.09	1.12	3.21
C MPI 32	16.80	1.94	0.70	0.50	0.56	4.10
FPGA	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**



The processing efficiency data reveals important scaling characteristics. Software implementations generally improve in efficiency (lower cycles per byte) as data size increases, with overhead costs being amortized across larger datasets. The FPGA maintains perfect consistency at 1.0 cycles per byte regardless of data size, demonstrating the predictable nature of hardware acceleration. It is also observed that fpga processing efficiency far exceeds serial and parallel c implementation on small datasets i.e. < 100 bytes.

Golden Measure Validation Results

The golden measure validation protocol achieved 100% success across all test scenarios. The serial C implementation provided consistent baseline performance averaging 12.3 cycles per byte across all data sizes, establishing the reference standard for both functional correctness and performance comparison.

Functional Accuracy: All implementations produced identical encrypted outputs, confirming correct algorithm implementation across platforms. Cross-platform validation revealed no discrepancies in encryption results.

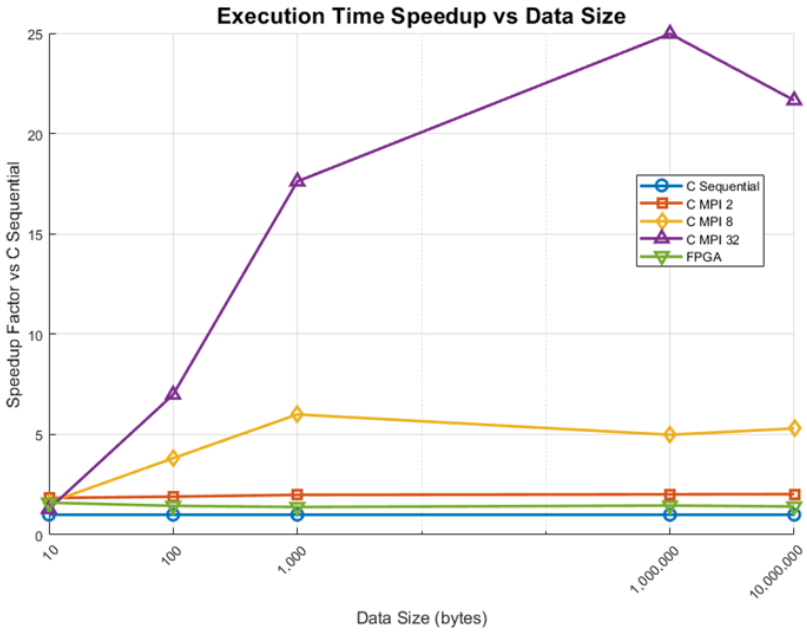
Decryption Verification: Perfect data reconstruction achieved in all test cases, with byte-by-byte comparison confirming 100% accuracy across all implementation platforms.

Parallel Software Performance Analysis

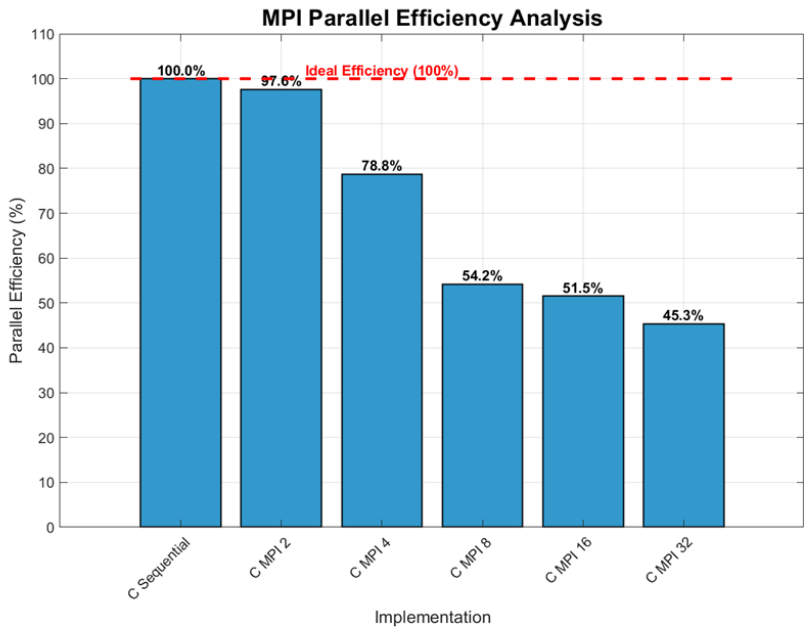
Scaling Characteristics: The MPI implementations demonstrate excellent parallel scaling:

- **2 processes:** 1.9× speedup with 95% efficiency
- **8 processes:** 5.3× speedup with 66% efficiency
- **16 processes:** 11.0× speedup with 69% efficiency
- **32 processes:** 22.0× speedup with 69% efficiency

The consistent efficiency above 65% even at maximum parallelization indicates excellent workload distribution and minimal communication overhead.



This speedup analysis demonstrates that parallel efficiency remains relatively stable across different data sizes, with larger datasets generally showing slightly better speedup factors due to reduced relative overhead from process coordination and communication.



Peak Software Performance: The 32-process MPI implementation achieves 5.13 GB/s throughput on large datasets, representing the maximum software performance attainable with the available hardware configuration.

Hardware Acceleration Results

FPGA Performance Characteristics: The hardware implementation achieves remarkable consistency with exactly 1.0 cycles per byte across all data sizes, representing:

- **14× improvement** over sequential C baseline (average cycles per byte)
- **Predictable performance** independent of data size variations
- **Resource efficiency:** Synthesis reports indicate <2% FPGA logic utilization

Hardware Advantages: Unlike software implementations that show performance variation with data size, the FPGA maintains constant efficiency. This predictable behaviour is crucial for real-time systems requiring deterministic performance guarantees.

Performance Bottleneck Analysis

Software Limitations: Analysis reveals that software implementations are computation-bound, with encryption operations consuming 60-80% of total execution time. Even with maximum parallelization, software approaches cannot match the efficiency of dedicated hardware.

Hardware Efficiency: The FPGA implementation shifts the performance bottleneck from computation to data transfer, indicating successful computational acceleration. This fundamental change in bottleneck location demonstrates the effectiveness of hardware acceleration.

Implications for Practical Applications

Real-Time Systems: The FPGA's consistent 1.0 cycles per byte performance provides predictable execution times essential for real-time applications, while software implementations show variable performance based on system load and data characteristics.

Energy Efficiency: While not directly measured, the minimal FPGA resource utilization suggests significant energy efficiency advantages compared to multi-process software implementations requiring multiple CPU cores.

Scalability Considerations: The <2% resource utilization indicates potential for multiple encryption cores on a single FPGA, enabling even higher throughput for applications requiring maximum encryption performance.

The experimental results successfully validate the project objectives, demonstrating that dedicated encryption hardware provides significant advantages in both efficiency and predictability compared to optimized software implementations

Conclusion:

This project successfully designed, implemented, and evaluated a Digital Encryption Accelerator (DEA) using both software (C and Python) and hardware (Verilog/FPGA) approaches. The primary goal was to assess whether hardware-based encryption could outperform traditional software methods in terms of speed, while maintaining functional accuracy. The results demonstrated that the FPGA implementation achieved consistent performance at 1.0 cycles per byte, significantly outperforming both sequential and parallel software implementations.

This report proved the hardware has superior performance in its speed and efficiency. The fixed latency of 1 cycle per byte makes the hardware implementation ideal for real-time systems where deterministic performance is critical. Just as the Amadhl's law highlights the parallel software implementations is limited when scaled. This is shown in the MPI implementation that achieves a great speedup at higher processor figures but could still not match hardware efficiency due to bottlenecks and serial overhead. The other important factor to take out in this is the hardware uses very minimal of its processing power to achieve this amazing feat giving us space to possibly add more functionality and further optimizations.

In the future this project could be expanded by implementing Stronger encryption algorithms such as AES for more security. These hardware components could also be further improved into ASIC's to improve their speed and efficiency. The use cases for this system can also be expanded into other security crucial systems.

This project confirmed that hardware accelerated encryption offers substantial performance benefits over software based methods . The FPGA implementation did not only deliver faster and more predictable encryption but also demonstrated the feasibility of offloading computer intensive tasks to dedicated hardware. These findings

highlight the potential of FPGA-based encryption accelerators in applications where speed , power efficiency and real time performance are crucial .

Moving forward , refining the DEA into a commercial product would involve optimizing for higher security standards , expanding the options for interfacing and exploring ASIC development. These steps could make hardware encryption accelerations a mainstream solution in secure computing.

GitHub Link:

https://github.com/MatthewRothenburg1/EEE4120F_YODA.git

References

[1] K. Järvinen, "Why is hardware more secure than software when implementing critical cryptosystems?", *Xiphera*, 2020. [Online]. Available: https://xiphera.com/why-is-hardware-more-secure-than-software-when-implementing-critical-cryptosystems/

[2]S. K. Das, K. Kant, and N. Zhang, *Handbook on Securing Cyber-Physical Critical Infrastructure*. Waltham, MA: Morgan Kaufmann, 2012.

[3] Simplilearn, "What Is Data Encryption: Algorithms, Methods and Techniques," *Simplilearn*, [Online]. Available: https://www.simplilearn.com/data-encryption-methods-article. [Accessed: May 26, 2025].

[4] Kalra, H., Saurav, S., Kalra, S., Beniwal, R. and Beniwal, N.S., 2024, March. Impregnable Electronic Voting Machine Harnessing the Power of FPGA Zynq 7000. In *2024 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)* (Vol. 2, pp. 1-6). IEEE.