YODA Project
2.18: Digital Encryption Accelerator (DEA) using simulated FPGA and Opencl

Cameron Clark,[†] Robert Dugmore, [‡] Kian Frassek [§] and Si Teng Wu [¶]
EEE4120F Class of 2024
University of Cape Town
South Africa
[†]CLRCAM007 [‡]DGMROB001 [§]FRSKIA001 [¶]WXXSIT001

***Abstract*—The report details methods used implement a Digital Encryption Accelerator (DEA) using simulated Field-Programmable Gate Arrays (FPGAs) and Open Computing Language (OpenCL). A golden standard is developed in Python using numpy and compared to parallel implementation. A basic and advanced encryption algorithm are explored.**

The basic encryption algorithm implements XOR gate encryption with a rotating key, while the advanced implements further mathematical manipulation to further obfuscate the key each for different blocks of data. This adjustment limits the power of common frequency analysis code-cracking methods.

A simulated FPGA implementation using Icarus Verilog (iVerilog) is explored. Parallelism is implemented through multiple internal encrypter modules which can operate on different blocks of data simultaneously. The simulated FPGA communicates the key, the data being encrypted, and the outputted encrypted data via Quad Serial Peripheral Interface (QSPI).

The OpenCL implementation utilises the power of a General-purpose Graphics Processing Unit (GP-GPU). Data to be encrypted is transferred to the GP-GPU, where individual blocks of data can be encrypted in parallel.

The results from testing revealed that the two DEAs consistently produced the same encrypted data as the golden standard. Some acceleration was achievable with hardware solutions. A comparison of the two different DEA systems revealed that the FPGA DEA was faster for smaller data packages, while the OpenCL solution was faster for larger data packages.

I. Introduction

Fast and secure encryption has become critical in an age where vulnerable personal and business information must frequently be transmitted over the internet. This report outlines the investigation into the efficiency of parallelising a high-speed DEA using FPGAs and OpenCL. These implementations are compared to a sequential golden standard developed in Python.

The primary goal of this project is to achieve faster data encryption speeds by using FPGAS, OpenCL, and parallelisation. The simulated FPGA implementation aims to achieve this by splitting data among multiple parallel FPGA encryption modules which can encrypt different blocks of data simultaneously. Similarly, the OpenCL implementation achieves parallelism through execution on multiple cores of a GP-GPU.

Two implementations of encryption were examined. A basic implementation was first created as a proof of concept which used XOR encryption with a rotating key. Once the basic implementation was created, a more complex encryption algorithm was implemented to increase the safety of the encrypted data. Importantly, both encryption methods can be reused as the decryption method provides the key is known.

A. Prototype Specifications

The FPGA DEA is programmed using Verilog [1]. Simulation is performed using Icarus Verilog (iVerilog). The OpenCL DEA is programmed using OpenCL on top of C/C++. As previously stated, there will be two implementations available - a basic version and an advanced version. Below are the tables for basic implementation specifications I and advanced implementation specifications II.

Index	Description
S-B1	Uses XOR encryption
S-B2	Key size and data block size of 32 bits
S-B3	Rotate the key for each subsequent block of data
S-B4	Correctly implement a golden standard in Python
S-B5	Correctly prototype the DEA using iVerilog
S-B6	Correctly prototype the DEA using OpenCL

TABLE I: Specifications for basic design

Index	Description
S-A1	Encryption remains invertible (decryption uses the same algorihtm)
S-A2	Key size and data block size of 32 bits
S-A3	Obfuscate the key for subsequent blocks of data further than a simple rotation
S-A4	Correctly implement a golden standard in Python
S-A5	Correctly prototype the DEA using iVerilog
S-A6	Correctly prototype the DEA using OpenCL

TABLE II: Specifications for advanced design

II. Background

A. Data Encryption

Data encryption is a process that transforms data into a specific format, making it unreadable without the appropriate decryption key. This technique is crucial for maintaining the confidentiality and integrity of data, especially when sensitive and personal data is being transmitted through the public domain.

Some commonly used encryption algorithms include the Data Encryption Standard (DES) [2], Triple DES (3DES) [3], Advanced Encryption Standard (AES) [4] and Blowfish [5].

B. Exclusive Or (XOR) Gate Encryption Algorithm

The XOR algorithm encrypts data by performing the XOR $\oplus$ operation with a key. The XOR truth table is shown in Table III and an example of encryption is shown in Table IV. Note that a decryption of an XOR encryption is achieved by re-applying the XOR operation using the same encryption key.

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

TABLE III: XOR operation

Encryption								
Data	1	0	0	1	1	0	1	0
Key	1	0	1	0	1	0	1	0
Encrypted	0	0	1	1	0	0	0	0
Decryption								
Encrypted	0	0	1	1	0	0	0	0
Key	1	0	1	0	1	0	1	0
Decrypted data	1	0	0	1	1	0	1	0

TABLE IV: XOR Example

C. Data Encryption Accelerator (DEA)

As the amount of data that can be stored and used continues to increase with advancements in computing, encrypting all such data becomes an increasingly honourous task. A DEA is a dedicated tool used to increase the speed of data encryption, which would typically be used with large quantities of data. Such a dedicated tool additionally moves the task of data encryption away from a computing device's Central Processing Unit, allowing a complex system which must perform many different tasks to offload the encryption process and allocate it's processing time to other tasks.

Hardware acceleration can be performed either using purpose-made hardware devices, or general purpose devices such as external GP-GPUs, or parallel computing clusters. The former can typically be built to be more efficient than the latter, but general purpose devices often allows more flexibility in the control of the operation of the device, and can be used for other tasks at times when the device is not being used for encryption.

D. Parallelism

Parallelism in computing is a method of executing multiple tasks simultaneously. It involves breaking down a problem into smaller sub-problems that can be processed independently, ideally resulting in a faster execution of the problem [6].

E. Field Programmable Gate Array (FPGA)

An FPGA is an array of programmable circuit logic blocks and programmable interconnections that can be used to build custom digital circuits [6] [7]. This allows for faster speeds because, once the FPGA has been programmed, it becomes a dedicated piece of equipment for a specific task. FPGAs can be configured to run multiple tasks at the same time by using different gates to perform different procedures simultaneously, enabling parallelism [6].

III. Methodology

A. Experimental setup

The block diagram in Fig 1, below shows the general methodology for this project.

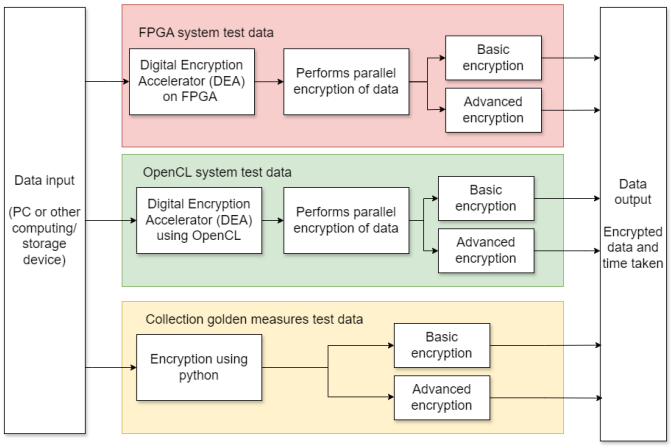

Fig. 1: Block diagram of the DEA in the context of the broader system

As can be seen from the block diagram the methodology behind the design of the FPGA DEA is as follows:

- Data from a storage device needs to be transmitted to the FPGA DEA as quickly as possible. Due to the nature of FPGAs and embedded systems, communication protocols are required to transfer this data. Some of the commonly used embedded system communication protocols are Inter-Integrated Circuit (I2C) communication, Serial Peripheral Interface (SPI), and Universal asynchronous receiver-transmitter (UART).
  SPI is one of the faster communication protocols [8]. An even faster form of SPI communication is quad SPI (QSPI) [9]. For this reason, QSPI communication is used to quickly transfer data to the FPGA. This section will be tested by checking the speed at which data can be transferred to the FPGA via QSPI.

- The inputted data will be broken up into sections of variable length, normally 32 bits, which will be encrypted by parallel encryptors.
- The basic encryption method will use XOR encryption with a rotating key.
- The advanced encryption method also uses XOR encryption, however, the key is further obfuscated for different blocks of data using multiple mathematical operations.
- Once data has been encrypted it will be transmitted back to the host device via QSPI.
- The time taken to encrypt the data was recorded.

As can be seen from the block diagram the methodology behind the design of the OpenCL DEA is as follows. Data was read from a text file. This data was then encrypted using parallel workers in the kernel and the output was exported to a text file. the time it took to encrypt the data was recorded.

The following methods were used to test the DEA system:

- the FPGA and OpenCL DEAs encryption times were compared.
- A golden measure was developed in Python for the purpose of checking the DEA encryption and setting up a benchmark.

B. Hardware

The computer used for Python golden standard and the OpenCL implementation was an Acer Predator Helios 300 laptop. FPGA analysis was performed entirely in simulation using iVerilog. The computer had the following specifications:

CPU:

- Model: Intel i5-9300H
- Processors: 4 physical, 8 logical (with hyperthreading)
- DRAM: 32GB DDR4-2666MHz
- Base Frequency: 2.4 GHz
- Max Turbo Frequency: 4.1 GHz

Dedicated GPU:

- Model: Nvidia Geforce GTX 1660 Ti
- VRAM: 6GB GDDR6
- Processors: 1536 cores
- Streaming Multiprocessors (Compute Modules): 24
- Base clock: 1500 MHz
- Boost clock: 1770 MHz

C. Validation

The validation procedure is used to ensure that each implementation is correctly encrypting the data. For each of the basic and advanced implementation, the golden standard is first developed. The code is written as simply as possible, to make it less likely that any errors would be present. A few pieces of sample data are calculated by hand, and compared to the outputted results from the golden standard, to ensure that the output is correct. Further validation comes from the fact that the same data is successfully decrypted by running it through the program a second time, with the same key.

Once the golden standard has been validated, it is used as the ground-truth data for the OpenCL and FPGA implementations. The data produced by these implementations is compared to the golden standard, ensuring that it has been correctly encrypted.

D. Timing

The speed of the various implementations (Python golden standard, OpenCL, FPGA) is timed to allow for an analysis of the speedup of the DEA modules. Timing tests are performed on randomly generated data, to avoid the need to store very large files

Timing tests included the time required to set up the kernel and data buffers for OpenCL implementation, but otherwise did not include the time taken to load data from the data files onto the heap. The assumption behind this is that the specific programs may process, load, and store their data in different ways; management of this is outside of the the scope of a dedicated hardware device.

For each test condition (number of blocks encrypted), the timing test was executed multiple times. The fastest value was regarded as the closest to the ‘true’ performance of the hardware and thus used in timing analysis. Typically, the variation in execution time for the same program comes from a combination of other tasks taking up processing time on general purpose devices, and ‘cache warm-up’, where programs execute faster on subsequent iterations owing to certain variables already being loaded into higher levels of cache. Taking the minimum time ensures that the test represents the closest to performance of the hardware in a case where other programs are properly optimised and the device itself is optimised for use. The use of new random data to be encrypted each time means that the execution time will not be disproportionately fast from having previous data loaded into the program.

IV. Design

The project was broken down into multiple parts for easier implementation and understanding.

A. Encryption Algorithm

A core element of the design process was to choose an appropriate encryption algorithm to parallelise. An initial concept used a simple exclusive-or (XOR) cipher, with a block and key size of 32 bits. This cipher is performed by taking the output of a bitwise XOR operation using the key and block of data as inputs. The cipher is reversible, since the mathematical properties of the XOR operation dictate that performing a XOR on the encrypted data and the key will yield the original data.

An XOR cipher, however, is not particularly secure. It is well known to be vulnerable to frequency analysis, where cipher-text outputs are compared to the most frequently occurring plaintext outputs. An example of this in ciphers operated on the English language is that the letter ’e’ occurs most frequently in the language - correlation of this to the most commonly occuring letter in the cipher quickly yields results that make the cipher much easier to crack. This process can be

applied to the most commonly occuring patterns in whatever data type is expectred to be transmitted using the cipher. An application of the well-known birthday problem would show that only a small amount of data is necessary for said data to start to show repetitions, leading to frequency analysis vulnerabilities for any but the smallest of data transmissions.

A simple extension to an XOR cipher is perform a bitwise rotation of the key by $n \mod 32$, where n number of the block being encrypted and $\mod$ represents a modulo operation. This ensures that each subsequent block of data is encrypted slightly differently, making the system less vulnerable to frequency analysis. However, the trivial relationship between each changed iteration of the key still leads to a vulnerable method of encryption.

There are many standard encryption algorithms known to be much more seure than the ciphers discussed. These include the Data Encryption Standard (DES) [2], Triple DES (3DES) [3], Advanced Encryption Standard (AES) [4] and Blowfish [5]. However, these algorithms can get extremely complex, and implementing any of them was considered beyond the scope of this project. The goal of the encryption algorithm was thus not to implement achieve high levels of security. Indeed, even the complex DES algorithm has been shown to crackable [10]; and even this much complexity was not within the scope of this project.

Instead, the goal was to extend the principles of the rotating XOR cipher, including using a 5-bit position value to obfuscate the key used in the XOR operation on the final data. This involved ensuring that the relationship between the key used in each position was less trivial than a simple rotation, to make it more difficult to compare data blocks at different 5-bit positions. Frequency analysis could, however, still theoretically be performed by only analysing data blocks at the same positions. To limit the harms of this, an additional goal was to make the process of converting from the primary key to the key for the specific position somewhat complex; such that decrypting data in a position does not lead directly to knowledge of the key for the entire set of data.

The final is represented below. K is the key. P is the 5-bit position. ROTL represents a bitwise rotation to the left. $\oplus$ is the bitwise XOR operator. D_{in} and D_{out} are the input and output data. x_1, x_2 and x_3 are temporary variables, used to make reading the equation more readable.

$$x_1 = P \oplus (K \& \texttt{0x1F}) \tag{1}$$

$$x_2 = ((K \oplus (x_1 << 27) \ \mathrm{ROTL}\ x_1) \tag{2}$$

$$x_3 = ({x_2}^2 \mod 4294967311) \& \texttt{0xFFFFFFFF} \tag{3}$$

$$D_{out} = D_{in} \oplus x_3 \tag{4}$$

The purpose of the operation performed on x_2 (squaring and modulo) is to make the relationship between the keys used at different positions less trivial than a simple rotation. The squaring and modulus should yield a seemingly unrelated "positional key" for each position. 4294967311 is a prime number just larger than 2^{32}. The use of a prime number ensures that there is no risk of x_2 being a factor of the number and the output of the modulo operation thus being 0 uncharacteristically often. The size was chosen such that the outputted results still utilise a full 32-bit range. After the modulo operation is performed, the bitwise AND and truncates the result back to exactly 32 bits.

Prior to the above position is modified by a bitwise XOR with the last 5 bits of the key. The pigeon-hole principle means that this will yield a unique sequence of 5-bit positions for different primary keys, that is not just $1, 2, 3, ...$ The first 5 bits of the primary key are then bitwise XORed with the new position value, and the entire output is subject to a bitwise rotation by the new position. This combination of operations makes it harder to find a primary key by reverse-engineering the encryption process from frequency analysis on a specific position. Even if squaring and modolu steps described above are mathematically reverse engineered, the relationship with the original key is not necessarily clear.

B. Golden Standard Implementation

The golden measure was implemented in Python using vectorised numpy operations. This allows for simple code, which makes it easier to ensure that the golden standard is capable of producing reliable ground-truth data for comparison to other implementations. Additionally, vectorised numpy operations are designed for efficiency, allowing good performance (within the limits of a sequential program). The basic flow of the program is as follows:

h!

- Load input data from a specified file and store said data in a numpy array of 32-bit unsigned integers
- Create an array of rotated keys, with each key corresponding to a rotation by (index%32)
- Apply the defined (basic or advanced) encryption algorithm
- The array of rotated keys is used to apply encryption to the entire array in one operation, eliminating the need for slow iterative looping
- Write the data back to a file to store encrypted data

XOR encryption is a reversible process - applying the same encryption algorithm twice will return the original data. The golden measure was validated by confirming that, upon re-applying encryption to the outputted data, the original data was returned.

C. OpenCL

The OpenCL implementation was created using C++, OpenCL 3.0 and Nvidia CUDA.

Open Computing Language (OpenCL) is a language platform for interfacing with heterogenous processors such as CPUs, GPUs and FPGAs. It allows for the development of a kernel (which uses the C99 programming language) to be initialised and queued on the hardware for performing dedicated tasks.

This implementation created a DEA on the GPU by reading data from a text file, storing the data on the heap and

encrypting it with parallelized workers. A simple and complex version of the encryption was created. The data was stored as 32 bit unsigned integers which are each encrypted by a single worker. The local size was left as NULL so that the compiler can choose the most efficient size, however, it should be noted that 4096 is the maximum work group size.

An OpenCL kernel can be setup in the following steps:

- Obtain the Device ID and platform information
- Create a context for the transfer of data
- Create a program which is the kernel
- Create a command queue which starts the kernel

Three buffers were used as arguments to the kernel, all of which were in global memory. Data, output and the key.

The data buffer contains the input data. The data itself was stored in an array created on the heap before passing it to the OpenCL buffer. Each element was a 32 bit unsigned integer so the size in bytes is $32/8 * globalSize$.

The key is a 32 bit unsigned int constant which was passed to every worker. This was the original encryption key and was not changed throughout testing.

The output buffer stored the outputs from the kernel and was the same size as the data buffer. when the buffer is read, the output data is stored in an array created on the heap.

The timing was measured from the start of the kernel setup to when the output data was retrieved. This includes the CPU overhead time, kernel overhead time and kernel execution.

D. FPGA

In order to parallelize the FPGA implementation, it was broken down into 4 sections: Top-level, Parallelizer, Encrypter(s), and Collector. The Parallelizer was responsible for receiving data from the top-level module and distributing the data among the encrypters, which were responsible for implementing the encryption algorithm. The encrypters output their encrypted data to the collector, which would sequentially collect this data and send it back to the top-level module. All modules were coded with a "write on the negative edge, read on the positive edge" communication method.

E. Parallelizer

The parallelizer's main task during the encrypter of the data is to parallelize the data stream from the host device and pass packets of data to each encrypter. It has a secondary task of programming all the encrypters with the encryption key and the rotation of the key for the data packet they are encrypting. The parallelizer was coded as a state machine to manage its flow.

Initially the device is reset, and then the key is programmed into the device which happens as follows:

- Device is reset
- The parallelizer is signalled that the key is being sent over QSPI
- The parallelizer receives 8 QSPI transactions and stores them as the key.
- The parallelizer puts the key on the encrypters data bus, and signals the encrypters to record the key.
- Once all the encrypter have acknowledged the key and are ready for data packets, the parallelizer signals that it is ready for data packets from the input source.

The data packets are also sent over QSPI to the parallelizer which then distribute it among the encrypters as follows:

- The parallelizer receives 8 QSPI transactions and stores them as the data packet.
- The parallelizer puts the data packet on the encrypters data bus and the key rotation on the encrypters key rotation bus. It then signals the first encrypter that its data is ready.
- The parallelizer then moves to the next encrypter, waits for it to be ready, increments the key rotation, and starts this process again. When it reaches the last encrypter, it circles back to the first encrypter again.

F. Encrypter

The Encrypter module (E module) is the main processing module of the DEA. Its main functions include:

- Receive a known bit width key
- Receive a known bit width data
- Receive an offset
- Rotate the key by the offset
- XOR the rotated key and data
- Output the data to the Collector Module (C module)

The E module was a stand-alone module that gets duplicated by the Parallelizer module (P module). The idea is that it does the encryption for a single set of data very fast and multiple E modules are run concurrently on the FPGA, effectively parallelizing the DEA process. Both the simple and complex version of the encryption can be achieved with minimal change to the encpypter module and other modules; only the encryption state needs to be changed.

The E module operates as follows.

- P prog_p $\uparrow$
- E grabs key from data bus
- E ready_p $\uparrow$
- P data_ready_in_p $\uparrow$
- E grabs data and offset
- E ready_p $\downarrow$
- E rotates and encrypts data
- E data_ready_out_c $\uparrow$
- C capture_c $\uparrow$
- E data_ready_out_c $\downarrow$
- E ready_p $\uparrow$
- *repeats*

The encryption process was implemented on the falling edge of the clock in simulation, which effectively achieves the encryption in one clock cycle, however, more complex algorithms and implementing on hardware may result in the encryption process taking longer, to which the parallelism of the design would be beneficial.

The E module uses a Finite State Machine (FSM) model which means that its switches between states due to events which define its behaviour. It has the following states:

- IDLE
- READING_KEY
- WAITING_RDY_DATA_IN
- READING_DATA_ENCRYPTING
- SENDING_DATA
- ACKNOWLEDGING_CAPTURE

G. Collector

The Collector module collects data from the encryptors and sends it to an external device using quad SPI.

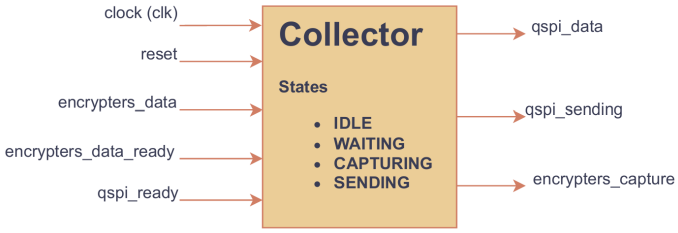

Fig. 2: representation of the collector model

The collector was coded as a state machine with the expected operation state flow below.

Command	State	Description
Starting state	IDLE	This is the module initialized state.
If in Idle state and the reset line is high on positev edge of clock.	WAITING	Waiting for the encrypter data at the corect indix to be ready.
If the encrypter data at the corect indix is high on the positive edge of the clock.	CAPTURING	The module is capturing the encrypter data.
The module enters this state from CAPTURING state on the next decending clock egde.	SENDING	Represents the state where the module is sending the captured encrypter data via QSPI.

TABLE V: Collector State Operation

V. Proposed Development Strategy

The FPGA implementation could be used in two commercial ways. It could be built into computers to free up CPU processing power or it could be designed as a standalone product that could be connected to a computer or microcontroller and used to encrypt data.

The version that is integrated into a CPU could connect directly to sections of memory, allowing it to run a 32-bit wide or wider bus and encrypt data with a set encryption method. Alternatively, it could be included in WiFi or Bluetooth chips which handle a large amount of encrypted communication and an integrated, logic gate based computational unit would improve their performance.

To make it a standalone product, it would need to be encapsulated into a small case, and would require some form of communication protocol and user interface to work. The preferred method of communication would be the widely available USB-C protocol allowing for up to 40 GBps of data, and paired with a user interface, it would be capable of offloading large amounts of work from the CPU.

VI. Planned Experiments

The FPGA comprised of multiple different units and such their internal workings and interconnectivity had to be tested. The following functionality tests were made:

- The top level module can read file data and send it over QSPI.
- The parallelizer can read data over QSPI from the top level module.
- The parallelizer can program the encrypters with a key.
- The parallelizer can send a data packet and key rotation to a selected encrypter.
- The parallelizer uses all available encrypters in a circular fashion.
- The encrypters are able to process a data packet and key rotation and output its XORed form.
- The collector can capture data from all available encrypters in a circular fashion.
- The collector can send data to the top level over QSPI.
- The top level module can receive data over QSPI and save it to a new file.

The experiment used to test the accuracy and efficiency of the three different encryption systems Verilog DEA, OpenCL DEA, and Python Golden Measure was done by:

- The accuracy of each method was assessed by comparing its outputted encrypted data with the golden measure outputted encrypted data for equivalency.
- Generating timing data for all three methods by encrypting the same data five times using each method and then recording the shortest time taken by each method. The data size was then increased exponentially and timing data was rerecorded.
- The timing data was then compared using visual methods such as speed up graphs.

VII. Results

This section details and discusses the results from the numerous tests run on the DEA

A. Results From Testbench Tests

A top level module was setup and used to test the simple XOR encryption using a small, 32 byte input string from the Star Wars movie script was used:

```
"STAR WARS\n\nEpiso"
```

This test was run with two encrypters for ease of viewing, however it was proven that any number of encrypters worked. All variable names in the top level module were suffixed with the units they connect to: *tp* is between the top level and parallelizer modules, *pe* is between the parallelizer and encrypters modules, *ec* is between the encrypters and collector modules, and *ct* is between the collector and top level modules. Finally, the clock (*clk)* and reset (*reset*) were connected to all modules. The full-sized images can be found in the Appendix in section A

Initially, the device was reset and then the parallelizer was put in program mode using (*prog_tp*). The key 0xB4352B93

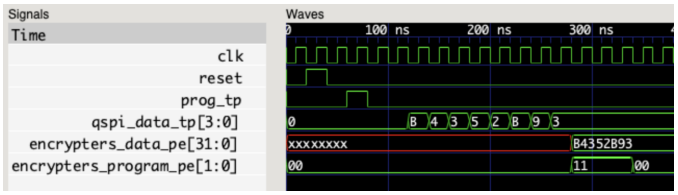

Fig. 3: Programming transaction between the top level and parallelizer modules

was sent hex by hex along the QSPI data bus (*qspi_data_tp*) and once the parallelizer obtained the full key, it was output on the parallelizer-encrypters data bus (*encrypters_data_pe*) and the parallelizer signalled all the encrypters that the key was ready using *encrypters_program_pe*.

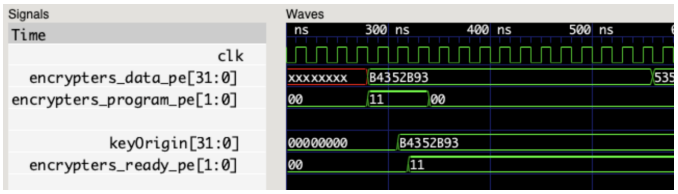

Fig. 4: Programming transaction between the parallelizer and encrypters modules

Once the key was put on the encrypters' data bus, the encrypters stored the key in a local register and signalled that they were ready for data packets using (*encrypters_ready_pe*)

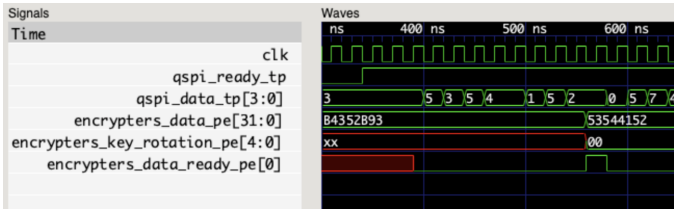

Fig. 5: Data transaction between the top level and parallelizer modules

The first packet of data 0x53, 0x54, 0x41, 0x52 which corresponded to S, T, A, R was sent to the parallelizer. Once the full packet was received, it was put on the encrypters' data bus with the key rotation which started at 0 and the first encrypter was signalled that its data was ready.

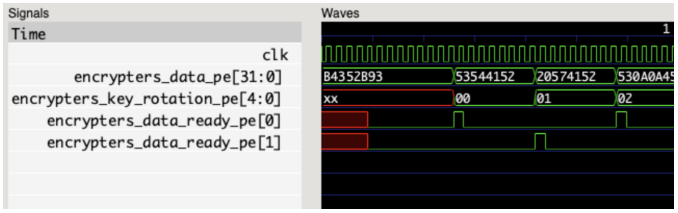

Fig. 6: Continuous data transactions between the top level and parallelizer modules

Fig. 16 shows that the data packets were sent between the different encrypters in a circular or alternating fashion.

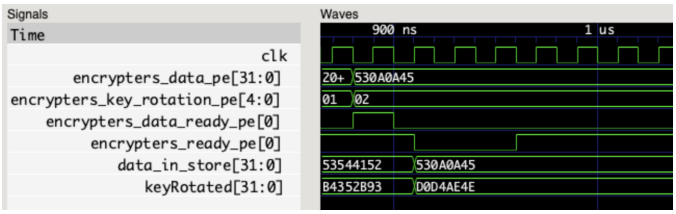

Fig. 7: Data transaction between the parallelizer and encrypters modules

Fig. 17 demonstrates how the key was rotated using the key rotation value before the data was encrypted.

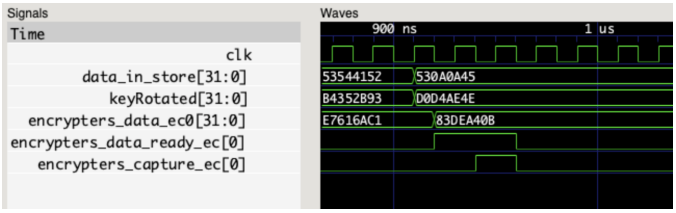

Fig. 8: Data encryption by the encrypters modules

Once the encrypter had rotated its key, it was XORed with its stored data packet and outputted to the collector (*encrypters_data_ec*). Additionally, the encrypter signalled to the collector that there was a packet ready (*encrypters_data_ready_ec*) and the collector captured the packet and signalled that data was captured using *encrypters_capture_ec*. Once the encrypters reads that its packet was captured, it dropped the *encrypters_data_ready_ec* line and readied itself for another packet from the parallelizer.

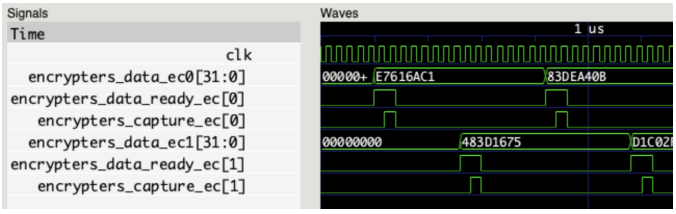

Fig. 9: Continuous encrypted data transactions between the encrypters and collector modules

Fig. 19 shows how the collector alternated between capturing data from each encrypter.

Finally, the collector sent its encrypted packet back to the top level module over the *qspi_data_ct* bus where it was read and stored into a file.

B. Accuracy results

The outputs of the two (FPGA and OpenCL) hardware-accelerated encrypter modules were compared against the golden standard. All outputted files were found to be the same, indicating successful encryption. Additionally,

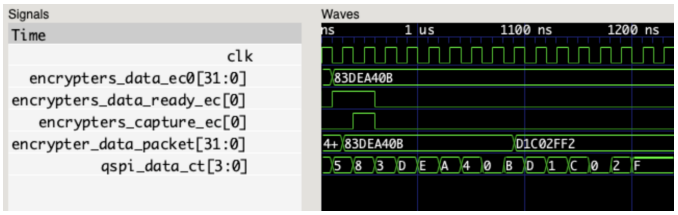

Fig. 10: Data transaction between the collector and top level modules

the files were able to be decrypted by running them through the encrypter modules again.

C. Timing results

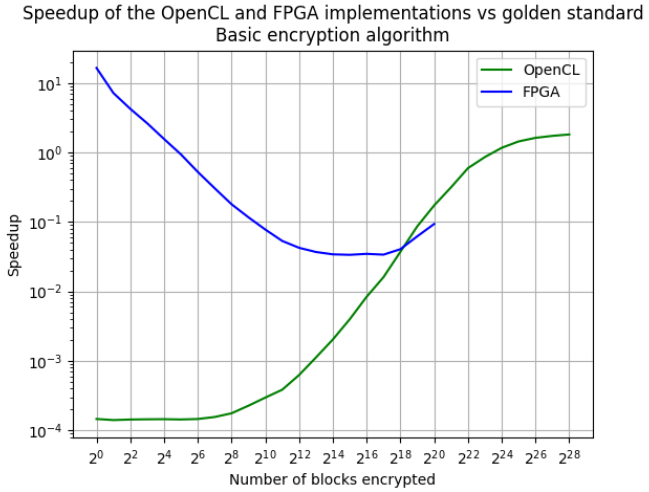

Fig. 11: Speedup results for the basic encryption algorithm

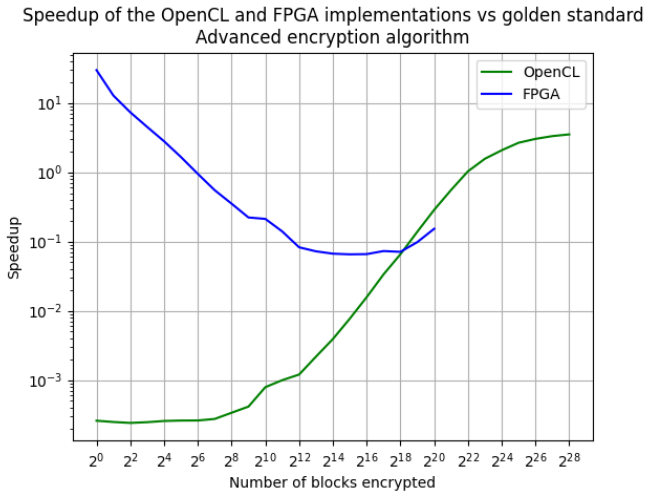

Fig. 12: Speedup results for the advanced encryption algorithm

Figures 11 and 12 show the speedup of the FPGA and OpenCL implementations as measured against the golden standard. The results for both the simple and advanced algorithms show similar trends.

The most significant limitation of the FPGA implementation is the data transfer rate of the QSPI line. Inside the module, the hardware connections and parallel implementation were found to allow the encryption of a block of data to happen within one clock cycle.

For small numbers of data blocks, the FPGA based implementation shows up to $11\times$ speedup against the golden standard. Interestingly, this speedup becomes smaller as the number of data blocks increase. A likely explanation that there is some element of overhead in the golden measure that is only applied when setting up the arrays for encryption, which becomes less significant as number of blocks increases. A limitation on this is that the computing hardware that connects to the FPGA would need to be highly efficient in sending data to and from the module on the QSPI line to be able to realise this speedup. Nevertheless, there may be an application for such a hardware module if rapid encryption of many small but separate files is required.

A limitation to the interpretation of the results for the FPGA implementation comes from the fact that it is only tested in simulation. This may make it difficult to assess some sources of delays in the implementation.

The OpenCL implementation is substantially slower than the golden standard for small numbers of data blocks, but starts to show tangible speedup for larger numbers of blocks. The maximum speedup is seen for the advanced algorithm at ≈ 4. The speedup curve seems to reach a plateau, indicating that the OpenCL implementation is reaching the limit of it's achievable speedup with the available parallel computing hardware. Although this is not necessarily an extremely large speedup, it is sufficient to make a difference to a user who may have to wait 4 times as long for their encryption to complete, e.g., a 15 minute wait instead of a 1 hour wait.

Both the FPGA and OpenCL graphs show better speedup across the board for the advanced algorithm than the simple algorithm. Testing showed that the advanced algorithm could still be implemented within one clock cycle on the FPGA, and that the OpenCL timing results were similar for both the basic and advanced algorithms. The change in speedup is thus most likely indicative of the slowdown of the golden measure when it must perform many more operations in succession for the more advanced algorithm.

Overall, the results show that the general principle behind parallel application for digital encryption can achieve tangible parallel speedup, though for the specific algorithms implemented, the added hardware expense and/or development time may not be justifiable. However, the fact that the advanced algorithm showed better speedup than the simple algorithm can be logically extended to a much more complex algorithm which may achieve much more substantial speedup. Indeed, many commonly used secure encryption algorithms require significantly more complexity than those implemented in this investigation, and could thus benefit much more substantially from hardware acceleration. The

hardware acceleration principles examined in this investigation are extensible to such an application.

D. Acceptance Test Procedure (ATP)

Rather than demoing the DEA, the following, alpha acceptance tests were conducted. The tables below list and detail the acceptance tests for the basic and advanced DEA.

Index	Tests	Description
ATP-B1	S-B1, S-B2, S-B5	Make a developer use the basic iVerilog prototype system to encrypt and decrypt data with a 32-bit rotating key.
ATP-B2	S-B1, S-B2, S-B6	Make a developer use the basic OpenCL DEA system to encrypt and decrypt data with a 32-bit rotating key.
ATP-B3	S-B4	Develop tests the golden measure Python program works as expected.

TABLE VI: ATPs for Basic DEA

Index	Tests	Description
ATP-A1	S-A1, S-A2, S-A3, S-B6	Make a developer use the advanced iVerilog prototype system to encrypt and decrypt data with a 32-bit obfuscate rotating key.
ATP-A2	S-A1, S-A2, S-A3, S-A6	Make a developer use the advanced OpenCL DEA system to encrypt and decrypt data with a 32-bit obfuscate rotating key.
ATP-A3	S-A4	Develop tests the golden measure Python program works as expected.

TABLE VII: ATPs for Advanced DEA

Below are the results from the ATPs.

- ATP-B1 : This test was passed as can be seen on the project's GitHub repository in the *data* folder here by looking at the *starwarsscript_verilog.enc*.
- ATP-B2 This test was passed as can be seen on the project's GitHub repository in the *data* folder here by looking at the *starwarsscriptOpenCLSimple.enc*.
- ATP-B3 This test was passed as can be seen on the project's GitHub repository in the *data* folder here by looking at the *starwarsscriptgoldensimple.enc*.
- ATP-A1 This test was passed as can be seen on the project's GitHub repository in the *data* folder here by looking at the *starwarsscript_verilogADV.enc*.
- ATP-A2 This test was passed as can be seen on the project's GitHub repository in the *data* folderhere by looking at the *starwarsscriptOpenCLAdvanced.enc*.
- ATP-A3 This test was passed as can be seen on the project's GitHub repository in the *data* folderhere by looking at the *starwarsscriptgoldencomplex.enc*.

VIII. Conclusion

In conclusion, utilizing the results from testing revealed that the two DEAs consistently produced the same encrypted data as the golden standard in both simple and complex versions. Some acceleration was achievable with the hardware solutions. A comparison of the two different DEA systems revealed that the FPGA DEA was faster for smaller data packages, while the OpenCL solution was faster for larger data packages. Due to the basic encryption algorithm used, the speed-up results do not justify the hardware implementations. However, if a more computationally intensive encryption algorithm is required it could be integrated into either DEAs architecture for significant speed-up results.

IX. Recommendations

Recommendations for further developments and iterations of this project are:

- Implementing the FPGA simulation code onto hardware.
- Testing single-threaded versus parallel FPGA implementations.
- The advanced DEA should be further adjusted to allow it to send any length of data without the data becoming vulnerable to frequency attacks by making every 32 32-bit packets have a different key that could be obfuscated from the previous key, the encrypted data or both.
- Testing and comparing the effects of different data bit widths for both OpenCL and FPGA implementations.
- Investigate the use of local memory on the GPU to accelerate data access speeds of individual work groups in OpenCL.

X. GitHub repository

Here is a link to the GitHub repository of this project: https://github.com/rothdu/EEE4120F-YODA.git

A. How to navigate this repository

golden-measure: Contains the golden measure implementation of an XOR encryption standard, written in Python data: Contains various files representing data to be analyzed by the encrypter(s) Icarus-Verilog-testing: Contains verilog modules for an FPGA-based implementation of the DEA, and appropriate testbenches for execution in Icarus Verilog.

OpenCL testing: The OpenCL file contains code to replicate the DEA functionality using OpenCL.

The Icarus Verilog testing: file contains the Verilog code used to run simulations of the FPGA DEA system. The subfolders inside the Icarus Verilog testing can be seen below:

- Collector: Contains code to extract data from the encrypters and transmit it out of the FPGA via QSPI.
- Constants: Contains constants that can be used to set up the functionality of the DEA such as bit width, key width a number of encrypters used.
- Encrypter: Contains the Verilog code for the basic and advanced encrypter modules.
- Parallelizer: Parallelizer reads data via QSPI breaks it up into parallel sections and then sends it to the appropriate encrypter module.
- TopLevel: Contains the Verilog code to integrate the above modules.
- GKTWave: Contains GKTwave images

The data file: Contains the input data for the tests and the test result data.

The report file: Contains the latex code used for creating the PDF report.

REFERENCES

[1] Digilent, "Nexys 4 reference manual." [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-4/reference-manual

[2] J. O. Grabbe, "The des algorithm illustrated," *Laissez Faire City Time*, vol. 2, no. 28, 2018. [Online]. Available: https://uomustansiriyah.edu.iq/media/lectures/9/9_2018_12_29!09_51_11_AM.pdf

[3] M. Y. J. Aamer Nadeem, "A performance comparison of data encryption algorithms," *International Conference on Information and Communication Technologies*, pp. 84–89, Aug 2005. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1598556

[4] A. M. Abdullah, "Advanced encryption standard (aes) algorithm to encrypt and decrypt data," June 2019. [Online]. Available: https://www.researchgate.net/profile/Ako-Abdullah/publication/317615794_Advanced_Encryption_Standard_AES_Algorithm_to_Encrypt_and_Decrypt_Data/links/59437cd8a6fdccb93ab28a48/Advanced-Encryption-Standard-AES-Algorithm-to-Encrypt-and-Decrypt-Data.pdf

[5] N. Khatri, "Blowfish algorithm," *International Journal OFEngineering Sciences Management Research*, vol. 2, no. 10, October 2015.

[6] S. H. Roosta, *Parallel processing and parallel algorithms: theory and computation*. Springer Science & Business Media, 2012, ch. Chapter 1.

[7] A. Pandit, "Introduction to fpga and it's programming tools," Apr 2019. [Online]. Available: https://circuitdigest.com/tutorial/what-is-fpga-introduction-and-programming-tools

[8] "Communication protocols in embedded systems," may 2023. [Online]. Available: https://electricalfundablog.com/communication-protocols-embedded-systems

[9] B. Gunasekaran, "Quad-spi, everything you need to know!" sep 2023. [Online]. Available: embeddedinventor.com

[10] E. Biham and A. Shamir, "Differential cryptanalysis of des-like cryptosystems," *Journal of CRYPTOLOGY*, vol. 4, pp. 3–72, 1991.

APPENDIX

A. Full-size images from RESULTS section VII

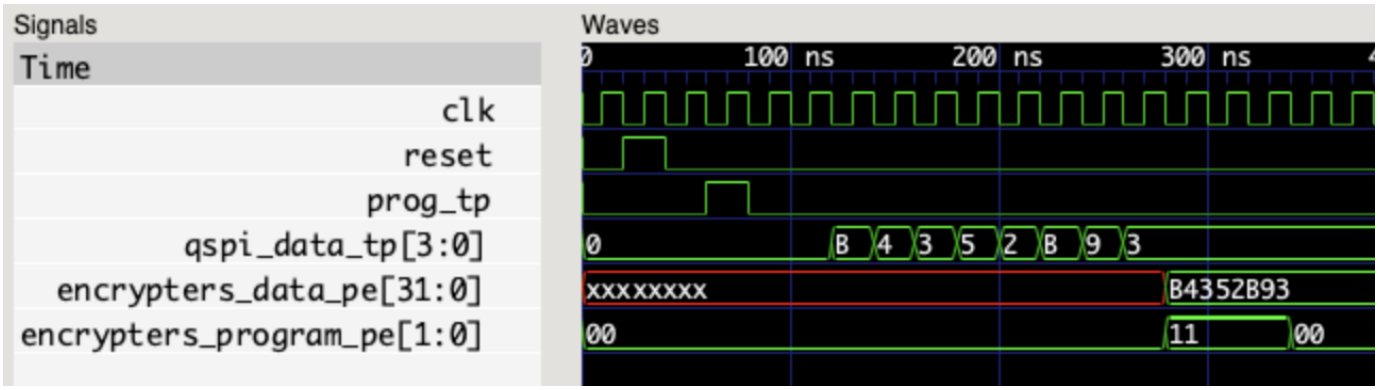

Fig. 13: Programming transaction between the top level and parallelizer modules

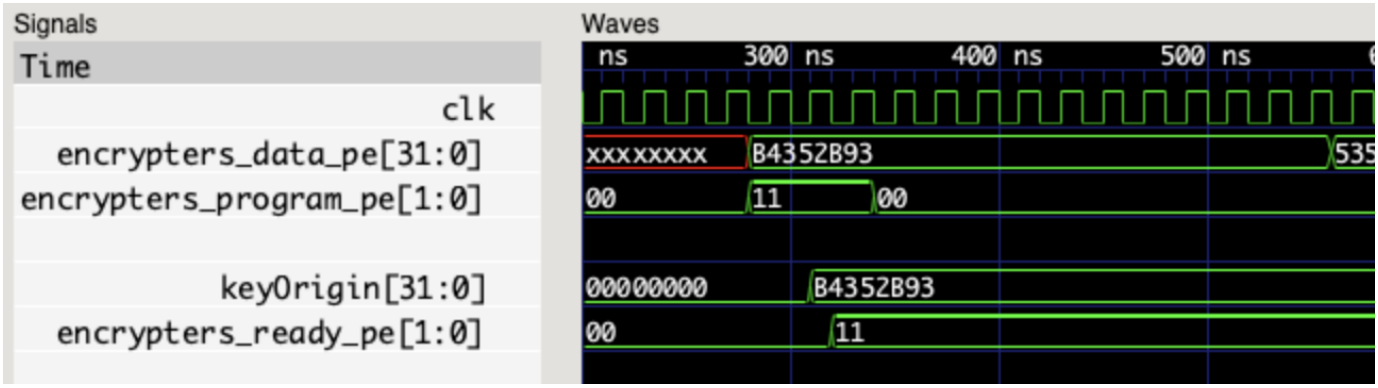

Fig. 14: Programming transaction between the parallelizer and encrypters modules

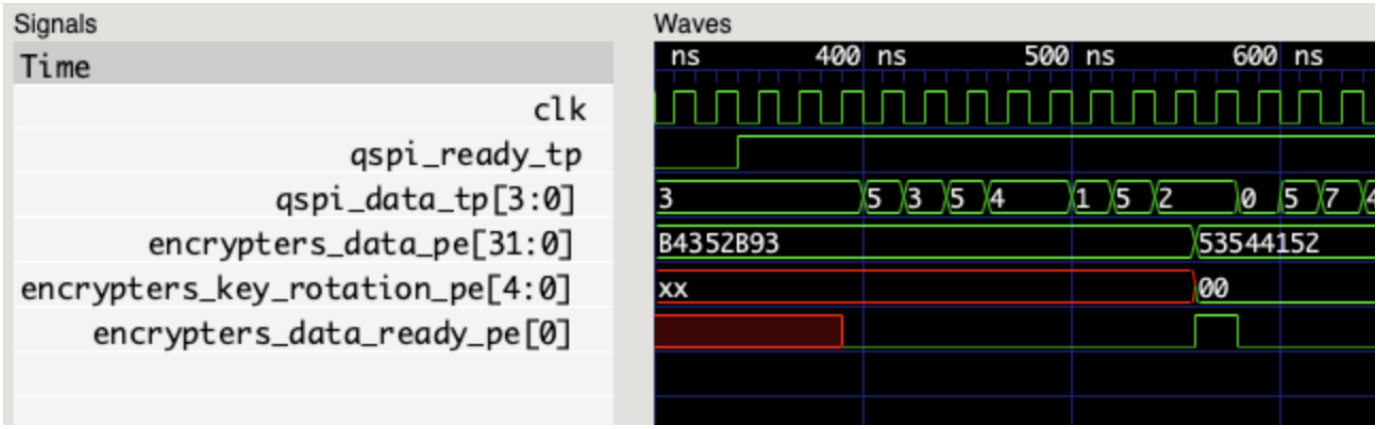

Fig. 15: Data transaction between the top level and parallelizer modules

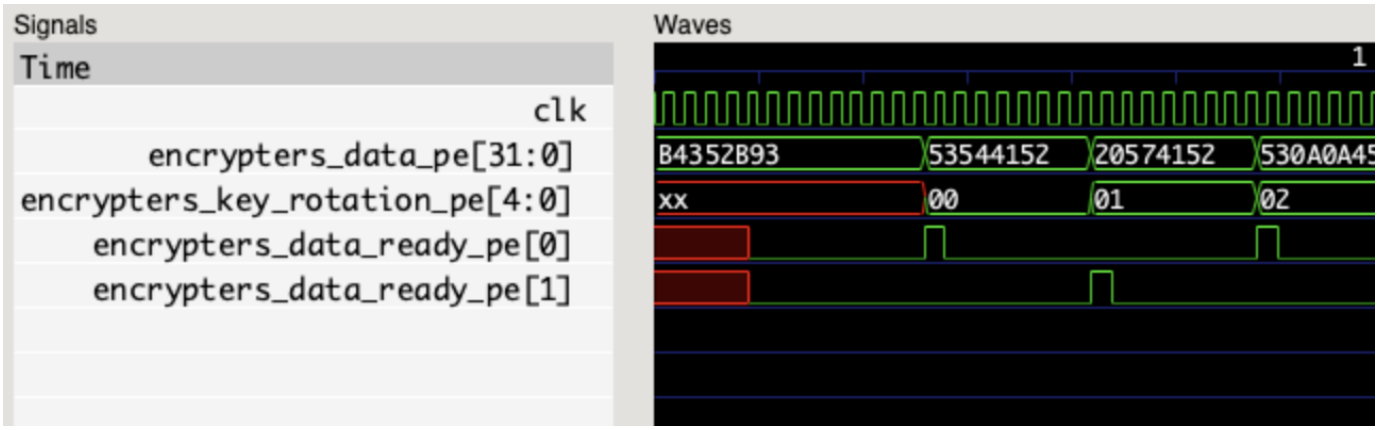

Fig. 16: Continuous data transactions between the top level and parallelizer modules

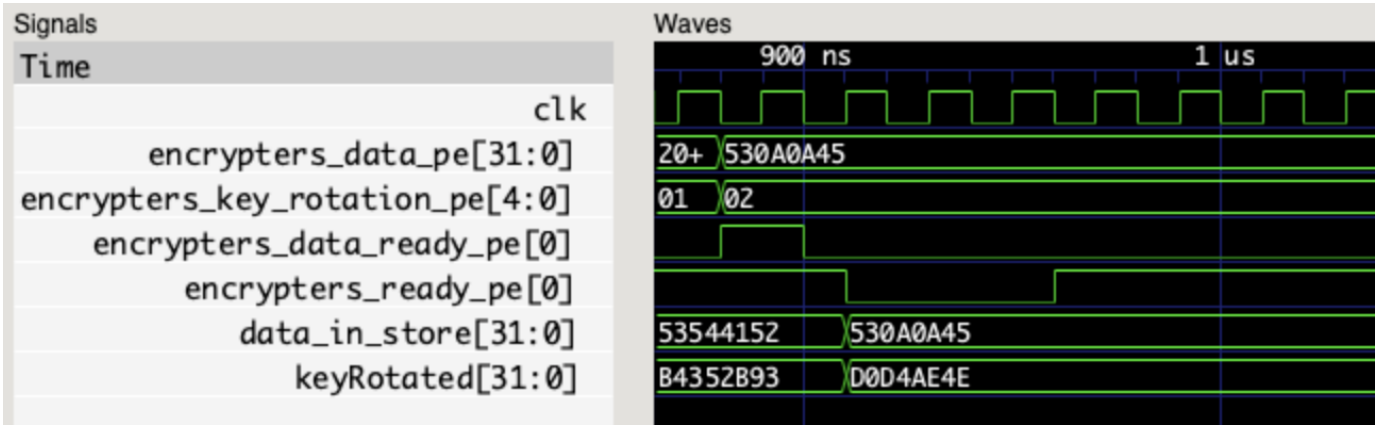

Fig. 17: Data transaction between the parallelizer and encrypters modules

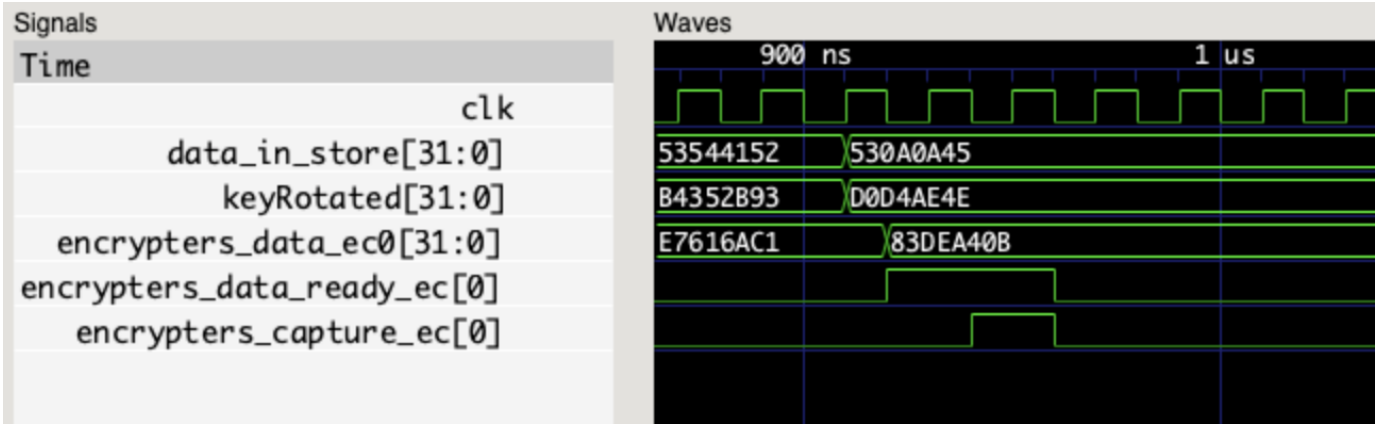

Fig. 18: Data encryption by the encrypters modules

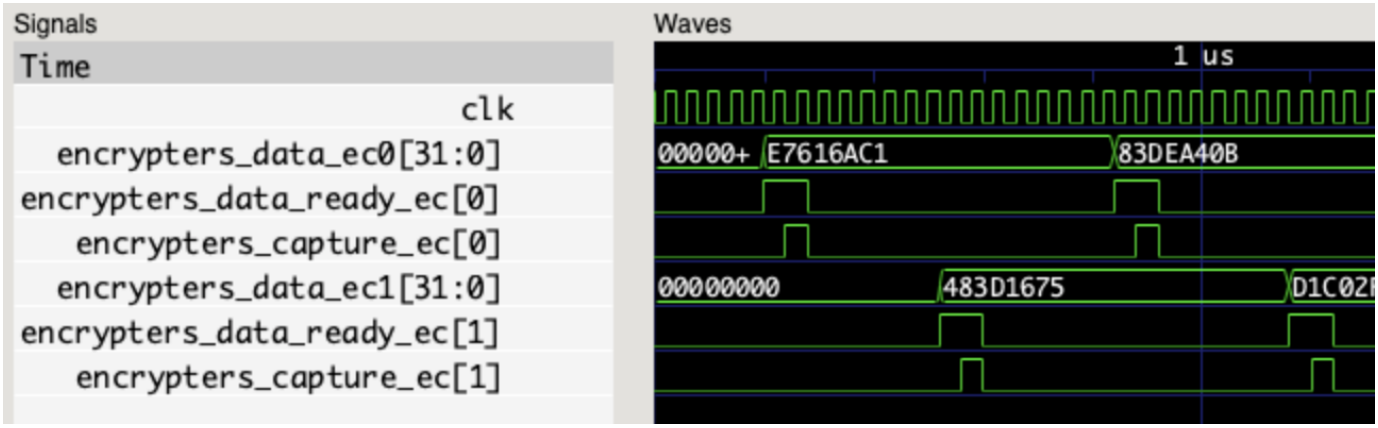

Fig. 19: Continuous encrypted data transactions between the encrypters and collector modules

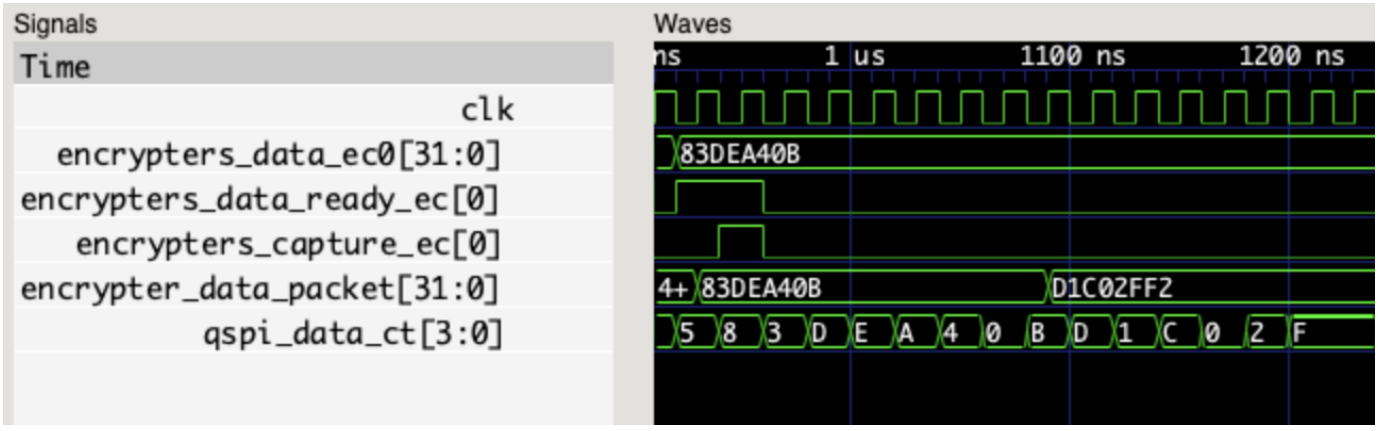

Fig. 20: Data transaction between the collector and top level modules