Encrypting Inter-FPGA Data Streams

Edwin Lee Mahdi Abbaszadeh Mohammadmahdi Mazraeli

Abstract— **This work presents the design of a hardware-accelerated encrypted data stream within a heterogeneous system. The system consists of two CPUs which perform a public-private key exchange, and two FPGAs which encrypt and decrypt the data respectively. As part of this work, AES encryption and decryption cores were developed in Vivado HLS and optimised to attain a throughput of 1.4 Gbps. This system has been validated in hardware with a single FPGA and is expected to be demonstrated on two FPGAs shortly.**

I. Introduction

Network security has long been a critical subject with far reaching implications [1]. As "smart" devices become increasingly prevalent, it has become ever more important to secure our data.

Cryptographic algorithms are a common method of protecting information and have been well studied, yielding a plethora of techniques [2]. While such algorithms must always keep cryptographic strength at their core, executing these algorithms in a computationally efficiently manner remains a challenge and is key to their adoption [3]. One approach to this problem is to accelerate these algorithms in hardware, thereby improving both performance and power efficiency [4] [5] [6]. Unfortunately, implementing cryptographic algorithms in Hardware Description Languages (HDLs) can be a challenging endeavour as such algorithms often consist of several complex rounds. High-level synthesis (HLS) tools can be used to ease this implementation by bridging high-level languages (such as C++) with the performance and efficiency of hardware.

This work uses Vivado HLS to implement the Advanced Encryption Standard (AES) on a Zynq Ultrascale+ FPGA. This is integrated with the Galapagos framework [7] and combined with a public-private key exchange to establish an encrypted data stream between two FPGAs.

II. Background

A. Advanced Encryption Standard (AES)

AES is a block cipher used to encrypt data [8]. AES is "symmetric", meaning that the same key is used for the encryption and decryption of data. The AES algorithm encrypts and decrypts data using the key to apply multiple "rounds" to the plain text. Each round consists of a mix of permutation, substitution, and bitwise arithmetic, making deciphering the encrypted data infeasible for those without the key. There are multiple variants of AES, with a range of key sizes and round requirements. One variant is "AES-128" which uses 128-bit keys to perform 10 rounds during encryption and decryption.

B. Diffie-Hellman Key Agreement Algorithm

Symmetric encryption algorithms require both the encryption and decryption to be performed using the same key. While this could be simply achieved by publicly transmitting a key (e.g. from the encryption core to the decryption core), this would present a security hole as a third party snooping the communications could also attain the key and decrypt the cipher text. One method of addressing this is the Diffie-Hellman algorithm, which allows both cores to independently generate an identical key using only public communication [9]. Diffie-Hellman operates by having both the encrypter and decrypter publicly agree on a base and modulus, then generate and share their respective "public keys". They then use each other's public key to independently calculate a final "shared secret" key. The shared secret is so called because its value is identical for both the encrypter and decrypter, but intractable for anyone else to replicate (even with perfect observation of all communication messages). This makes it ideal for symmetric encryption algorithms such as AES.

C. Galapagos

Galapagos is a framework for connecting disparate compute resources within heterogeneous data centers [7]. A Galapagos system is built from Galapagos kernels, which consist in turn of user defined compute logic wrapped in Galapagos shells. The shells facilitate all communication, exposing a simple streaming input/output interface to the user logic. The connectivity between kernels is defined using two descriptor files: the logical descriptor indicating the connectivity between all kernels (e.g. kernel #1 connects to kernels #2 and #3), and the mapping file which physically locates each kernel (e.g. kernel #1 is on FPGA #4). The shells are then linked through Galapagos, regardless of whether underlying kernels are running on an FPGA or CPU. In this way, Galapagos abstracts the low-level networking logic and interfaces away from the development of heterogeneous compute solutions.

III. Design

A. Initial Design

Our initial block diagram is shown in Fig. 1. The design uses two FPGAs to facilitate a minimal demonstration of FPGA-to-FPGA communication, connected through the network. In the diagram, each FPGA has two AES cores capable of respectively encrypting and decrypting a data stream, as well as a MicroBlaze soft-processor. The MicroBlaze handles the glue surrounding the encryption and decryption kernels - namely the communication protocols and the initial key exchange. While the communication and key exchange could be done in hardware (similar to the AES cores), using the MicroBlaze would simplify both the implementation and verification of our logic. Note that the key exchange logic is a one-time setup, unlike the encryption/decryption kernels which would continuously process a data stream, and so performance is not a high priority.

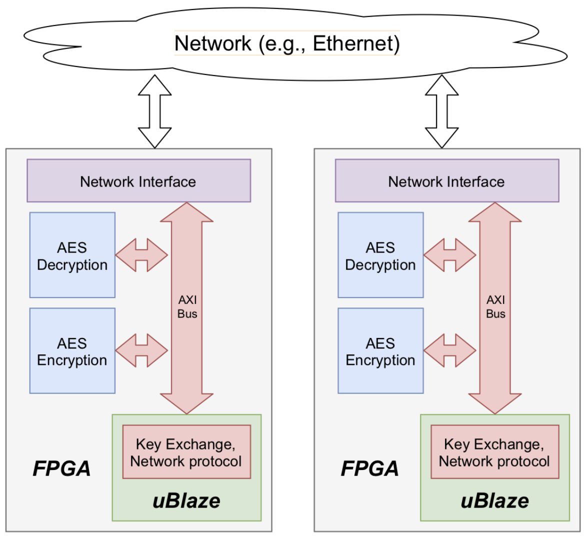

Fig. 1: Initially proposed block diagram.

B. *Final Design*

Figure 2 shows our final design. It retains a similar spirit to Fig. 1, with one FPGA encrypting data for a second FPGA to then decrypt.

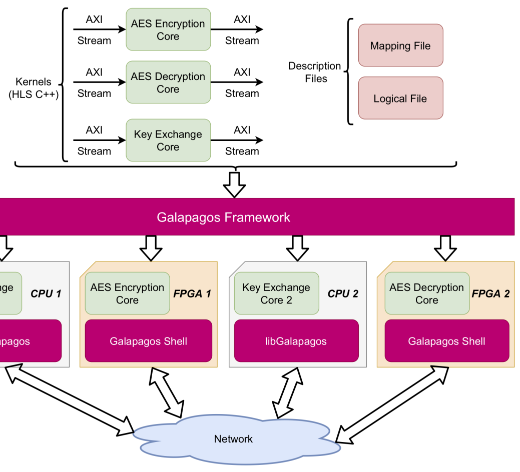

Fig. 2: Block diagram of the final proposed design, reflecting the transition to Galapagos.

However, the design now uses Galapagos for all communications (i.e. communication between the encryption core, decryption core, and the two key exchange cores). The switch was made as Galapagos eliminates the need to develop a bespoke networking stack, and as a learning exercise for the framework. Using Galapagos, the CPU and FPGA are now connected to each other using a local network and communicate using UDP.

The use of Galapagos has a few implications for the rest of the design. The first is that our kernels are not directly programmed to hardware - they are instead mapped to the hardware through Galapagos using descriptor files. The second is that our communication logic is obsoleted, with Galapagos shells around each of our kernels handling the interconnect. Thirdly, Galapagos allows the key exchange kernels to run on x86 CPUs (which would presumably also be generating the raw data stream to be encrypted by the FPGA) rather than on the MicroBlaze. When combined with the elimination of the custom communication logic, this allowed us to remove the MicroBlaze from the system entirely. The final change is not reflected in the block diagram, but concerns the interface at our kernels. As all kernels are now wrapped in Galapagos shells, they operate on Galapagos streams for the I/O.

Figure 3 shows the five-step dataflow required to establish and perform an encrypted data transfer. The first step is for the CPU-based host kernel to send three messages to prepare the system for the upcoming transaction (**1**). It provides each key exchange kernel with the other key exchange kernel's ID and provides the encryption kernel with a plain text message (which it will encrypt and transmit once it attains an encryption key). The key exchange kernels then perform a series of exchanges to generate a common key (**2**). They first establish common modulus and base values and use these values to generate their respective public keys. Each kernel then uses the other kernel's public key to calculate the shared secret, and sends this value to their corresponding AES kernel (**3**). At this point, the encryption kernel has its key, and begins encrypting all data on its input stream. This stream of cipher text is sent to the decryption kernel (**4**) where it is decrypted and sent back to the Host for validation (**5**).

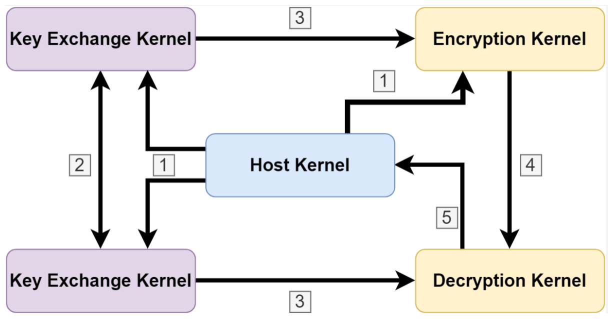

Fig. 3: Communication flow between kernels to perform an encrypted data transfer.

C. *Current Design*

At the time of writing, the system shown in Fig. 2 has yet to be fully implemented due to technical difficulties. Instead, Fig. 4 shows the block diagram of the currently implemented design. Comparing the designs, there are two main differences. The first is the absence of Descriptor files

- due to issues mapping the kernels to hardware through the Galapagos descriptor files, kernels were manually converted to Galapagos shells. To do this, the AES cores were compiled within the Galapagos shell while the key exchange kernels similarly used libGalapagos as a wrapper.

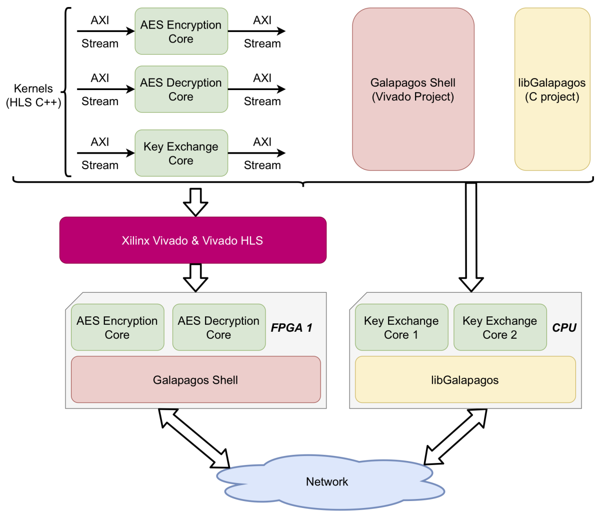

Fig. 4: Block diagram of the current design.

The second difference is in the collocation of the encryption and decryption kernels within the same FPGA. This is required due to the system's inability to properly transmit a packet from FPGA. This seems to be due to an issue in the Galapagos UDP bridge. This IP is responsible for applying the appropriate headers to packets, and is the final operation for packets before they are sent to the network (using Gulf-Stream). Presently, it seems this IP places a size of 0 on all our packets leaving the FPGA, which causes them to be dropped. By collocating the two AES kernels on the same FPGA, almost the entire dataflow shown in Fig. 3 can be verified, with only the final FPGA-CPU packet (step **5** in the diagram) being dropped. Debugging the Galapagos UDP bridge to these 0-size packets is the primary remaining work, as understanding this issue should enable the rapid deployment of the target two FPGA system.

IV. Design Methodology

Our goal was to implement the AES and key exchange kernels on a heterogeneous platform benefiting from the Galapagos framework. As a result, we developed our AES kernels using Vivado HLS and generated the corresponding IP cores. In the next step, we used the Galapagos shell, a Vivado project, to set up our system. We imported the generated AES IP cores to the Galapagos project and set up the networking configuration. On the other hand, we developed the host kernel and key exchange cores using the libGalapagos library to set up the entire system. By including the libGalapagos library, we compiled the software codes using the g++ compiler. Finally, we generated the bitstream in Vivado to program the FPGAs.

TABLE I: Throughput comparison of the developed AES cores in HLS vs. existing cores developed using HDL.

	This Work	**[10]**	**[11]**	**[12]**	**[13]**	**[14]**
Throughput (Gbps)	1.4	0.36	1.19	1.33	4.34	34.1

TABLE II: Resource utilization of each AES cores and the full-system including the Galapagos shell.

	LUT (%)	**FF (%)**	**BRAM (%)**	**DSP (%)**
AES-enc	0.39	0.23	0.4	0
AES-dec	0.42	0.19	0.5	0
Full-system	9.71	10.53	95.8	0

A. HLS cores

1) Key Exchange Cores: The key exchange cores perform a Diffie-Hellman key exchange, with cores acting as either the "receiver" or the "transmitter" in the exchange. Development began in a system model created in Python to solidify the communication protocol and interactions between all actors. This protocol was then translated into a single processing core using C++, which chooses to either be the receiver or the sender at run time. While these cores were intended to run in software in both the initial and final system design, the option of running these cores in hardware was raised early in the project in order to create a more homogeneous system. As a result, the final C++ cores were developed in Vivado HLS and operate with just an input and output data stream, which made them highly compatible with the eventual transition to use Galapagos.

2) AES Cores: The AES encryption and decryption cores implement AES-128 and were first prototyped in software using C++. They were then ported to hardware using Vivado HLS and optimised for throughput using HLS pragmas to tune their pipelining, unrolling, and partitioning.

Table I shows the throughput of the final design, which is comparable with existing AES implementations developed using HDLs. Table II shows the resource utilization of each individual AES IP core, as well as the utilisation of the complete system as a percentage of the chip (xczu19eg-ffvc1760-2-i). Note that BRAMs are fully utilized in the full-system as we have imported eight ILAs within the project for debugging purposes, and these could be omitted in a complete design.

B. Verification Methodology

As described in section IV-A, both the AES and key exchange kernels were initially developed in software. To verify these prototypes, we developed an AES testbench in C++ using hardcoded keys and two key exchange testbenches in Python using fixed base and modulus values. This enabled the independent verification of each IP's core functionality. After their conversion to hardware in Vivado HLS, this validation was repeated for both the AES cores and the key exchange cores using C++ based testbenches, run in CSim and CoSim.

Our next step was to verify the overall system, including the AES and key exchange kernels, within the context of

Galapagos. As an intermediate step, we created the full system in software using libGalapagos. We defined the kernels in the Galapagos environment and tested all communication between the cores and the host using the libGalapagos library.

With the high level system design validated, we transitioned our design to silicon for actual operation. Our cores were imported to the Galapagos project, and ILAs were used to observe each core's operation in hardware. These ILAs also allowed us to validate the connectivity and ensure no data was lost between the cores by sending debug packets from the host kernel and observing them using both Wireshark and these ILAs. In order to validate our configuration of Gulf-Stream (a UDP bridge inside the Galapagos project), we created a small Gulf-Stream project with all other logic stripped away. We then used a simple Python script to send UDP packets to the FPGA and verified the FPGA's connectivity using arping.

After validating all these pieces of the design, we finally validated the complete system by snooping the transactions in Wireshark, observing the transactions shown in Fig. 3.

C. Design Environment

Our target hardware platform was Xilinx Sidewinder boards containing Zynq UltraScale+ FPGAs. As a result, we used the Vivado 2019.1 design suite (including its HLS tool) to create our project and IP cores. We also leveraged the Galapagos framework for our CPU-FPGA and FPGA-FPGA communication via the network. To wrap our cores in Galapagos shells, our AES IP cores were implemented on top of the Galapagos Vivado project while our key exchange kernels used the libGalapagos library. We used GitHub for version control throughout the development process.

D. Partitioning

We partitioned our design into three major sections: AES development (Mohammadmahdi), key exchange development (Edwin), and Galapagos configuration (Mahdi). This partitioning was chosen as all three parts were independent of each other and could therefore be developed in parallel. Configuring Galapagos proved to be the long pole and so all members eventually transitioned to help debug and solve communication issues.

V. Project Timeline

Table III shows the project timeline. Developing, testing, and verifying our AES and key-exchange kernels for Vivado HLS took just three weeks. The majority of the development time was spent configuring the automated version of Galapagos. As part of this, three weeks were spent setting up the full system in software using libGalapagos, with the hope of directly translating this system to hardware through Galapagos. To configure and test the network communication, we created a separate Vivado project including Gulf-Stream. Our system was finally integrated and presented in November.

TABLE III: Project Timeline

	Start	**End**	**Duration (weeks)**
AES software and HLS implementations	Apr. 1st	Apr. 24th	3
Key Exchange software and HLS implementations	Apr. 1st	Apr. 24th	3
Configuring the automated Galapagos	Mar. 1st	Aug. 31st	22
Setting up the full-system in software using libGalapagos	Aug. 1st	Aug. 31st	4
Configuring the CPU-FPGA network communication	Oct. 1st	Oct. 31st	4
Integrating the full-system using the Galapagos project	Sep. 1st	Nov. 13th	10

VI. Problems

A. Galapagos Teething Issues

We initially tried to use Galapagos' automated flow, with the understanding that Galapagos would be able to map our kernels to hardware through our descriptor files. Only after investing considerable effort in this flow did we realise that Galapagos is not yet fully automated and requires manual bitstream generation. In August, we were advised to implement the full system in software using libGalapagos, as the automated flow would be ready imminently and directly support generation from a libGalapagos system. While we successfully verified our full system in libGalapagos, our kernels required adaptations for usage in libGalapagos (i.e. they were not directly transferable between libGalapagos and Galapagos) and ultimately we were never able to use the automatic flow.

B. Tools and Development Environment Setup

When setting up our development environment, we first tried to install Galapagos on our personal machines using Docker. Unsolvable issues led us to use the course "dev machines" instead. We then tried to work on the default "HCAL_HLS4ML" project, only to find the dev machines lacked necessary packages to compile the project. After unsuccessful attempts to solve these dependencies with the head-TA, we were asked to switch to another machine, "agent7", which had previously compiled large Galapagos projects successfully.

With this machine we were able to develop our Galapagos kernels and validate the system in libGalapagos. We then began testing the full Galapagos system, but found we were unable to receive packets sent from the CPU to the FPGA. Initially, we thought there was an issue with our hardware project, prompting a top-to-bottom hunt to understand the issue. After debugging our kernels and using ILAs to trace the operation of the Galapagos shell, we found out that the FPGA could not get any packets at all. We dug into this further by tracing the outgoing packets in our software kernels, and even physically verifying that the agent was connected to the data plane network in the server room. The issue was only found when we started using Wireshark and the "arping" command, which revealed that agent7 was not connected to the data plane despite being physically connected to it. This appeared to be the result of another project which had been running on agent7 as well, forcing us to relocate one more time to agent1, with which we finally validated our system.

C. Zero-Size Packet Headers

After resolving the aforementioned communication issues, we found that the FPGA was still unable to send appropriate packets to the CPU. After observing the packets being sent by the FPGA in Wireshark, we found that all the packets coming from the FGPA had a packet size of zero. The packets were otherwise intact, with the full data payload present. This issue has yet to be resolved and will require a deeper understanding of the Galapagos UDP bridge. We plan to continue debugging this issue in our future work.

D. Faulty Modulo Operation

Another unresolved issue was observed when running the key exchange IP cores in hardware using Vivado HLS[1]. As discussed in Sect. III, we have two of these IP cores in the system: a sender core and a receiver core, with the distinction only being made at run time. When running these cores on the FPGA within the Galapagos project, we found out the sender kernel works flawlessly while the receiver kernel generates an invalid shared secret. Using ILAs to further debug the cores, we found that the modulus operator was placing random data in the MSB portion of the modulo result, invalidating the calculation. This issue was not found in CoSim and could not be resolved.

VII. Contributions

A. Mahdi

Debugged the Galapagos framework to use the automated flow. Attended several meetings with head-TA to find out how to fix issues. Contributed in presenting the project and writing the report.

B. Edwin

Developed the key-exchange cores, spanning the design of the protocol's proof-of-concept in Python, the development of the Galapagos-compatible core using C++ in Vivado HLS, and the verification of the above in Python/CSim/CoSim. Planned the overall system layout and defined the interface/communication protocol between cores. Helped debug communication issues during final system integration and developed workarounds in the design. Primarily responsible for laying out and editing the reports and presentations.

[1]Our initial and final designs have the key-exchange running in software; this was simply an experiment during development.

C. Mohammadmahdi

Developed and tested the AES encryption and decryption cores in software. Ported and optimized the AES kernels within Vivado HLS. Modified and tested the AES and RSA kernels to set up the whole system in software using the libGalapagos. Integrated the AES kernel IP cores generated by Vivado HLS to the Galapagos project and resolved the issues using the ILAs. Configured the CPU-FPGA communication by creating a simple Vivado project using only the Gulf-Stream. Worked on full-system integration and testing it using the ILAs and Wireshark. Contributed in presenting the project and writing the report.

VIII. Retrospective

A. HLS vs HDL

Throughout the project, we used HLS to develop the AES and key exchange kernels. The primary advantage we attained using HLS (compared to an HDL such as Verilog) was our ability to have much faster design iteration. During the implementation of our AES core, we were able to quickly compare different versions of the core just by modifying the high-level source code. Moreover, HLS presented many switches to tune the design, such as pragmas and timing constraints. This enabled us to compare design choices when optimising our design for throughput. HLS also proved useful when verifying and debugging our design, if not only because there were fewer lines of code to inspect. In particular, CSim allowed for earlier and faster testing of our design, and the direct reuse of our testbenches for CoSim provided sanity checks for the generated HDL.

The main downside we experienced as developers was the learning curve of using HLS. HLS tools are sensitive to the way of coding, and it was often difficult beginners to know how our code would be synthesized. Further, while CSim and CoSim makes it much easier to verify the cores during development, HLS also presents a black box when trying to debug issues on the chip (such as the faulty modulo error).

Overall, we found that HLS made the development process easier and boosted our design productivity despite the expertise required to use it.

B. If We Could Do It Again

Table III shows that we spent 22 weeks dealing with the automated version of the Galapagos - time which could have been better spent. If we were to do the project again, the first change would be to not attempt the automatic flow at all, starting instead with the Galapagos Vivado project and libGalapagos library. Secondly, we should have requested help from other TAs and members in Professor Chow's research group sooner. This would have enabled us to find and resolve several teething issues much sooner. For example, we would have saved a month by finding the issue with agent7's data plane connection sooner, instead of debugging our software and hardware projects. Thirdly, we should have employed networking debugging frameworks sooner, as most of our UDP packet issues were quickly resolved once we familiarised ourselves with Wireshark. These changes would have saved months of development and debugging effort from our project.

IX. Conclusion

We have designed a system for encrypting a data stream between two FPGAs. Three IP cores were developed in C++ to enable this: a key exchange core, an AES encryption core, and an AES decryption core. The AES cores were optimised for throughput in Vivado HLS, ultimately processing data at **1.4 Gbps** which is comparable to existing work done with HDLs. Our cores were then stitched together using the Galapagos framework, with the key exchange cores running on x86 CPUs while the encryption and decryption cores ran on FPGAs. While the entire design was verified in software using libGalapagos, only a single-FPGA solution was implemented due to issues transmitting packets off of the FPGA. Further study of the UDP bridge will enable the deployment of multi-FPGA designs in hardware.

References

[1] B. Daya, "Network security: History, importance, and future," *University of Florida Department of Electrical and Computer Engineering*, vol. 4, 2013.

[2] E. Thambiraja, G. Ramesh, and D. R. Umarani, "A survey on various most common encryption techniques," *International journal of advanced research in computer science and software engineering*, vol. 2, no. 7, 2012.

[3] A. Nadeem and M. Y. Javed, "A performance comparison of data encryption algorithms," in *2005 international Conference on information and communication technologies*, pp. 84–89, IEEE, 2005.

[4] A. M. Deshpande, M. S. Deshpande, and D. N. Kayatanavar, "Fpga implementation of aes encryption and decryption," in *2009 International Conference on Control, Automation, Communication and Energy Conservation*, pp. 1–6, 2009.

[5] A. M. Borkar, R. V. Kshirsagar, and M. V. Vyawahare, "Fpga implementation of aes algorithm," in *2011 3rd International Conference on Electronics Computer Technology*, vol. 3, pp. 401–405, 2011.

[6] P. Chodowiec and K. Gaj, "Very compact fpga implementation of the aes algorithm," in *Cryptographic Hardware and Embedded Systems - CHES 2003* (C. D. Walter, Ç. K. Koç, and C. Paar, eds.), (Berlin, Heidelberg), pp. 319–333, Springer Berlin Heidelberg, 2003.

[7] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network fpga clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 237–246, ACM, 2017.

[8] W. Stallings, "Cryptography and network security: Principles and practice," Harlow: Pearson Education, 2017.

[9] E. Rescorla *et al.*, "Diffie-hellman key agreement method," 1999.

[10] P. B. Ghewari, M. Jaymala, K. Patil, and A. B. Chougule, "Efficient hardware design and implementation of aes cryptosystem," in *International Journal of Engineering Science and Technology*, pp. 213–219, 2010.

[11] T. Hoang and V. L. Nguyen, "An efficient fpga implementation of the advanced encryption standard algorithm," *2012 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future*, pp. 1–4, 2012.

[12] M. E. Maraghy, S. Hesham, and M. A. A. E. Ghany, "Real-time efficient fpga implementation of aes algorithm," *2013 IEEE International SOC Conference*, pp. 203–208, 2013.

[13] M. H. Rais, S. M. Qasim, and S. Arabia, "Efficient hardware realization of advanced encryption standard algorithm using virtex-5 fpga," 2009.

[14] S. M. Soliman, B. Magdy, and M. A. A. El-Ghany, "Efficient implementation of the aes algorithm for security applications," *2016 29th IEEE International System-on-Chip Conference (SOCC)*, pp. 206–210, 2016.