İYTE
Department of Electrical & Electronics Engineering
EE499 CoOp Midterm Report

Name & Last Name: Harun Durmuş

School Number: 270206025

Company Name: TTTech Auto Turkey Yazılım A.Ş.

TTTech Auto, founded in Vienna in 1998 by Hermann Kopetz, Georg Kopetz, and Stefan Poledna, is a leading global company dedicated to improving the safety and reliability of future vehicle generations through cutting-edge software and hardware solutions. The company's purpose, which began with the development of dependable computer systems that provide real-time safety, has since extended to include automotive, aerospace, and chip development. TTTech Auto has its primary headquarters in Vienna and additional locations in France, South Korea, Germany, Serbia, Croatia, Spain, and the United States, employing over 1200 people worldwide.

Department Name in The Company:

I am in TCM department which gives engineering services to collaborator companies in TTTech Auto Turkey. Aim of the department is to contuine their ongoing projects while training their interns by considering the principles of the company.

Supervisor in the Company
Name & Last Name: Hüseyin Karaçalı
E-mail: huseyin.karacali@tttech-auto.com
Phone: +90-(541)-527-02-06

Project Title: Performance Analysis of Linux Inter-Process Communication with "Named Pipe" by Using Different Encryption Methods

Definition of The Project: This research investigates the efficiency of Linux IPC technologies, namely named pipes, utilizing various encryption methods such as Ceasar Cipher, RSA, DES, AES-128, AES-192 and AES-256. The study examines how different encryption algorithms affect the performance of named pipes in Linux IPC, including data throughput, latency, and resource utilization. The study's goal is to provide insights on enhancing secure IPC in Linux settings while addressing both security requirements and performance implications.

Weekly Work Plan:

- **Week 1-4:** Research and Understanding the Principles of Linux IPC Technique and Encryption Methods
- **Week 4-8:** Evaluating Encryption Methods in C via Named Pipe IPC
- **Week 8-12:** Performance Analysis of Each Encryption Methods Through Named Pipe

Project Details: The project aims to improve secure Inter-Process Communication (IPC) in Linux by implementing and testing various encryption algorithms in the context of named pipes. Named pipes, often known as FIFOs (First-on, First-Out), provide a means of process communication via a specific file on the filesystem. The project digs into the technical details of key cryptographic algorithms like Caesar Cipher, RSA, DES, and AES in C, using the OpenSSL library for cryptographic operations. An important part of the research is the use of encryption within named pipes to safeguard data flow between processes. Performance parameters, such as encryption/decryption time and CPU use, are painstakingly examined to determine the computational overhead introduced by certain encryption algorithms. The project also assesses each algorithm's practical security efficacy, taking into account parameters such as key size and encryption mode. By investigating both performance and security, the project hopes to provide a thorough understanding of the trade-offs involved in employing various encryption methods for IPC in Linux, ultimately leading to the creation of more safe and efficient IPC protocols.

Conclusion: From the results, it is observed that, while performing encryption methods in different terminals, expected results and differences can be clearly observed. But using an automation method such as bash script to perform executions several times, the time and CPU utilization results become close to each other. Detailed conclusion is at the end of the report.
Notes: This is a performance analysis based project which evaluates different encryption methods through named pipe IPC.

Table of Contents

Table of Figures

Table of Appendices

Performance Analysis of Linux Inter-Process Communication with "Named Pipe" by Using Different Encryption Methods

Harun Durmuş (270206025)

Abstract—This research investigates the efficiency of Linux IPC technologies, namely named pipes, utilizing various encryption methods such as Ceasar Cipher, RSA, DES, AES-128, AES-192 and AES-256. The study examines how different encryption algorithms affect the performance of named pipes in Linux IPC, including latency, resource utilization and security. The study's goal is to provide insights on enhancing secure IPC in Linux settings while addressing both security requirements and performance implications.

I. INTRODUCTION

A. Inter-Process Communication (IPC) with Linux

IPC is a mechanism using which two or more process running on the same or different machines that exchange their personal data with each other. Different machines refers to processes running on separate computers or devices that communicate with one another, rather than processes running on the same system. This demonstrates IPS's capacity to enable communication across a network, connecting systems in different locations. Processes running on same machine, often need to exchange data with each other in order to implement some functionality. Linux OS provides several mechanisms using which user space processes can carry out communication with each other, each mechanism has its own pros and cons.

A.1. Computer Architecture

OS system interacts with hardware layer with device drivers that actually train the OS regarding how to communicate with hardware. For every hardware, there has to be associated device driver.

In Linux, OS system and applications interacts with each other with:

- Netlink sockets
- Input/Output Control System Calls (IOCTLs)
- Device files
- System calls

by using these types of communication channels.

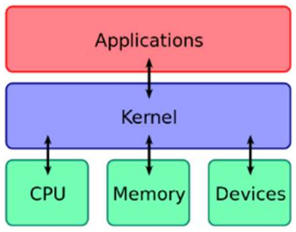


Figure 1: *An oversimplification of how a kernel connects application software to the hardware of a computer*

For example, by using the system calls such as *malloc(.)*, *free(.)* or *dalloc(.)*, application requests OS (Kernel space) to allocate or deallocate the memory. Communication between two applications running in application layer (user space) is termed as IPC.

A.2. IPC Techniques

- ***Using Unix Domain Sockets:*** They provide a technique for IPC on the same host that makes use of the file system namespace and allows processes to communicate via stream or datagram. They are more efficient than network sockets for local IPC because they do not incur network protocol overhead.
- ***Using Network Sockets:*** They promote network communication by allowing processes on multiple machines to exchange data through protocols such as TCP/IP. They're critical for distributed systems since they provide a consistent approach to connect applications across different networked settings.
- ***Message Queues:*** They allow processes to communicate asynchronously by sending and receiving messages via queues, allowing for complicated messaging patterns and giving a mechanism to prevent blocking operations, as opposed to direct communication strategies.
- ***Shared Memory:*** A way of exchanging information in which many processes access the same region of physical memory, allowing for high-speed communication while eliminating data copying. Concurrent access requires synchronization techniques.

- ***Signals:*** Lightweight, primitive IPC enables processes to transmit minimal notifications to one another. Signals are used to indicate events such as interrupts, termination requests, and minor status changes.
- ***Pipes:*** They provide a one-way communication channel between the output of one process and the input of another. Pipes are straightforward to use, but they only support a parent-child process relationship unless named pipes (FIFOs) are utilized for broader IPC.

A.3. Named Pipes in IPC

A named pipe, often known as FIFO (First-In, First-Out), is a Linux IPC technique that allows processes to communicate through a specific file in the filesystem. This file-based technique allows for sequential data transmission between processes, ensuring that data is read in the same order it was written. Named pipes enable communication across unrelated processes, providing a simple yet effective approach for data transmission that avoids the complications of direct memory access and synchronization required by shared memory systems. Named pipes are an expansion of the basic pipe notion in Linux. Named pipes, like conventional pipes, are used to convey data across linked processes; however, named pipes can be accessed using a name rather than a file descriptor.

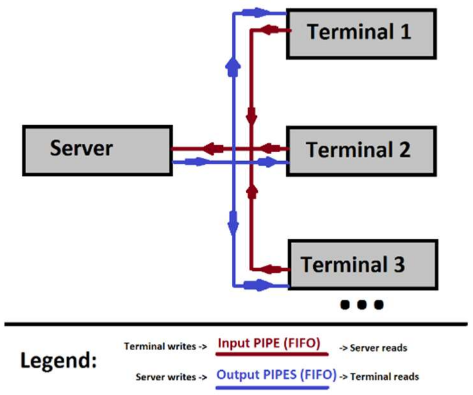


Figure 2: *FIFOs in Linux*[1]

Named pipes[2] are useful when data needs to be transferred reliably and synchronously across processes. They can be used to transfer data between processes running on the same machine or between machines connected via a network. Named pipes are also handy for communicating with programs that don't use regular input/output streams or sockets.

There are few key differences between named pipe and pipe such as:

- Named pipes are accessible by name rather than file descriptor. This enables processes that are unrelated to one other to communicate over a common named pipe.
- Multiple processes can open named pipes in either reading or writing mode. This enables processes to communicate in both directions.
- Named pipes are persistent, which means they remain in the file system long after all processes using them have completed. This enables processes to communicate with one another at varying times.

Named pipes[2], or FIFOs, are used to enable inter-process communication (IPC) capabilities. Unlike conventional files, they do not keep data on disk indefinitely, but rather permit communication between programs. They follow the first-in, first-out principle, which ensures that data is read in the same order it was written.

To understand how named pipes work, consider a scenario in which two processes must interact with one another. We'll refer to them as Process 0 and 1. Process 0 wants to communicate data to Process 1. It transfers data to a named pipe, often known as the "in" pipe in this context. The data supplied to the "in" FIFO is usually destroyed, with the exception of the byte count, which may be retained for tracking by the receiving process. Meanwhile, Process 1 reads from another named pipe, also known as the "out" pipe. When Process 1 reads from the "out" FIFO, it receives the whole amount of data accumulated thus far, allowing communication between the two processes.

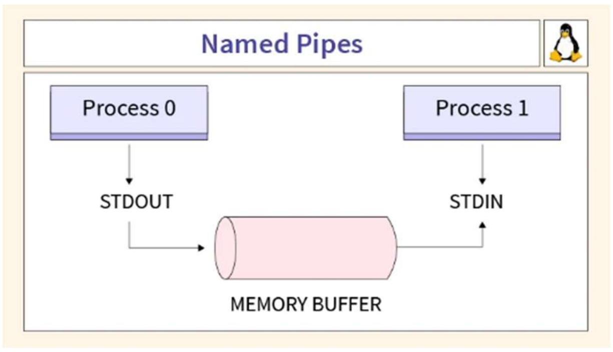


Figure 3: *Example of named pipes in Linux*

To create named pipes[3], system calls are used such as "*mkfifo*" or "*mknod*". These are system calls used in Unix-like operating systems to create special files, but they have slightly different functions. *"mkfifo"* generates named pipes (FIFOs), which provide a simple technique to support IPC without requiring direct file data storage. *"mknod"*, on the other hand, is more versatile, with the ability to create a variety of unique files, including device files, named pipes, and others, based on the flags and parameters passed in. While *"mkfifo"* is limited to named pipes, *"mknod"* requires more specific inputs to identify the file type being generated.

These calls generate a custom file or file system node that functions as a FIFO. The system call accepts arguments such as pathname, mode, and device information, with pathname identifying the FIFO's location in the file system.

One significant advantage of named pipes is their ability to provide bidirectional communication. Unlike regular pipes, which are typically used for one-way communication, named pipes can handle communication in both ways at the same time. This property qualifies them for scenarios in which processes must share data bidirectionally, such as client-server communications.

Named pipes run effectively in memory, reducing the expense of disk I/O by storing data in memory buffers rather than writing it to disk. Additionally, the kernel controls synchronization between read and write operations on named pipes, ensuring that processes communicate smoothly.

Despite their advantages, named pipes have constraints similar to normal files. Processes that communicate via named pipes must first agree on the pipe's name and file system placement. Furthermore, many processes sharing the same named pipe may interfere with one another, demanding careful coordination to avoid conflicts.

To summarize, named pipes provide a simple and efficient mechanism for inter-process communication in Unix-like operating systems. They provide bidirectional communication capabilities, efficient memory consumption, and kernel-managed synchronization, making them an effective tool for establishing communication channels between unrelated processes.

B. *Information Security*

Information security[4] encompasses the techniques and processes that secure digital and non-digital information from unauthorized access, use, disclosure, disruption, alteration, or destruction. It includes a wide range of tactics for defending against cyber threats and protecting sensitive data. The goal of information security is to ensure data confidentiality, integrity, and availability, ensuring that information is accessible to authorized users when needed while also protecting against breaches or data loss. This field is crucial for controlling the risks connected with information technology and communications in a variety of industries, including government, banking, healthcare, and more.

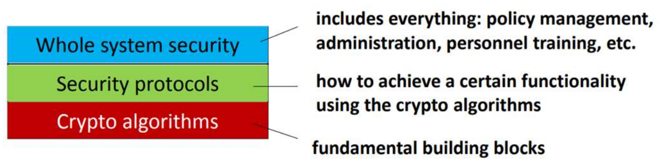

Whole system security
Security protocols
Crypto algorithms
includes everything: policy management, administration, personnel training, etc.
how to achieve a certain functionality using the crypto algorithms
fundamental building blocks

Figure 4: *Layers of a Security System*

B.1. Computer & Network Security

Computer security protects computer systems and its components from illegal access or damage, which includes hardware, software, and data. It includes methods to avoid malware infections and illegal data intrusions.

Network security, a type of computer security, focuses on securing data transfer via networks. It involves policies and procedures for preventing and monitoring unauthorized access, misuse, or alteration of network-accessible resources.

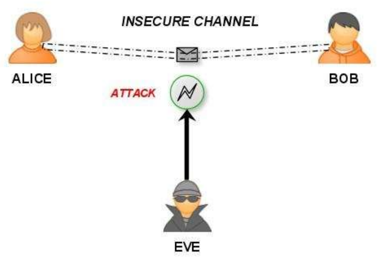


Figure 5: *Schematic of a Two-Party Communication*

Information security, handles with several issues:

- ***Confidentiality:*** It ensures that data is only available to authorized users, protecting privacy and secrecy.
- ***Authentication:*** It confirms the identification of people or systems, allowing only authorized access.
- ***Data integrity:*** It ensures the correctness and completeness of data while avoiding unauthorized changes.
- ***Anonymity:*** It safeguards users' identities throughout interactions.
- ***Non-repudiation:*** It precludes denial of actions, such as transmitting messages or conducting transactions.
- ***Availability:*** It ensures that data and resources are available to authorized users when needed.
- ***Traceability:*** It enables for the tracking of actions back to their source, which is critical for audits and investigations.

B.2. Cryptography

Cryptography is the activity and study of ways for protecting communication and information from potential attackers. Its origins may be traced back to ancient times, with simple substitution ciphers employed in Egypt and Rome giving way to the complicated algorithms of the digital age. This evolution was fueled by the ongoing arms race between code makers and code breakers, which shaped the present landscape of information security. In the internet age, cryptography has become essential for protecting data in a variety of domains, including military communications, financial transactions, and personal privacy.

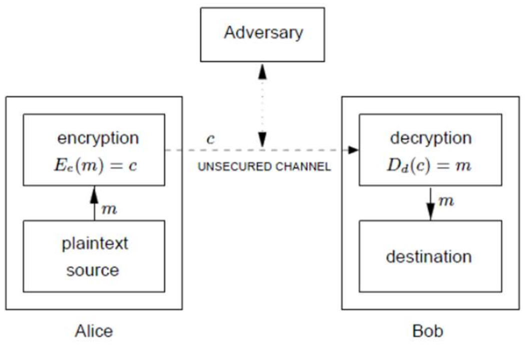


Figure 6: *Schematic of a Two-Party Communication Using Encryption*

Cryptography is critical for maintaining the secrecy, integrity, and validity of information in the digital age. It safeguards sensitive data against unwanted access, guarantees that data is not altered during transmission, and confirms the identities of communication parties. Cryptography protects online transactions, communications, and data storage, making it vital for internet security, digital banking, and personal privacy. Its significance has grown in tandem with the increasing volume of data shared via the internet and the rising threat of cyber-attacks.

Cryptography is essential for safeguarding inter-process communication (IPC) in Linux, as it ensures data integrity, confidentiality, and authentication. This chapter digs into the core encryption mechanisms, providing insights into their operation, strengths, and uses in digital communication.

B.3. Basic Terminology

Communication Participants

- An ***entity*** or ***party*** is someone or something which sends, receives, or manipulates information. Alice and Bob are entities. An entity may be a person, a computer terminal, etc.
- A ***sender*** is an entity in a two-party communication which is the legitimate transmitter of information. In figure, the sender is Alice.
- A ***receiver*** is an entity in a two-party communication which is the intended recipient of information. In Figure, the receiver is Bob.
- An ***adversary*** is an entity in a two-party communication which is neither the sender nor receiver. Various other names are synonymous with adversary such as enemy, attacker, opponent, tapper, eavesdropper, intruder, and interloper. An adversary will often attempt to play the role of either the legitimate sender or the legitimate receiver.

Channels

- A ***channel*** is a means of conveying information from one entity to another.
- A ***physically secure channel*** or secure channel is one which is not physically accessible to the adversary.
- An ***unsecured channel*** is one from which parties other than those for which the information is intended can reorder, delete, insert, or read.
- A ***secured channel*** is one from which an adversary does not have the ability to reorder, delete, insert, or read.
- An ***information security service*** is a method to provide some specific aspect of security. For example, integrity of transmitted data is a security objective, and a method to ensure this aspect is an information security service.
- ***Breaking an information security service*** (which often involves more than simply encryption) implies defeating the objective of the intended service.

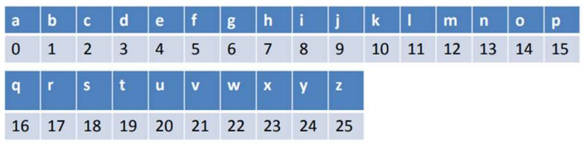
Cryptology

- A ***cryptosystem*** is a general term referring to a set of cryptographic tools used to provide information security services. Most often the term is used in conjunction with primitives providing confidentiality, i.e., encryption.

C. Encryption Methods

C.1. Ceasar (Shift) Cipher

The Caesar cipher is one of the most simple and well-known encryption methods. It is a type of substitution cipher in which each letter in the plaintext is shifted a set number of positions down or up the alphabet. For example, with a shift of one, 'A' would be replaced by 'B', 'B' by 'C', and so on. Also, spaces and punctuations are reserved. This approach is named after Julius Caesar, who allegedly used it to communicate with his generals. The Caesar cipher's simplicity makes it easy to comprehend, but equally easy to break, restricting its practical relevance to modern security requirements.



a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
q	r	s	t	u	v	w	x	y	z						
16	17	18	19	20	21	22	23	24	25						

Figure 7: *Table of Latin Alphabet*

For an n-letter alphabet; P, C, K ∈ Z_n, encryption $E_K(P) = P + K \pmod n$, decryption $D_K(P) = C - K \pmod n$. Let consider the K (shift key) is 4, and plain text is *"This is an encrypted message."*, cipher text can be created from the alphabet table in the figure above. For example, the letter T becomes X since the key is 4. So cipher text becomes:

"XLMW MW ER IRGVCTXIH QIWWEKI."

The Caesar cipher's simplicity is also its primary drawback. It is vulnerable to frequency analysis, which involves an attacker comparing the frequency of letters or groups of letters in the ciphered text to known frequencies in the original message's language. Because the cipher does not dramatically modify these frequencies, it is quite simple to calculate the shift and decrypt the message. Furthermore, because there are only 25 potential shifts in the English alphabet, an attacker can easily try all combinations to decrypt the message.

Breaking Ceasar Cipher

- Eliminating spaces can help to increase safety.
- Counting the number of characters in cipher text (letter E occurs the most frequently in English). So the substraction gives the shift key.
- If just one letter of the plain text along with the corresponding letter cipher text is known, it is easy to deduce the key.

- Choosing a letter as a plain text, The ciphertext gives the key. By trying this method it is possible to break the cipher in maximum 25 iterations.
- Choosing a letter as the cipher text, the plain text is the negative of the key. For example if the plain text is H, the key is -7 ≡ 19 (mod 26).

C.2. RSA (Rivest, Shamir and Adleman) Cryptosystem

The RSA method is a frequently used asymmetric encryption technology for safe data transport. RSA, named after its founders Rivest, Shamir, and Adleman, encrypts data with a public key that can be freely transmitted, whereas decryption requires a private key that the receiver keeps secret. RSA's security is predicated on the practical difficulty of factoring the product of two large prime numbers, making it a key component of digital security in applications such as safe online browsing, email, and corporate data protection.

RSA encryption uses two keys: a public key for encryption and a private key for decryption. Its security arises from the difficulty of factoring huge numbers into primes, which is computationally expensive for large numbers. RSA is used to ensure secure data transmission, digital signatures, and key exchange. Its effectiveness is determined by key size, with longer keys providing higher security but needing more processing resources. RSA's use in SSL/TLS protocols for secure web connections demonstrates its importance in current cryptography.

Considering one letter sized plain text, letter B to be encrypted. The number representation of B is 2. Let public key for encryption is defined as (5, 14). Thus, cipher text is calculated 2^5 (mod 14) ≡ 4 in this case. Then, the letter equivalent of this is D. Also, private key for decryption is chosen as (11, 14). When back-propagation is applied, original text can be reconstructed. 4^{11} (mod 14) ≡ 2 (B).

Generating Keys

In RSA, keys are generated by selecting two large prime numbers (p and q), calculating their product (n), and then determining a number (e) that is coprime with (n) and the product of the primes' decrements. The public key consists of (n) and (e), but the private key is composed of (n) and a number (d) that solves a certain modular equation involving (e). The procedure assures that public and private keys are mathematically connected, allowing for secure encryption and decryption processes.

Let us choose p= 2 and p= 7. n= p*q= 14 in this case. This value will be the modulo in encryption and decryption keys. Remainder numbers which are coprime with n (sharing no common factors with n) which are 1, 3, 5, 9, 11 and 13 in this case. In real scenerio, p and q might be enormous, so calculating coprimes would be hard. The number of remainder numbers is equal to Ø(n)= 6. This term can be also obtained as Ø(n)= (p-1)*(q-1). Then, choosing number e under two conditions:

- 1 < e < Ø(n)
- Coprime with n, Ø(n)

5 can be choosen from four options (2, 3, 4, 5), because only 5 is coprime with n and Ø(n). Then the next step is determining number d for private key. Choosing d: d*e (mod Ø(n)) ≡ 1. In this case, 11 can be chosen from all options (4, 11, 18, 25…). Finally, public and private keys for RSA cryptosystem is generated.

Breaking RSA

The information of the key has a form as: (e, n)(d). Since n and Ø(n) are unknown, p and q can not be determined easily while considering very large n. But it is known that n= p*q, Ø(n)= (p-1)*(q-1) and e*d ≡ 1 (mod Ø(n)). Fastest way to calculate Ø(n) is using Fermat's factorization:

- $n = a^2 - b^2 = (a+b)*(a-b)$
- $b^2 = a^2 - n$
- $a = \lceil\sqrt{n}\rceil$

After *i* numbered iterations of a, b and corresponding a for it can be calculated. Then p= (a+b) and q= (a-b) assumption will enable calculating the private key after this step. But p and q must be checked if they are prime numbers.

C.3. Data Encryption Standard (DES) and Advanced Encryption Standard (AES)

The Data Encryption Standard (DES) is a symmetric-key block encryption algorithm created by IBM in the 1970s and later standardized by NIST. Its principal function is to encrypt and decrypt digital data with a common secret key. DES encrypts and decrypts plaintext blocks, which are typically 64 bits in size, using a 56-bit key. Although DES has been mostly replaced by more secure algorithms such as AES, knowing its operation provides insight into the fundamentals of modern encryption.

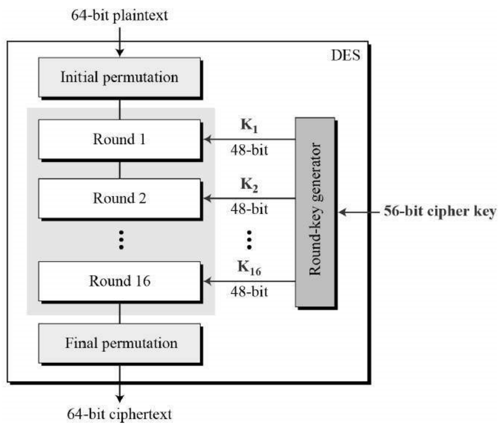


Figure 8: *DES Structure*

The first stage in DES is key generation[5], which converts the 56-bit key into 16 subkeys, each 48 bits long. The procedure starts with an initial permutation and then separates the key into two 28-bit halves. Circular left shifts are made to each half, and subkeys are formed by selecting specified groups of bits using a technique known as key scheduling. These subkeys are subsequently used for further encryption and decryption operations.

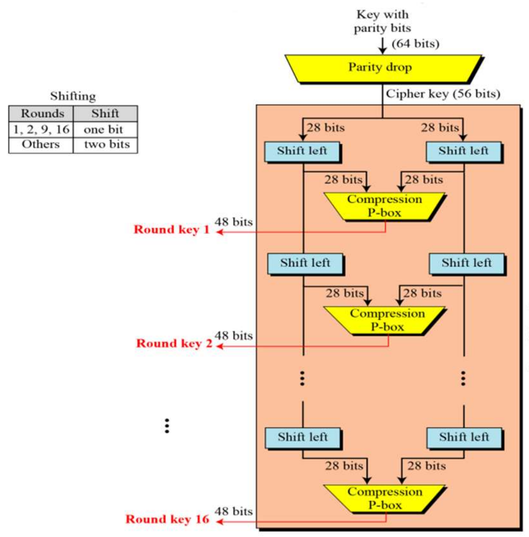


Figure 9: *Key Generation in DES*

DES encryption involves 16 rounds of processing for each plaintext block. Each round includes multiple operations such as substitution, permutation, and key mixing. The plaintext block is first permuted, then transformed in a sequence of rounds. Each round, the block is divided into two halves, expanded, XORed with a subkey, substituted using S-boxes, permuted, and XORed with the other half. This procedure scrambles plaintext into ciphertext in a reversible manner using the proper key.

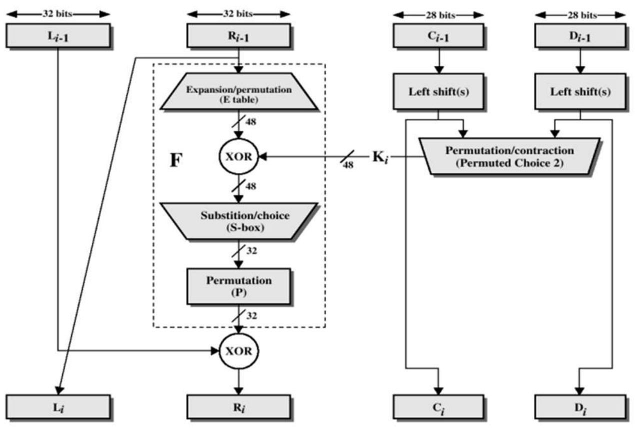


Figure 10: *Single Round of DES Algorithm*

DES decryption is the same as encryption, but in reverse. The ciphertext is initially permuted, then the subkeys are applied in reverse order across 16 rounds of processing. Each round involves the identical operations as encryption, except the subkeys are used in the reverse order. After the final round, the ciphertext block is treated to an inverse initial permutation, yielding the original plaintext block.

Despite its historical relevance, DES has a number of flaws that make it unsuitable for modern cryptography applications. Its small key length renders it vulnerable to brute-force assaults, in which all potential keys can be checked within a reasonable timeframe. Furthermore, developments in cryptanalysis have revealed flaws in the DES algorithm, reducing its security. As a result, DES has been replaced by more powerful encryption protocols such as AES, which provide higher security assurances and improved performance.

AES[6] (Advanced Encryption Standard) is a symmetric encryption algorithm that has become an essential component of modern cryptographic protocols. AES was developed to replace the outdated Data Encryption Standard (DES) and was adopted as a standard by the United States National Institute of Standards and Technology (NIST) in 2001 following a rigorous selection process. Unlike asymmetric encryption algorithms, which use separate keys for encryption and decryption, AES uses a single key for both operations, resulting in a symmetric encryption method. This key is shared by the communicating parties and must be kept secure in order to ensure the secrecy of the encrypted data.

AES's processing on data blocks is crucial to its functionality. Each block is made up of 128 bits, or 16 bytes, and if the plaintext is not a multiple of this block size, padding is used to ensure consistency. AES provides key sizes of 128, 192, and 256 bits, with bigger key sizes providing more security at the expense of computational complexity. The algorithm uses a substitution-permutation network (SPN) to perform its operations. These procedures take place over numerous rounds, with the number of rounds varied according to the key size: 10 rounds for AES-128, 12 rounds for AES-192, and 14 rounds for AES-256.

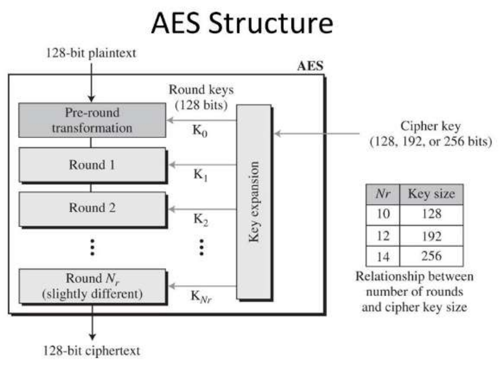


Figure 11: *AES Structure*

The AES encryption process begins with key expansion[7], which converts the initial key into a series of round keys, one for each round of encryption. Each round of AES encryption consists of four major steps: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. In the *SubBytes* step, each byte in the input block is replaced with a corresponding byte from a substitution table, introducing non-linearity into the encryption process. *ShiftRows* is the process of cyclically shifting the bytes within each row of the block, which contributes to data diffusion. *MixColumns* treats the block's columns as polynomials and multiplies them with fixed polynomials modulo an irreducible polynomial. Finally, *AddRoundKey* applies bitwise XOR to merge the state and round key.

SubBytes: It is the first stage in each round of AES encryption. Its aim is to replace each byte in the input block with a corresponding byte from a prepared substitution table, known as the S-box. This replacement is a nonlinear process that causes confusion in the data. The S-box is a fixed 16x16 matrix with pre-computed values. Each byte in the input block is replaced with the value from the relevant row and column in the S-box. This transformation ensures that even minor changes in the input block cause huge changes in the output, making it difficult for attackers to identify patterns.

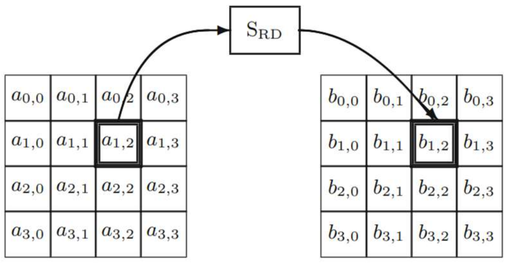


Figure 12: *SubBytes acts on the individual bytes of the state*

ShiftRows: It is the next stage in each round of AES encryption. This procedure consists of cyclically shifting the bytes within each row of the block. The bytes in the second row are shifted one position to the left; those in the third row are shifted two places to the left; and those in the fourth row are shifted three positions to the left. This phase adds to data diffusion by spreading each byte's influence across numerous columns. ShiftRows ensures that neighboring bytes interact with one another during successive encryption steps, hence improving the algorithm's overall security.

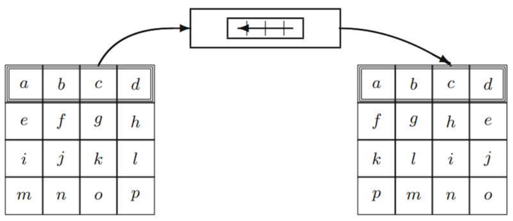


Figure 13: *ShiftRows operates on the rows of the state*

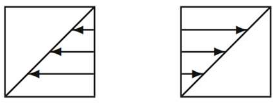


Figure 14: *Pictograms for ShiftRows (left) and InvShiftRows (right)*

MixColumns: With the exception of the final round of AES encryption, MixColumns follows ShiftRows. This procedure treats the block's columns as polynomials over the finite field GF(2^8). MixColumns multiplies each byte in a column by a fixed polynomial modulo an irreducible polynomial. The result replaces the original byte, yielding a linear transformation of the data. This process further distributes the data and ensures that each byte of the output is dependent on multiple bytes from the input. By creating this reliance, MixColumns improves the overall security of AES encryption, making it more resistant to cryptanalytic attacks.

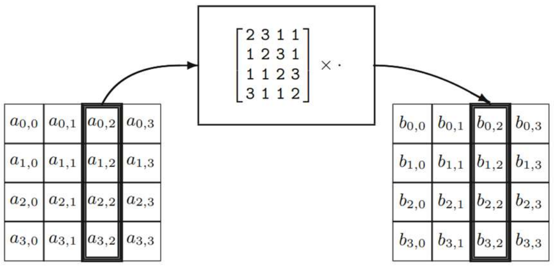


Figure 15: *MixColumns operates on the columns of the state*

AddRoundKey: It is the last step in each round of AES encryption. In this stage, the block's current state is joined with a round key generated from the main encryption key. Each byte of the state is bitwise XORed with its matching byte from the round key. The round key created during key expansion is unique to the current round of encryption. AddRoundKey ensures that each round of encryption is separate and depends on the key by introducing the round key's unique effect into the state. This phase further obscures the relationship between the plaintext and the ciphertext, which improves the security of AES encryption.

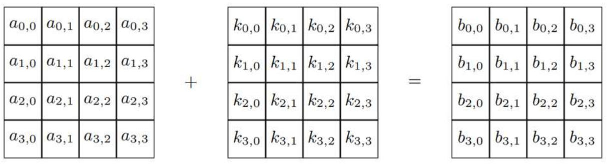


Figure 16: *The round key is added to the state with a bitwise XOR*

AES is a symmetric key algorithm, which means it uses the same key for encryption and decoding. This differs from asymmetric key methods, which use two separate keys (public and private) for encryption and decryption. Symmetric key methods are often faster and more efficient for large volumes of data; nevertheless, key management can be difficult because securely communicating the key with the intended recipient is critical.

AES operates on fixed-size data blocks (128 bits). Different modes of operation can be used to encrypt data that does not fit into a single block, or to encrypt numerous blocks with increased security, such as Electronic Codebook (ECB), Cipher Block Chaining (CBC), or Galois/Counter Mode (GCM). These modes specify how the plaintext is divided into blocks and how the encryption process is performed on each block:

Electronic Codebook (ECB): This mode encrypts each block separately with the same key, which can reveal patterns in the ciphertext if the plaintext contains repeated data. It is typically regarded as less secure and not recommended for most purposes.

Cipher Block Chaining (CBC): This mode creates a dependency between blocks by XORing the previous block's ciphertext with the current block's plaintext prior to encryption. The first block uses an Initialization Vector (IV) to add randomization. This mode offers more security than ECB, but it is vulnerable to certain attacks if the IV is predictable or overused.

AES is regarded extremely secure and is widely used in a variety of applications, including government, military, and commercial use. The algorithm's security stems mostly from its key size, which makes it resistant to brute-force attacks. Brute-force attacks attempt to decrypt the ciphertext by trying every conceivable key combination; however, with AES key sizes (128, 192, or 256 bits), the number of possible possibilities is so huge that it is currently deemed impossible to break AES encryption using this technique.

Today, AES is a cornerstone of digital security, with use ranging from government communications to commercial transactions. Its exceptional resilience against brute-force attacks, combined with key size versatility (128, 192, and 256 bits), ensures strong defense mechanisms for sensitive information protection. AES's efficiency and security have won it a key role in global standards and protocols, making it a must-have tool in the fight against cybersecurity threats. As we traverse the digital age, AES's importance grows, emphasizing its important role in protecting digital assets and communications around the world.

II. DEMONSTRATION

A. Linux Inter-Process Communication with Named Pipe

In C programming on Linux systems, named pipes, often known as FIFOs (First In, First Out), provide a means for inter-process communication (IPC) that allows unrelated programs to share data. Unlike anonymous pipes, which are commonly used for parent-child process communication, named pipes exist independently of the processes that utilize them and persist in the file system, offering a path for communication between any processes that have access to the filesystem.

To create a named pipe[8], use the ***mkfifo*** system function or command. This method creates a FIFO special file with the specified name in the filesystem. The syntax for ***mkfifo*** in a C program is the following:

```c
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

Where ***"pathname"*** is the name of the FIFO to be created and ***"mode"*** specifies the permissions for the FIFO.

The ***"mode"*** is specified as an octal (base-8) number and represents the file's permission bits. It's similar to the permissions used for regular files and directories. The ***"mode"*** is influenced by the process's umask, which may restrict the permissions set during the creation of the FIFO. The ***"mode"*** parameter is composed of three groups of permissions:

- **Owner permissions:** What actions the owner of the file can perform.
- **Group permissions:** What actions users who are members of the file's group can perform.
- **Others permissions:** What actions all other users can perform.

Each group can have permissions for reading (r), writing (w) and execution (x), represented by octal numbers:

- 4 (100 in binary) stands for **read** permission.
- 2 (010 in binary) stands for **write** permission.
- 1 (001 in binary) stands for **execute** permission.
- 0 stands for **no permission**.

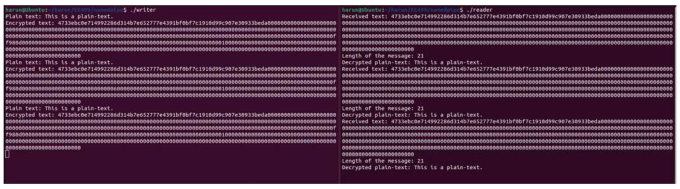
These permissions are added to together to get the total permission value for each group. The final mode is a concatenation of these values for the owner, group and others in that order.

A.1. Opening the Named Pipe

Once created, processes can open the named pipe using ***open(.)***, just as they would with regular files. A process can open the FIFO in read-only (RDONLY) or write-only (WRONLY) mode, depending on its role:

- The **writer** process, which send data into the FIFO, opens it for writing.
- The **reader** process, which reads the data, opens it for reading.

When a process is done wit the FIFO, it can close it using ***close(.)***, similar to files.



```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./writer
Plain text: This is a plain-text.
Encrypted text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000f
f980d000000000000000000000000000000000600000000000000000000000000000000000100000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
Plain text: This is a plain-text.
Encrypted text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000f
f980d000000000000000000000000000000000600000000000000000000000000000000000100000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
Plain text: This is a plain-text.
Encrypted text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000f
f980d000000000000000000000000000000000600000000000000000000000000000000000100000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
```

```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./reader
Received text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
Length of the message: 21
Decrypted plain-text: This is a plain-text.
Received text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
Length of the message: 21
Decrypted plain-text: This is a plain-text.
Received text: 4733ebc0e714992286d314b7e652777e4391bf0bf7c1910d99c907e30933beda000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000
Length of the message: 21
Decrypted plain-text: This is a plain-text.
```

Figure 17: *Writer and reader processes*

A.2. Reading and Writing to a Named Pipe

Reading from and writing to a named pipe are accomplished with the ***read(.)*** and ***write(.)*** system methods, respectively. These calls halt the calling process: a ***read(.)*** call on an empty FIFO will block until there is data to read, and a ***write(.)*** call on a full FIFO will block until there is space to write new data. This blocking feature allows the producer and consumer processes to synchronize without the need for additional coordination code.

```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./writer
█
```

Figure 18: *Writer process waits the data in FIFO to be read*

B. *Ceasar, RSA, AES and DES Algorithms in C*

B.1. Overview of the OpenSSL

The OpenSSL project[9], a powerful, commercial-grade, and feature-rich toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols, has expanded dramatically over time. One critical component of its evolution is the creation and improvement of its cryptography library, which includes a diverse set of cryptographic algorithms and capabilities like as AES, DES, and RSA. In subsequent releases, OpenSSL has continued to evolve, improving its support for these methods via its digital envelope library.

B.2. Digital Envelope Library

The digital envelope technique[10] secures a communication by using asymmetric encryption to encrypt a symmetric key, which is then used to encrypt the message or data. This method combines the effectiveness of symmetric encryption algorithms (such as AES and DES) for large-scale data encryption with the security of asymmetric encryption algorithms (such as RSA) for safe key exchange.

The EVP (Envelope) interface serves as the foundation for OpenSSL's digital envelope capabilities. The EVP interface provides a higher level abstraction of the different cryptographic methods. It provides a uniform method to encryption and decryption, hashing, and digital signature operations using diverse algorithms. By abstracting the complexities of each cryptographic technique, the EVP interface makes it easier to integrate encryption into programs while also ensuring that the algorithms are utilized appropriately and securely.

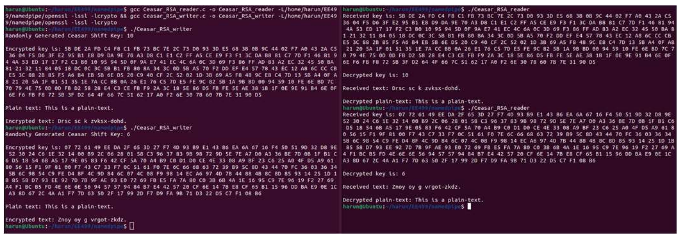
B.3. Usage in C and Named Pipes

When implementing cryptographic operations in C with OpenSSL[11], developers use the EVP interface to execute encryption and decryption. This method enables a seamless transition between multiple algorithms (AES, DES, RSA) without requiring significant changes to the codebase. For example, developers can encrypt data with AES for efficiency and then use RSA to encrypt the AES key, resulting in a secure digital envelope.

As it mentioned before, cryptography is essential in scenarios involving inter-process communication (IPC), such as when employing named pipes (FIFOs), to ensure the confidentiality and integrity of the data being transmitted. In Unix-like operating systems, named pipes can be built and accessed using functions such as ***mkfifo*** to simplify communication between processes running on the same machine. Developers can use OpenSSL's cryptographic capabilities to encrypt data before sending it via the pipe and decrypt it upon receipt. This method assures that even if the data is intercepted while in transit over the designated pipe, it is protected and unreadable without the correct decryption key.

This combination of OpenSSL with named pipes for safe IPC is especially useful in applications that require secure transport of sensitive information between various components or services running on the same system. Using OpenSSL's digital envelope features ensures that data enclosed within a secure envelope is efficiently encrypted and securely sent, combining the strengths of symmetric and asymmetric cryptography.

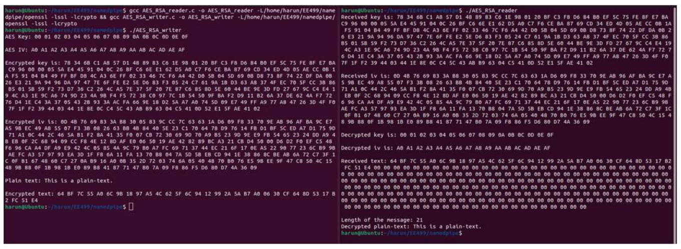
OpenSSL's digital envelope library, which supports the AES, DES, and RSA algorithms, is a powerful and versatile toolset for performing cryptographic operations in C, including secure inter-process communication via named pipes. By abstracting the intricacies of cryptographic operations and assuring secure key and data handling, OpenSSL allows developers to create more secure applications that can confidently protect sensitive information from interception and unwanted access.



```
harun@Ubuntu:~/harun/EE499/namedpipe$ gcc Ceasar_RSA_reader.c -o Ceasar_RSA_reader -L/home/harun/EE49
9/namedpipe/openssl -lssl -lcrypto && gcc Ceasar_RSA_writer.c -o Ceasar_RSA_writer -L/home/harun/EE49
9/namedpipe/openssl -lssl -lcrypto
harun@Ubuntu:~/harun/EE499/namedpipe$ ./Ceasar_RSA_writer
Randomly Generated Ceasar Shift Key: 10

Encrypted key is: 5B DE 2A FD C4 F8 C1 FB 73 BC 7E 2C 73 D0 93 3D E5 68 3B 0B 9C 44 02 F7 A0 43 2A C5
 36 04 F5 D6 3F E2 95 B1 E8 D9 DA 9E 70 A3 D8 C1 E1 C2 FF A5 CE E9 F3 F1 3C DA B8 81 C7 7D F1 46 B1 9
4 4A 53 ED 17 17 F2 C3 B0 10 95 94 5D 0F 9A E7 41 EC 4C 6A 0C 3D 69 F3 86 FF AD 83 A2 EC 32 45 50 BA
81 21 32 11 84 05 18 DC 0C 3C 5B B1 FB 80 8A 34 3C 0D 5B A5 70 F2 DD EF E4 57 78 43 EC 12 A8 6C CC CB
 E5 3C 88 2B 85 F5 A6 B4 EB 5B 6E D5 20 C9 40 CF 2C 52 02 1D 3B 69 A5 F8 48 9C E8 C4 7D 13 5B A4 0F A
8 21 20 5A 1F 01 51 35 1E 7A CC BB 0A 26 E1 76 C5 7D E5 FE 9C 82 5B 1A 9B BD 00 94 59 10 FE 6E BD 7C
70 79 4E 75 0D 0D FB D2 58 28 E4 C3 CE FB F9 2A 3C 18 5E 86 D5 FB FE 5E AE 38 1B 1F 0E 9E 91 B4 6E 0F
 6E F6 FB F8 72 5B 3F D2 64 4F 66 7C 51 62 17 A0 F2 6E 30 78 60 7B 7E 31 90 D5

Plain text: This is a plain-text.

Encrypted text: Drsc sc k zvksx-dohd.
harun@Ubuntu:~/harun/EE499/namedpipe$ ./Ceasar_RSA_writer
Randomly Generated Ceasar Shift Key: 6

Encrypted key is: 07 72 61 49 EE DA 2F 65 3D 27 F7 4D 93 B9 E1 43 86 EA 6A 67 16 F4 50 51 9D 32 D8 9E
 52 30 24 C6 1E 32 14 00 B9 2C 06 28 01 58 C3 96 37 83 9B 98 72 9D 5E 7E A7 D0 A3 36 BE 7D 0B 1F B1 C
6 D5 18 54 6B A5 17 9E 05 83 F6 42 CF 5A 70 A4 B9 C0 D1 D0 CE 4E 33 08 A9 BF 23 C6 25 A0 4F D5 A9 61
80 56 15 F1 9F 81 00 F7 43 C7 33 F7 0C 51 61 F0 7E 6C 66 68 63 72 39 B9 5C 8D 43 44 70 FC 36 03 36 34
 5B 6C 98 54 C9 FE D4 8F 4C 9D 84 6C 07 4C 08 F9 98 14 EC A6 97 4D 7B 44 88 4B 8C 8D 85 93 14 25 1D 1
B 85 58 D7 93 EE 92 7D 7B 9F AE 93 E0 72 69 FB E5 FA 7A 80 C0 3B 6B 4A 1E 16 95 C9 7E 96 19 F2 27 69
A4 F1 BC B5 FD 4E 6E 6E 56 94 57 57 94 84 B7 E4 42 57 20 CF 6E 14 7B E8 CF 65 B1 15 96 DD BA E9 0E 1C
 A3 8D 67 2C 4A A1 F7 7D 63 50 2F 17 99 2D F7 D9 FA 9B 71 D3 22 D5 C7 F1 08 B6

Plain text: This is a plain-text.

Encrypted text: Znoy oy g vrgot-zkdz.
harun@Ubuntu:~/harun/EE499/namedpipe$ ▯
```

```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./Ceasar_RSA_reader
Received key is: 5B DE 2A FD C4 F8 C1 FB 73 BC 7E 2C 73 D0 93 3D E5 68 3B 0B 9C 44 02 F7 A0 43 2A C5
36 04 F5 D6 3F E2 95 B1 E8 D9 DA 9E 70 A3 D8 C1 E1 C2 FF A5 CE E9 F3 F1 3C DA B8 81 C7 7D F1 46 B1 94
 4A 53 ED 17 17 F2 C3 B0 10 95 94 5D 0F 9A E7 41 EC 4C 6A 0C 3D 69 F3 86 FF AD 83 A2 EC 32 45 50 BA 8
1 21 32 11 84 05 18 DC 0C 3C 5B B1 FB 80 8A 34 3C 0D 5B A5 70 F2 DD EF E4 57 78 43 EC 12 A8 6C CC CB
E5 3C 88 2B 85 F5 A6 B4 EB 5B 6E D5 20 C9 40 CF 2C 52 02 1D 3B 69 A5 F8 48 9C E8 C4 7D 13 5B A4 0F A8
 21 20 5A 1F 01 51 35 1E 7A CC BB 0A 26 E1 76 C5 7D E5 FE 9C 82 5B 1A 9B BD 00 94 59 10 FE 6E BD 7C 7
0 79 4E 75 0D 0D FB D2 58 28 E4 C3 CE FB F9 2A 3C 18 5E 86 D5 FB FE 5E AE 38 1B 1F 0E 9E 91 B4 6E 0F
6E F6 FB F8 72 5B 3F D2 64 4F 66 7C 51 62 17 A0 F2 6E 30 78 60 7B 7E 31 90 D5

Decrypted key is: 10

Received text: Drsc sc k zvksx-dohd.

Decrypted plain-text: This is a plain-text.
harun@Ubuntu:~/harun/EE499/namedpipe$ ./Ceasar_RSA_reader
Received key is: 07 72 61 49 EE DA 2F 65 3D 27 F7 4D 93 B9 E1 43 86 EA 6A 67 16 F4 50 51 9D 32 D8 9E
52 30 24 C6 1E 32 14 00 B9 2C 06 28 01 58 C3 96 37 83 9B 98 72 9D 5E 7E A7 D0 A3 36 BE 7D 0B 1F B1 C6
 D5 18 54 6B A5 17 9E 05 83 F6 42 CF 5A 70 A4 B9 C0 D1 D0 CE 4E 33 08 A9 BF 23 C6 25 A0 4F D5 A9 61 8
0 56 15 F1 9F 81 00 F7 43 C7 33 F7 0C 51 61 F0 7E 6C 66 68 63 72 39 B9 5C 8D 43 44 70 FC 36 03 36 34
5B 6C 98 54 C9 FE D4 8F 4C 9D 84 6C 07 4C 08 F9 98 14 EC A6 97 4D 7B 44 88 4B 8C 8D 85 93 14 25 1D 1B
 85 58 D7 93 EE 92 7D 7B 9F AE 93 E0 72 69 FB E5 FA 7A 80 C0 3B 6B 4A 1E 16 95 C9 7E 96 19 F2 27 69 A
4 F1 BC B5 FD 4E 6E 6E 56 94 57 57 94 84 B7 E4 42 57 20 CF 6E 14 7B E8 CF 65 B1 15 96 DD BA E9 0E 1C
A3 8D 67 2C 4A A1 F7 7D 63 50 2F 17 99 2D F7 D9 FA 9B 71 D3 22 D5 C7 F1 08 B6

Decrypted key is: 6

Received text: Znoy oy g vrgot-zkdz.

Decrypted plain-text: This is a plain-text.
harun@Ubuntu:~/harun/EE499/namedpipe$ █
```

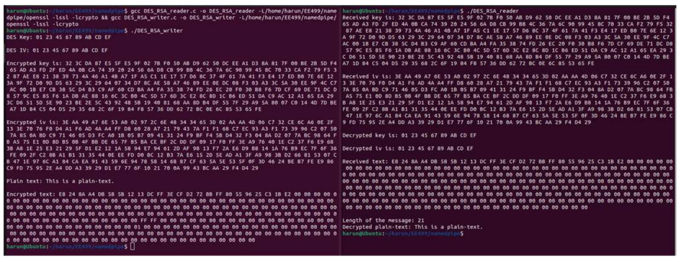
Figure 19: *Ceasar Encryption with RSA secured key (From Appendix A.1. and A.2.)*



```
harun@Ubuntu:~/harun/EE499/namedpipe$ gcc AES_RSA_reader.c -o AES_RSA_reader -L/home/harun/EE499/name
dpipe/openssl -lssl -lcrypto && gcc AES_RSA_writer.c -o AES_RSA_writer -L/home/harun/EE499/namedpipe/
openssl -lssl -lcrypto
harun@Ubuntu:~/harun/EE499/namedpipe$ ./AES_RSA_writer
AES Key: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

AES IV: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

Encrypted key is: 78 34 6B C1 A8 57 D1 48 89 83 C6 1E 98 01 20 BF C3 F8 D6 84 B0 EF 5C 75 FE 8F E7 BA
 C9 96 00 00 05 5A E4 45 91 04 0C 26 BF C6 6E E1 62 D5 A0 C7 F6 CE BA 87 69 CD 34 ED 4D 05 AE CC 0B 1
A F5 91 04 B4 49 FF BF D8 4C A3 6E FF 02 33 46 7C F6 A4 42 D0 5B 04 5D 69 0B D8 73 8F 74 22 DF DA 0B
26 E3 21 9A 94 96 DA 97 47 7E 6F FE E2 5E D6 83 F3 05 24 C7 61 9A 1B D3 63 A8 37 4F EC 70 5F CC 38 86
 B5 01 5B 59 F2 73 D7 36 C2 26 4C A5 7E 37 5F 20 7E 87 C6 85 8D 5E 60 44 BE 9E 3D FD 27 67 9C C4 E4 1
9 4C A3 1E 9C A6 74 9D 23 4A 9B F4 F5 72 38 C0 97 7C 1B 54 50 9F BA F2 D9 11 B2 6A 37 DE 62 4A F7 72
76 D4 1E C4 3A 37 05 43 2B 93 3A AC FA 66 9E 18 D2 5A A7 A0 74 5D 09 E7 49 FF A9 77 A8 47 26 3D 4F F0
 7F 1F F2 39 44 03 44 1E 8E 0C C4 5C 43 AB B9 63 04 C5 41 0D 52 E1 5F AE 41 02

Encrypted iv is: 0D 4B 76 69 83 3A B8 30 05 83 9C CC 7C 63 63 1A D6 09 F8 33 70 9E AB 96 AF BA 9C E7
A5 9B EC 49 A8 55 07 F3 3B 08 26 63 B8 4B 84 40 5E 23 C1 70 64 78 D9 76 14 FB D1 BF 5C ED A7 D1 75 9D
 71 A1 0C 44 2C 46 5A B1 F2 8A 41 35 F0 07 CB 72 30 69 9D 70 A9 B5 23 9D 9E E9 FB 54 65 23 24 DD A9 4
B EB 0F 2C 68 94 09 CC F8 4E 12 8D AF E0 06 50 19 AE 42 82 89 BC A3 21 CB D4 50 00 D6 D2 F0 EF C5 48
F6 96 CA A4 DF A9 E9 42 4C 05 85 4A 9C 79 B0 A7 FC 69 71 37 44 EC 21 6F 17 0E A5 22 90 77 23 6C B9 9B
 AE FC A3 57 97 93 EA 3D 1F F8 6A 11 FA 13 70 B8 04 7A 5D 5B EB CD 94 1E 38 86 8C BE AB 6A 72 C7 3F 1
C 0F B1 67 48 60 C7 27 0A B9 16 A0 0B 35 2D 72 03 74 6A 05 40 48 70 B0 76 E5 9B EE 9F 47 C8 50 4C 15
48 9B 88 0F 1B 98 1B E0 89 B8 41 87 71 47 B0 7A 09 F8 86 F5 D6 80 D7 4A 36 09

Plain text: This is a plain-text.

Encrypted text: 64 BF 7C 55 A0 6C 9B 18 97 A5 4C 62 5F 6C 94 12 99 2A 5A B7 A0 06 30 CF 64 8D 53 17 B
2 FC 51 E4
harun@Ubuntu:~/harun/EE499/namedpipe$ ▯
```

```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./AES_RSA_reader
Received key is: 78 34 6B C1 A8 57 D1 48 89 83 C6 1E 98 01 20 BF C3 F8 D6 84 B0 EF 5C 75 FE 8F E7 BA
C9 96 00 00 05 5A E4 45 91 04 0C 26 BF C6 6E E1 62 D5 A0 C7 F6 CE BA 87 69 CD 34 ED 4D 05 AE CC 0B 1A
 F5 91 04 B4 49 FF BF D8 4C A3 6E FF 02 33 46 7C F6 A4 42 D0 5B 04 5D 69 0B D8 73 8F 74 22 DF DA 0B 2
6 E3 21 9A 94 96 DA 97 47 7E 6F FE E2 5E D6 83 F3 05 24 C7 61 9A 1B D3 63 A8 37 4F EC 70 5F CC 38 86
B5 01 5B 59 F2 73 D7 36 C2 26 4C A5 7E 37 5F 20 7E 87 C6 85 8D 5E 60 44 BE 9E 3D FD 27 67 9C C4 E4 19
 4C A3 1E 9C A6 74 9D 23 4A 9B F4 F5 72 38 C0 97 7C 1B 54 50 9F BA F2 D9 11 B2 6A 37 DE 62 4A F7 72 7
6 D4 1E C4 3A 37 05 43 2B 93 3A AC FA 66 9E 18 D2 5A A7 A0 74 5D 09 E7 49 FF A9 77 A8 47 26 3D 4F F0
7F 1F F2 39 44 03 44 1E 8E 0C C4 5C 43 AB B9 63 04 C5 41 0D 52 E1 5F AE 41 02

Received iv is: 0D 4B 76 69 83 3A B8 30 05 83 9C CC 7C 63 63 1A D6 09 F8 33 70 9E AB 96 AF BA 9C E7 A
5 9B EC 49 A8 55 07 F3 3B 08 26 63 B8 4B 84 40 5E 23 C1 70 64 78 D9 76 14 FB D1 BF 5C ED A7 D1 75 9D
71 A1 0C 44 2C 46 5A B1 F2 8A 41 35 F0 07 CB 72 30 69 9D 70 A9 B5 23 9D 9E E9 FB 54 65 23 24 DD A9 4B
 EB 0F 2C 68 94 09 CC F8 4E 12 8D AF E0 06 50 19 AE 42 82 89 BC A3 21 CB D4 50 00 D6 D2 F0 EF C5 48 F
6 96 CA A4 DF A9 E9 42 4C 05 85 4A 9C 79 B0 A7 FC 69 71 37 44 EC 21 6F 17 0E A5 22 90 77 23 6C B9 9B
AE FC A3 57 97 93 EA 3D 1F F8 6A 11 FA 13 70 B8 04 7A 5D 5B EB CD 94 1E 38 86 8C BE AB 6A 72 C7 3F 1C
 0F B1 67 48 60 C7 27 0A B9 16 A0 0B 35 2D 72 03 74 6A 05 40 48 70 B0 76 E5 9B EE 9F 47 C8 50 4C 15 4
8 9B 88 0F 1B 98 1B E0 89 B8 41 87 71 47 B0 7A 09 F8 86 F5 D6 80 D7 4A 36 09

Decrypted key is: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

Decrypted iv is: A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF

Received text: 64 BF 7C 55 A0 6C 9B 18 97 A5 4C 62 5F 6C 94 12 99 2A 5A B7 A0 06 30 CF 64 8D 53 17 B2
 FC 51 E4 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Length of the message: 21
Decrypted plain-text: This is a plain-text.
harun@Ubuntu:~/harun/EE499/namedpipe$
```

Figure 20: *AES Encryption with RSA secured key (From Appendix A.3. and A.4.)*



```
harun@Ubuntu:~/harun/EE499/namedpipe$ gcc DES_RSA_reader.c -o DES_RSA_reader -L/home/harun/EE499/name
dpipe/openssl -lssl -lcrypto && gcc DES_RSA_writer.c -o DES_RSA_writer -L/home/harun/EE499/namedpipe/
openssl -lssl -lcrypto
harun@Ubuntu:~/harun/EE499/namedpipe$ ./DES_RSA_writer
DES Key: 01 23 45 67 89 AB CD EF

DES IV: 01 23 45 67 89 AB CD EF

Encrypted key is: 32 3C DA 87 E5 5F E5 9F 02 7B F0 50 AB D9 62 50 DC EE A1 D3 8A 81 7F 00 BE 2B 5D F4
 65 AD A3 FD 2F ED 4A 0B CA 74 39 20 24 56 6A D8 CB 99 BB 4C 36 7A 6C 90 99 45 8C 70 33 CA F2 79 F5 3
2 87 AE E8 21 38 39 73 4A 46 A1 4B A7 1F A5 C1 1E 17 57 D6 8C 37 4F 61 7A 41 F3 E4 17 ED B0 7E 6E 12
3A 9F 72 D0 9D D5 63 29 3C 29 64 07 34 D7 8C AE 58 A7 46 89 EE 0E DC 08 F3 03 A3 3C 5A 30 EE 9F 4C C7
 AC 00 1B E7 CB 38 5C D4 B3 C9 AF 60 CD 8A A4 FA 35 38 74 FD 26 EC 20 F0 30 B8 F6 7D CF 69 DE 71 DC D
8 57 9C E5 85 F6 1A DB AE 88 16 6C 3C B0 4C 5D 57 6D 3C E2 8C 8D 1C B6 ED 51 DA C9 AC 12 A1 65 EA 29
3C D6 51 5D 5E 90 23 BE 2E 5C 43 92 48 5B 19 40 81 68 AA 8D B4 DF 55 7F 29 A9 5A 80 07 C0 14 4D 7D BE
 A7 1D 84 C5 04 D5 29 35 68 2C 6F 19 84 F8 57 36 DD 62 72 BC 0E 6C 85 53 65 FE

Encrypted iv is: 3E AA 49 A7 6E 53 A0 02 97 2C 6E 48 34 34 65 3D 02 AA AA 4D 06 C7 32 CE 6C A6 0E 2F
13 3E 70 76 F0 D4 A1 F6 AD 4A A4 FF D8 60 28 A7 21 79 43 7A F1 F1 68 C7 EC 93 A3 F1 73 39 96 C2 07 50
 7A 85 0A BD C9 71 46 05 D3 FC A0 1B 05 B7 09 41 31 24 F9 BF F4 5B D4 32 F3 04 8A D2 07 7A BC 98 64 F
B A5 75 E1 0D 8D B5 0B 4F BB DE 65 7F B5 BA CE BF 2C DD DF 09 17 F0 FF 3E A9 76 40 1E C2 37 F6 E9 68
3B A8 1E 25 E3 21 29 5F D1 E2 12 1A 58 94 E7 94 61 2D AF 98 13 F7 2A E6 D9 B8 14 1A 76 B9 EC 7F 6F 36
 FE 09 2F C2 8B A1 B1 31 35 44 0E EE FD D0 BC 12 B3 7A E6 15 2D 5E AD A1 3F A9 98 3B D2 66 81 53 07 C
B 47 1E 97 6C A1 84 CA EA 91 43 59 6E 94 78 58 14 68 87 CF 63 5A 5E 53 5F 0F 3D 46 24 BE B7 FE E9 86
C9 FD 75 95 2E A4 DD A3 39 29 D1 E7 77 6F 10 21 70 0A 99 43 BC AA 29 F4 D4 29

Plain text: This is a plain-text.

Encrypted text: E8 24 8A A4 DB 58 5B 12 13 DC FF 3E CF D2 72 BB FF 80 55 96 25 C3 1B E2 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 60 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
harun@Ubuntu:~/harun/EE499/namedpipe$
```

```
harun@Ubuntu:~/harun/EE499/namedpipe$ ./DES_RSA_reader
Received key is: 32 3C DA 87 E5 5F E5 9F 02 7B F0 50 AB D9 62 50 DC EE A1 D3 8A 81 7F 00 BE 2B 5D F4
65 AD A3 FD 2F ED 4A 0B CA 74 39 20 24 56 6A D8 CB 99 BB 4C 36 7A 6C 90 99 45 8C 70 33 CA F2 79 F5 32
 87 AE E8 21 38 39 73 4A 46 A1 4B A7 1F A5 C1 1E 17 57 D6 8C 37 4F 61 7A 41 F3 E4 17 ED B0 7E 6E 12 3
A 9F 72 D0 9D D5 63 29 3C 29 64 07 34 D7 8C AE 58 A7 46 89 EE 0E DC 08 F3 03 A3 3C 5A 30 EE 9F 4C C7
AC 00 1B E7 CB 38 5C D4 B3 C9 AF 60 CD 8A A4 FA 35 38 74 FD 26 EC 20 F0 30 B8 F6 7D CF 69 DE 71 DC D8
 57 9C E5 85 F6 1A DB AE 88 16 6C 3C B0 4C 5D 57 6D 3C E2 8C 8D 1C B6 ED 51 DA C9 AC 12 A1 65 EA 29 3
C D6 51 5D 5E 90 23 BE 2E 5C 43 92 48 5B 19 40 81 68 AA 8D B4 DF 55 7F 29 A9 5A 80 07 C0 14 4D 7D BE
A7 1D 84 C5 04 D5 29 35 68 2C 6F 19 84 F8 57 36 DD 62 72 BC 0E 6C 85 53 65 FE

Received iv is: 3E AA 49 A7 6E 53 A0 02 97 2C 6E 48 34 34 65 3D 02 AA AA 4D 06 C7 32 CE 6C A6 0E 2F 1
3 3E 70 76 F0 D4 A1 F6 AD 4A A4 FF D8 60 28 A7 21 79 43 7A F1 F1 68 C7 EC 93 A3 F1 73 39 96 C2 07 50
7A 85 0A BD C9 71 46 05 D3 FC A0 1B 05 B7 09 41 31 24 F9 BF F4 5B D4 32 F3 04 8A D2 07 7A BC 98 64 FB
 A5 75 E1 0D 8D B5 0B 4F BB DE 65 7F B5 BA CE BF 2C DD DF 09 17 F0 FF 3E A9 76 40 1E C2 37 F6 E9 68 3
B A8 1E 25 E3 21 29 5F D1 E2 12 1A 58 94 E7 94 61 2D AF 98 13 F7 2A E6 D9 B8 14 1A 76 B9 EC 7F 6F 36
FE 09 2F C2 8B A1 B1 31 35 44 0E EE FD D0 BC 12 B3 7A E6 15 2D 5E AD A1 3F A9 98 3B D2 66 81 53 07 CB
 47 1E 97 6C A1 84 CA EA 91 43 59 6E 94 78 58 14 68 87 CF 63 5A 5E 53 5F 0F 3D 46 24 BE B7 FE E9 86 C
9 FD 75 95 2E A4 DD A3 39 29 D1 E7 77 6F 10 21 70 0A 99 43 BC AA 29 F4 D4 29

Decrypted key is: 01 23 45 67 89 AB CD EF

Decrypted iv is: 01 23 45 67 89 AB CD EF

Received text: E8 24 8A A4 DB 58 5B 12 13 DC FF 3E CF D2 72 BB FF 80 55 96 25 C3 1B E2 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Length of the message: 21
Decrypted plain-text: This is a plain-text.
harun@Ubuntu:~/harun/EE499/namedpipe$
```

Figure 21: *DES Encryption with RSA secured key (From Appendix A.5. and A.6.)*

III. PERFORMANCE ANALYSIS

A. Time Analysis

To evaluate the performance of cryptographic functions inside the project, extensive time analyses were conducted on RSA operations, AES (Advanced Encryption Standard) in various modes (CBC, ECB), DES (Data Encryption Standard), and the Ceasar cipher. This extensive evaluation sought to quantify and evaluate the execution times of these cryptographic techniques under a variety of situations.

The analysis utilized accurate timing techniques, inserting clock measurements shortly before and after the execution of essential cryptographic processes. This comprised time measurements for:

- RSA key encryption and decryption,
- AES and DES encryption and decryption in different modes,
- The full execution time of writer and reader processes.

These measurements were critical for capturing the actual time spent by each cryptographic operation, which provided an accurate representation of their performance.

The writer and reader processes are runned with a bash script (A.13.). The timing data for each run was recorded programmatically and saved in CSV files using a C program created expressly for this purpose. This method enabled automatic and organized data collecting throughout numerous runs, ensuring consistency and reliability of timing information (Appendix A.14).

The timing data was then evaluated with a Python script (Appendix A.15.) that calculated average durations for each cryptographic operation across multiple sessions. The script generated bar graphs for each parameter (for example, RSA key encryption and AES encryption in CBC mode), allowing for a visual comparison of performance across different cryptography methods and modes.

The bar charts displayed average execution durations, allowing for a clear and comparative view of each cryptographic algorithm's efficiency and speed. This visual representation was useful in identifying performance bottlenecks and inefficiencies.

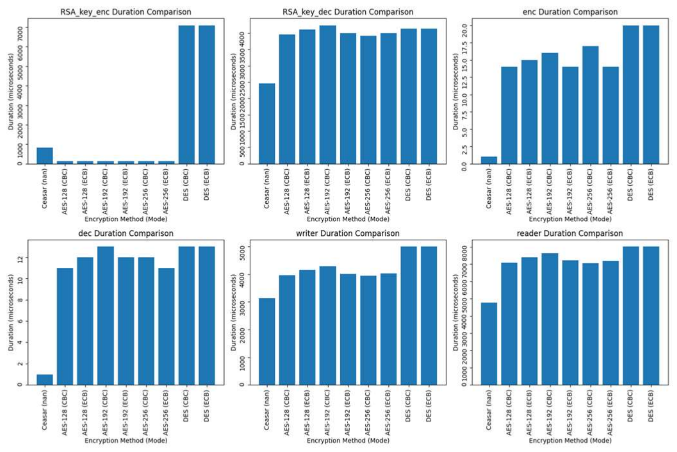
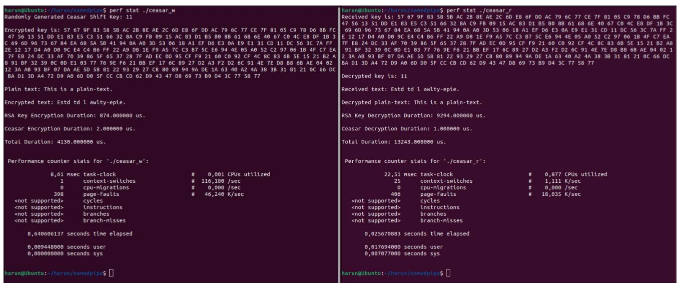


Figure 22: *Time duration comparisons of each parameters overall encryption methods*

B. CPU Utilization Analysis

In addition to time analysis, CPU utilization for cryptographic operations was thoroughly examined to better understand the computational load imposed by various cryptographic algorithms and modes. This research is critical for determining the efficiency and resource requirements of cryptographic operations, which can affect overall system performance, particularly in resource-constrained contexts.

The ***"perf stat"*** command was used to measure CPU utilization and provide precise data on the CPU resources required by each cryptographic operation. The command was run in an environment that had been set up to isolate the CPU core for each process, ensuring that no other system activity interfered with the measurements. This configuration assisted in determining the exact amount of CPU resources used by each task. Also with a ***"taskset -c {core-number}"*** command, the separate operation is applied in order to provide consistent measurements.



```
harun@Ubuntu:~/harun/namedpipe$ perf stat ./ceasar_w
Randomly Generated Ceasar Shift Key: 11

Encrypted key is: 57 67 9F 83 58 5B AC 2B 8E AE 2C 6D E8 6F DD AC 79 6C 77 CE 7F 81 05 C9 78 D6 BB FC
 47 56 13 51 DD E1 83 E5 C3 51 66 32 8A C9 FB 09 15 AC 83 D1 B5 B0 8B 61 68 6E 40 67 C0 4C E8 DF 1B 3
C 69 6D 96 73 67 84 EA 68 5A 5B 41 94 0A A0 3D 53 06 18 A1 EF D6 E3 0A E9 E1 31 CD 11 DC 56 3C 7A FF
2E 12 17 D4 A0 D0 9C E4 C4 B6 FF 22 A9 D8 1E F9 A5 7C C3 B7 5C E6 94 4E 05 A0 52 C2 97 06 1B 4F C7 EA
 7F E8 24 DC 33 AF 70 39 86 5F 65 37 28 7F AD EC 0D 95 CF F9 21 60 C0 92 CF 4C 8C 83 6B 5E 15 21 B2 A
8 91 BF 32 39 0C 0D E1 03 77 76 9E F6 21 BB EF 17 6C 89 27 D2 A3 F2 D2 6C 91 4E 7E D8 B8 6B AE 04 02
12 3A AB 93 BF 07 DA AE 5D 58 81 22 93 29 27 C8 80 89 94 9A DE 1A 63 40 A2 4A 38 3B 31 81 21 0C 66 DC
 BA D1 3D A4 72 D9 A0 6D D0 5F CC CB CD 62 D9 43 47 D8 69 73 B9 D4 3C 77 58 77

Plain text: This is a plain-text.

Encrypted text: Estd td l awlty-epie.

RSA Key Encryption Duration: 874.000000 us.

Ceasar Encryption Duration: 2.000000 us.

Total Duration: 4130.000000 us.


 Performance counter stats for './ceasar_w':

             8,61 msec task-clock                       #    0,001 CPUs utilized
                1      context-switches                 #  116,180 /sec
                0      cpu-migrations                   #    0,000 /sec
              398      page-faults                      #   46,240 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      8,640606137 seconds time elapsed

      0,009448000 seconds user
      0,000000000 seconds sys


harun@Ubuntu:~/harun/namedpipe$
```

```
harun@Ubuntu:~/harun/namedpipe$ perf stat ./ceasar_r
Received key is: 57 67 9F 83 58 5B AC 2B 8E AE 2C 6D E8 6F DD AC 79 6C 77 CE 7F 81 05 C9 78 D6 BB FC
47 56 13 51 DD E1 83 E5 C3 51 66 32 8A C9 FB 09 15 AC 83 D1 B5 B0 8B 61 68 6E 40 67 C0 4C E8 DF 1B 3C
 69 6D 96 73 67 84 EA 68 5A 5B 41 94 0A A0 3D 53 06 18 A1 EF D6 E3 0A E9 E1 31 CD 11 DC 56 3C 7A FF 2
E 12 17 D4 A0 D0 9C E4 C4 B6 FF 22 A9 D8 1E F9 A5 7C C3 B7 5C E6 94 4E 05 A0 52 C2 97 06 1B 4F C7 EA
7F E8 24 DC 33 AF 70 39 86 5F 65 37 28 7F AD EC 0D 95 CF F9 21 60 C0 92 CF 4C 8C 83 6B 5E 15 21 B2 A8
 91 BF 32 39 0C 0D E1 03 77 76 9E F6 21 BB EF 17 6C 89 27 D2 A3 F2 D2 6C 91 4E 7E D8 B8 6B AE 04 02 1
2 3A AB 93 BF 07 DA AE 5D 58 81 22 93 29 27 C8 80 89 94 9A DE 1A 63 40 A2 4A 38 3B 31 81 21 0C 66 DC
BA D1 3D A4 72 D9 A0 6D D0 5F CC CB CD 62 D9 43 47 D8 69 73 B9 D4 3C 77 58 77

Decrypted key is: 11

Received text: Estd td l awlty-epie.

Decrypted plain-text: This is a plain-text.

RSA Key Decryption Duration: 9294.000000 us.

Ceasar Decryption Duration: 1.000000 us.

Total Duration: 13243.000000 us.


 Performance counter stats for './ceasar_r':

            22,51 msec task-clock                       #    0,877 CPUs utilized
               25      context-switches                 #    1,111 K/sec
                0      cpu-migrations                   #    0,000 /sec
              406      page-faults                      #   18,035 K/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      0,025670883 seconds time elapsed

      0,017694000 seconds user
      0,007077000 seconds sys


harun@Ubuntu:~/harun/namedpipe$
```

Figure 23: *Example output of the perf stat command*

The writer and reader processes are runned with a bash script (A.16.). The output from perf stat, which contained CPU utilization percentages, was immediately extracted from the standard output and saved to text files. Each cryptographic process, including RSA key operations, AES and DES encryption/decryption in various modes, and Ceasar cipher operations, was repeated several times to collect thorough CPU utilization data (Appendix A.17.).

The obtained data were processed with a Python script (Appendix A.18.) that calculated the average CPU use for each cryptographic procedure. The script retrieved CPU utilization measurements from the text files, calculated averages, and saved the results in CSV files for future analysis.

Bar plots were then created to visually compare CPU usage across various cryptography algorithms and modes for both writer and reader processes. These charts gave a clear comparison of how each cryptographic approach affects CPU resources, showing the methods that are more computationally efficient.

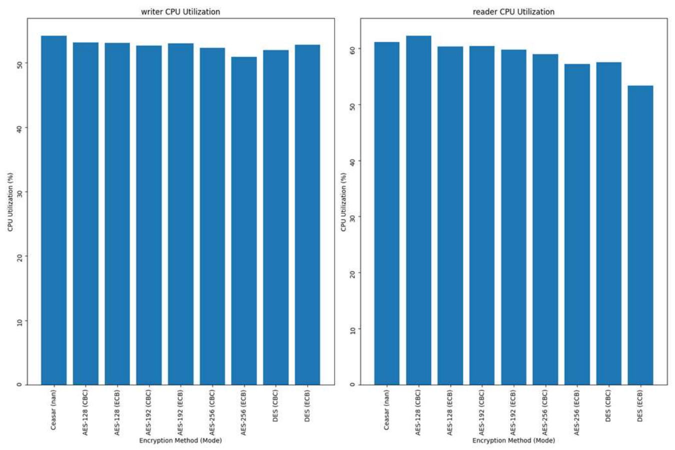


Figure 24: *CPU utilization comparison of both writer and reader processes over different methods*

C. *Theoritical Comparison About Security Parameter of Ceasar, RSA, DES and AES Methods*

In the field of cryptography, encryption methods must be secure in order to ensure data secrecy, integrity, and validity. This section compares the security of multiple cryptographic algorithms used in the project, including Caesar Cipher, RSA, DES, and AES in various configurations.

C.1. Ceasar Cipher

The Ceasar Cipher is one of the most basic and earliest encryption methods. It is a substitution cipher where each letter in the plaintext is shifted a specified number of places down or up the alphabet. However, because to its simplicity, the Ceasar Cipher is particularly vulnerable to frequency analysis and brute force attacks, rendering it unsuitable for any application that requires strong security measures. It is aimed to show the difference with big scales between simple and advanced encryption methods such as AES and DES by the demonstration of this method.

C.2. RSA

RSA is an asymmetric cryptographic algorithm that uses two keys: a public key for encryption and a private key for decryption. Its security is predicated on the difficulties of factorizing huge prime numbers. The strength of RSA encryption is proportional to its key size, with larger keys giving greater security. It is frequently used for secure data transfer and is regarded as secure when combined with appropriate key lengths (e.g., 2048 bits or greater) and padding techniques.

C.3. DES

Data Encryption Standard (DES) is a symmetric-key technique that encrypts data in 64-bit blocks with a 56-bit key. DES is no longer regarded secure due to its short key length, which makes it susceptible to brute-force assaults. The advent of increasingly powerful computing technology has rendered DES outdated, and it has been replaced with more secure protocols.

C.4. AES

AES (Advanced Encryption Standard) is the current gold standard for symmetric key encryption, which is widely utilized in a variety of industries to secure data. AES uses 128-bit blocks and offers key sizes of 128, 192, and 256 bits, resulting in a high level of security. AES is regarded exceedingly secure, and it has resisted significant cryptanalysis and attacks.

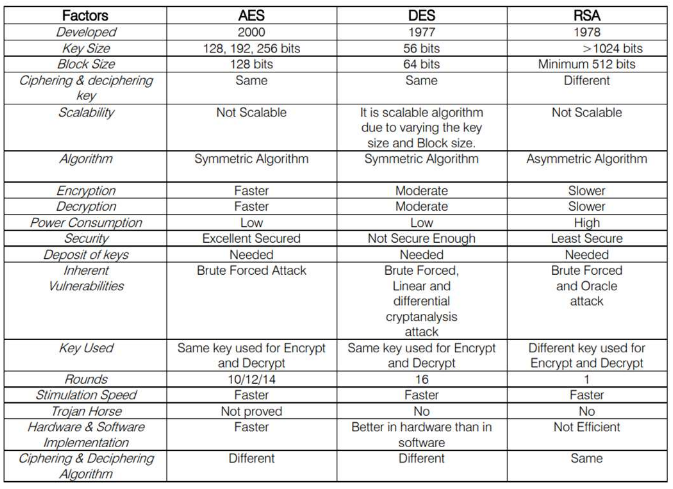
Modes of Operations (ECB and CBC)

- ***ECB (Electronic Codebook Mode):*** This mode encrypts each block of data separately and is noted for its simplicity. However, ECB has serious flaws because identical plaintext blocks produce identical ciphertext blocks, which can reveal information about the material being encrypted.
- ***CBC (Cipher Block Chaining Mode):*** Each plaintext block is XORed with the previous ciphertext block before being encrypted. This method uses an initialization vector (IV) to improve security by assuring that the same plaintext block produces multiple ciphertext blocks under different encryptions. CBC is more secure than ECB because it obscures patterns in plaintext, making it less vulnerable to certain of the flaws found in ECB.

IV. CONCLUSION

This study effectively illustrated the performance consequences of several encryption approaches for Linux inter-process communication via named pipes, taking into account both theoretical and practical elements of each method. The results of the time study, CPU utilization analysis, and theoretical security assessments provide a comprehensive understanding of how various cryptographic algorithms affect IPC performance and security.

The performance analysis shows that simpler encryption methods, such as the Caesar cipher, provide the fastest execution times due to their simplicity, but at the expense of insufficient security measures. This was expected, and it is consistent with theoretical understanding of cryptographic methods[12], where simplicity frequently impairs security. AES, on the other hand, as a modern encryption standard, strikes a strong balance between security and performance, despite the fact that increasing key sizes or new operational modes do not result in considerable performance loss, as was previously assumed. This shows that AES's design can handle increased complexity efficiently, making it appropriate for secure IPC implementations with minimal performance implications.



Factors	AES	DES	RSA
Developed	2000	1977	1978
Key Size	128, 192, 256 bits	56 bits	>1024 bits
Block Size	128 bits	64 bits	Minimum 512 bits
Ciphering & deciphering key	Same	Same	Different
Scalability	Not Scalable	It is scalable algorithm due to varying the key size and Block size.	Not Scalable
Algorithm	Symmetric Algorithm	Symmetric Algorithm	Asymmetric Algorithm
Encryption	Faster	Moderate	Slower
Decryption	Faster	Moderate	Slower
Power Consumption	Low	Low	High
Security	Excellent Secured	Not Secure Enough	Least Secure
Deposit of keys	Needed	Needed	Needed
Inherent Vulnerabilities	Brute Forced Attack	Brute Forced, Linear and differential cryptanalysis attack	Brute Forced and Oracle attack
Key Used	Same key used for Encrypt and Decrypt	Same key used for Encrypt and Decrypt	Different key used for Encrypt and Decrypt
Rounds	10/12/14	16	1
Stimulation Speed	Faster	Faster	Faster
Trojan Horse	Not proved	No	No
Hardware & Software Implementation	Faster	Better in hardware than in software	Not Efficient
Ciphering & Deciphering Algorithm	Different	Different	Same

Figure 25: *Theoretical comparison of AES, DES and RSA methods*[12]

The DES algorithm, despite its historical significance, demonstrates its age and limits during performance tests. The slower execution times correspond to the fact that it is no longer used in modern cryptographic applications, having been replaced by more secure and efficient algorithms like as AES.

CPU utilization metrics provided by the perf stat program demonstrated that both encryption and decryption processes require equivalent resources, confirming the efficacy of the cryptographic operations. The data did not reveal significant differences in performance between the writer and reader processes during automatic execution. This suggests that the Linux operating system effectively manages the overhead introduced by the IPC protocol.

Interestingly, the CPU use data did not differ considerably between different encryption algorithms and operational modes, which contradicted assumptions. This could imply that, for the sake of this project, CPU utilization is not the only or most important aspect in determining the performance of cryptographic algorithms in IPC settings. This observation underscores the importance of considering different performance indicators when evaluating cryptographic solutions.

The security research verified theoretical expectations: AES provides strong security across all modes, with CBC mode outperforming ECB by hiding patterns in ciphertext. RSA remains a viable option for secure data transfer, particularly with suitable key lengths and proper implementation. The Caesar cipher, while not safe, was an effective teaching tool for demonstrating basic cryptography ideas.

Overall, this study emphasizes the significance of choosing appropriate cryptographic algorithms based on the security and efficiency constraints unique to IPC in Linux settings. AES stands out as a particularly effective alternative for safeguarding IPC due to its excellent security guarantees and low performance overhead. Future research could look into the influence of IPC mechanisms other than named pipes, such as Unix domain sockets or shared memory, to gain a better understanding of safe IPC strategies in Linux.

This study also emphasizes the practical value of employing performance and security analysis tools like perf stat and OpenSSL to analyze and optimize IPC systems in a secure and efficient manner. As we continue to rely on sophisticated IPC systems in increasingly security-sensitive applications, the lessons learned from this project will be invaluable in guiding the design and implementation of resilient, efficient, and secure communication systems.

REFERENCES

[1] C/Linux - Server <-> Terminal communication with named pipes. (n.d.). Stack Overflow. https://stackoverflow.com/questions/40923725/c-linux-server-terminal-communication-with-named-pipes

[2] A. (2023, January 29). How Does A Named Pipe (FIFO) Work In Linux? Take The Notes, https://takethenotes.com/how-does-a-named-pipe-fifo-work-in-linux/#:~:text=A%20named%20pipe%2C%20also%20known,traditional%20pipe%20concept%20on%20Linux

[3] Class 13: IPC with pipes. (n.d.). https://www.usna.edu/Users/cs/wcbrown/courses/IC221/classes/L13/Class.html

[4] Computer Security: Art and Science 2nd Edition by Matt Bishop (Author), Addison-Wesley Professional, November 26, 2018.

[5] "Applied Cryptography: Protocols, Algorithms and Source Code in C", 2nd edition, Bruce Schneier, October 18, 1996.

[6] Hioureas, V. (2023, October 13). Encryption 101: How to break encryption. Malwarebytes. https://www.malwarebytes.com/blog/news/2018/03/encryption-101-how-to-break-encryption

[7] Information Security: The Design of Rijndael, The Advanced Encryption Standard (AES), 2nd Edition by Joan Daemen and Vincent Rijmen, 2002.

[8] W. Richard Stevens and Stephen A. Rago, Advanced Programming in the UNIX Environment, 3rd ed. Boston, MA: Addison-Wesley, 2013.

[9] I. Ristić, OpenSSL Cookbook. 1st ed. Zagreb, Croatia: Feisty Duck Ltd., 2014. Available: https://www.feistyduck.com/books/openssl-cookbook/

[10] J. Viega, M. Messier, and P. Chandra, Network Security with OpenSSL. Sebastopol, CA: O'Reilly Media, Inc., 2002.

[11] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th ed. Hoboken, NJ: Pearson, 2017.

[12] Prerna Mahajan and Abhishek Sachdeva: "A Study of Encryption Algorithms AES, DES and RSA for Security", 2013.

APPENDIX

A.1. Writer Process of the Ceasar Algorithm via RSA Secured Key

```
// Ceasar_RSA_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_ceasar"
#define MAX_SIZE 256 // maximum message size

char* encrypt(const char *plaintext, int shift) {
    char *encrypted = (char *)malloc(strlen(plaintext) +
1); // Allocate memory for the encrypted string

    for(int i = 0; plaintext[i] != '\0'; ++i) {
        char ch = plaintext[i];

        if(ch >= 'a' && ch <= 'z') {
            ch = ((ch - 'a') + shift) % 26 + 'a';
        }
        else if(ch >= 'A' && ch <= 'Z') {
            ch = ((ch - 'A') + shift) % 26 + 'A';
        }
        encrypted[i] = ch;
    }
    encrypted[strlen(plaintext)] = '\0'; // Null-
terminate the encrypted string

    return encrypted;
}

int main(){

    int fd;
    srand(time(NULL));
    int shift_key = rand() % 26;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char key[16];
    sprintf(key, "%d", shift_key);

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_ceasar_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_ceasar_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_ceasar_key) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len;

    if (EVP_PKEY_encrypt(ctx_ceasar_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);

    if (EVP_PKEY_encrypt(ctx_ceasar_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key on terminal (will be deleted after)
    printf("Randomly Generated Ceasar Shift Key: %d\n\n",
shift_key);
    //sleep(2);
    printf("Encrypted key is: ");
    for (size_t i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");

    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    // Ceasar Encryption
    unsigned char* encrypted_ceasar = encrypt(basemsg,
shift_key);

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: %s\n", encrypted_ceasar);

    if(write(fd, encrypted_ceasar,
strlen(encrypted_ceasar)) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    close(fd);
    EVP_PKEY_free(pubKey_key);
    free(encrypted_key);
    EVP_cleanup();
    ERR_free_strings();

    return 0;
}
```

A.2. Reader Process of the Ceasar Algorithm via RSA Secured Key

```c
// Ceasar_RSA_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "myfifo_ceasar"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256

char* decrypt(const char *ciphertext, int shift) {
    char *decrypted = (char *)malloc(strlen(ciphertext) + 
1); // Allocate memory for the decrypted string

    for(int i = 0; ciphertext[i] != '\0'; ++i) {
        char ch = ciphertext[i];

        if(ch >= 'a' && ch <= 'z') {
            ch = ((ch - 'a') - shift + 26) % 26 + 'a';
        }
        else if(ch >= 'A' && ch <= 'Z') {
            ch = ((ch - 'A') - shift + 26) % 26 + 'A';
        }
        decrypted[i] = ch;
    }
    decrypted[strlen(ciphertext)] = '\0'; // Null-
terminate the decrypted string

    return decrypted;
}

int main(){
    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];

    memset(received_message, 0, MAX_SIZE);

    // Loading the private key
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key = 
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_key = 
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0) 
ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_message_len = 
strlen(received_message);
    size_t decrypted_key_len, decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL, 
&decrypted_key_len, received_key, received_key_len) <= 0) 
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key = 
malloc(decrypted_key_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key, 
&decrypted_key_len, received_key, received_key_len) <= 0) 
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key and iv on terminal (will be deleted 
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    int decrypted_integer_key = atoi(decrypted_key);
    printf("Decrypted key is: %d\n\n"
,decrypted_integer_key);
    //sleep(2);

    if (read(fd, received_message, 
sizeof(received_message)) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    char* decrypted_message = decrypt(received_message, 
decrypted_integer_key);

    printf("Received text: %s\n\n", received_message);
    printf("Decrypted plain-text: %s\n", 
decrypted_message);

    EVP_PKEY_free(privKey_key);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    free(decrypted_key);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    return 0;
```

A.3. Writer Process of the AES Algorithm via RSA Secured Key

```
// AES_RSA_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "myfifo_aes"
#define MAX_SIZE 256 // maximum message size

int main(){
    // For AES-128
    unsigned char key[16] = {0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C,
0x0D, 0x0E, 0x0F};

    /****************************************************/
    /*
    // For AES-192
    unsigned char key[24] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17
    };
    */
    /****************************************************/
    /*
    // For AES-256
    unsigned char key[32] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
    };
    */

    // For CBC mode
    unsigned char iv[16] = {0xA0, 0xA1, 0xA2, 0xA3, 0xA4,
0xA5, 0xA6, 0xA7, 0xA8, 0xA9, 0xAA, 0xAB, 0xAC, 0xAD,
0xAE, 0xAF};

    int fd;
    int len;
    int aes_len;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char encrypted_aes[MAX_SIZE];

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    /*
    // AES random key and iv generation
    if (!RAND_bytes(key, sizeof(key)) || !RAND_bytes(iv,
sizeof(iv))) {
        perror("RAND_bytes failed.\n");
        return EXIT_FAILURE;
    }
    */

    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    FILE* pubKeyFile_iv = fopen("public2.pem", "rb");
    if (!pubKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_iv = PEM_read_PUBKEY(pubKeyFile_iv,
NULL, NULL, NULL);
    fclose(pubKeyFile_iv);
    if (!pubKey_iv) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    //sleep(2);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(pubKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len, encrypted_iv_len;
    size_t iv_len = sizeof(iv);

    if (EVP_PKEY_encrypt(ctx_rsa_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, NULL,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //leep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);
    unsigned char* encrypted_iv =
malloc(encrypted_iv_len);

    if (EVP_PKEY_encrypt(ctx_rsa_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, encrypted_iv,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key and iv on terminal (will be deleted
after)
    printf("AES Key: ");
    for (int i=0; i<key_len; i++){
        printf("%02X ", key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("AES IV: ");
    for (int i=0; i<iv_len; i++){
        printf("%02X ", iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted key is: ");
    for (int i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted iv is: ");
    for (int i = 0; i < encrypted_iv_len; i++) {
        printf("%02X ", encrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (write(fd, encrypted_iv, encrypted_iv_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    // AES Encryption
    EVP_CIPHER_CTX *ctx_aes = EVP_CIPHER_CTX_new();
    if (!ctx_aes) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    /*
    // For AES-128 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_128_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /****************************************************/
    /*
    // For AES-192 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_192_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /****************************************************/
    /*
    // For AES-256 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_256_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /****************************************************/

    // For AES-128 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_128_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    /****************************************************/
    /*
    // For AES-192 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_192_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /****************************************************/
    /*
    // For AES-256 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_256_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */

    if (1 != EVP_EncryptUpdate(ctx_aes, encrypted_aes,
&len, (unsigned char*)basemsg, strlen(basemsg))) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    aes_len = len;

    if (1 != EVP_EncryptFinal_ex(ctx_aes, encrypted_aes +
len, &len)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    aes_len += len;

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: ");
    for(size_t i = 0; i < aes_len; i++)
        printf("%X%X ", (encrypted_aes[i] >> 4) & 0xf,
encrypted_aes[i] & 0xf);
    printf("\n");

    if(write(fd, encrypted_aes, aes_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    close(fd);
    EVP_PKEY_free(pubKey_key);
    EVP_PKEY_free(pubKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    EVP_CIPHER_CTX_free(ctx_aes);
    free(encrypted_key);
    free(encrypted_iv);
    EVP_cleanup();
    ERR_free_strings();

    return 0;
}
```

A.4. Reader Process of the AES Algorithm via RSA Secured Key

```
// AES_RSA_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "myfifo_aes"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256
#define MAX_IV_SIZE_MODE 256

int main(){
    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_iv[MAX_IV_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];
    unsigned char decrypted_message[MAX_SIZE];

    memset(received_message, 0, MAX_SIZE);
    memset(decrypted_message, 0, MAX_SIZE);

    // Loading the private key 1
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key =
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    // Loading the private key 2
    FILE* privKeyFile_iv = fopen("private2.pem", "rb");
    if (!privKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_iv =
PEM_read_PrivateKey(privKeyFile_iv, NULL, NULL, NULL);
    fclose(privKeyFile_iv);
    if (!privKey_iv) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(privKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_iv_len = sizeof(received_iv);
    size_t received_message_len =
strlen(received_message);
    size_t decrypted_key_len, decrypted_iv_len,
decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (read(fd, received_iv, sizeof(received_iv)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, NULL,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key =
malloc(decrypted_key_len);
    unsigned char* decrypted_iv =
malloc(decrypted_iv_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, decrypted_iv,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key and iv on terminal (will be deleted
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Received iv is: ");
    for(int i=0; i<received_iv_len;i++){
        printf("%02X ", received_iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted key is: ");
    for(int i=0; i<decrypted_key_len;i++){
        printf("%02X ", decrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted iv is: ");
    for(int i=0; i<decrypted_iv_len;i++){
        printf("%02X ", decrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    EVP_CIPHER_CTX *ctx_aes = EVP_CIPHER_CTX_new();

    if (!ctx_aes) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    //sleep(2);
    int len = read(fd, received_message,
sizeof(received_message));
    if (len > 0){
        //sleep(2);
        //if (len>0){
        printf("Received text: ");
        for(int i = 0; i < sizeof(received_message); i++)
            printf("%X%X ", (received_message[i] >> 4) &
0xf, received_message[i] & 0xf);
        printf("\n\n");

        /*
        // For AES-128 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_128_ecb(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/
        /*
        // For AES-192 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_192_ecb(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/
        /*
        // For AES-256 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_256_ecb(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/

        // For AES-128 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_128_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }

        /*************************************************
***/
        /*
        // For AES-192 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_192_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/
        /*
        // For AES-256 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_256_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */

        if (!EVP_DecryptUpdate(ctx_aes,
decrypted_message, &plaintext_len, received_message,
len)) {
            ERR_print_errors_fp(stderr);
        }

        if (!EVP_DecryptFinal_ex(ctx_aes,
decrypted_message + plaintext_len, &final_len)) {
            ERR_print_errors_fp(stderr);
        }

        plaintext_len += final_len;

        decrypted_message[plaintext_len] = '\0'; //
Ensure null terminator to treat as C string
        printf("Length of the message:
%d\n",plaintext_len);
        printf("Decrypted plain-text: %s\n",
decrypted_message);
    }
    EVP_CIPHER_CTX_free(ctx_aes);
    EVP_PKEY_free(privKey_key);
    EVP_PKEY_free(privKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    free(decrypted_key);
    free(decrypted_iv);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    return 0;
}
```

A.5. Writer Process of the DES Algorithm via RSA Secured Key

```
// DES_RSA_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <openssl/provider.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "myfifo_des"
#define MAX_SIZE 256 // maximum message size

int main(){
    OSSL_PROVIDER *legacy_provider = NULL;
    legacy_provider = OSSL_PROVIDER_load(NULL,
"default");
    legacy_provider = OSSL_PROVIDER_load(NULL, "legacy");
    if (legacy_provider == NULL){
        fprintf(stderr, "Failed to load legacy
provider.\n");
        return 1;
    }

    // For DES
    unsigned char key[8] = {0x01, 0x23, 0x45, 0x67, 0x89,
0xAB, 0xCD, 0xEF};

    // For CBC mode
    unsigned char iv[8] = {0x01, 0x23, 0x45, 0x67, 0x89,
0xAB, 0xCD, 0xEF};

    /*
    // DES random key and iv generation
    if (!RAND_bytes(key, sizeof(key)) || !RAND_bytes(iv,
sizeof(iv))) {
        perror("RAND_bytes failed.\n");
        return EXIT_FAILURE;
    }
    */

    int len, fd;
    int ciphertext_len;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char encrypted_des[MAX_SIZE];

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    FILE* pubKeyFile_iv = fopen("public2.pem", "rb");
    if (!pubKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_iv = PEM_read_PUBKEY(pubKeyFile_iv,
NULL, NULL, NULL);
    fclose(pubKeyFile_iv);
    if (!pubKey_iv) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    //sleep(2);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(pubKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len, encrypted_iv_len;
    size_t iv_len = sizeof(iv);

    if (EVP_PKEY_encrypt(ctx_rsa_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, NULL,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //leep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);
    unsigned char* encrypted_iv =
malloc(encrypted_iv_len);

    if (EVP_PKEY_encrypt(ctx_rsa_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, encrypted_iv,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key and iv on terminal (will be deleted
after)
    printf("DES Key: ");
    for (int i=0; i<key_len; i++){
        printf("%02X ", key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("DES IV: ");
    for (int i=0; i<iv_len; i++){
        printf("%02X ", iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted key is: ");
    for (int i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted iv is: ");
    for (int i = 0; i < encrypted_iv_len; i++) {
        printf("%02X ", encrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (write(fd, encrypted_iv, encrypted_iv_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    // DES Encryption

    EVP_CIPHER_CTX *ctx_des = EVP_CIPHER_CTX_new();

    if (!ctx_des) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //
    // For DES (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_des, EVP_des_ecb(),
NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    //

    /*/
    // For DES (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, key, iv)) {
        EVP_CIPHER_CTX_free(ctx_des);
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    /*/

    if (1 != EVP_EncryptUpdate(ctx_des, encrypted_des,
&len, (unsigned char*)basemsg, strlen(basemsg))) {
        EVP_CIPHER_CTX_free(ctx_des);
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    ciphertext_len = len;

    if (1 != EVP_EncryptFinal_ex(ctx_des, encrypted_des +
len, &len)) {
        perror("EVP_EncryptFinal_ex failed.\n");
        EVP_CIPHER_CTX_free(ctx_des);
        close(fd);
        return EXIT_FAILURE;
    }

    ciphertext_len += len;

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: ");
    for(size_t i = 0; i < sizeof(encrypted_des); i++)
        printf("%X%X ", (encrypted_des[i] >> 4) & 0xf,
encrypted_des[i] & 0xf);
    printf("\n");

    if(write(fd, encrypted_des, ciphertext_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    close(fd);
    EVP_PKEY_free(pubKey_key);
    EVP_PKEY_free(pubKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    EVP_CIPHER_CTX_free(ctx_des);
    OSSL_PROVIDER_unload(legacy_provider);
    free(encrypted_key);
    free(encrypted_iv);
    EVP_cleanup();
    ERR_free_strings();

    return 0;
}
```

A.6. Reader Process of the DES Algorithm via RSA Secured Key

```
// DES_RSA_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <openssl/provider.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "myfifo_des"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256
#define MAX_IV_SIZE_MODE 256

int main(){

    OSSL_PROVIDER *legacy_provider = NULL;
    legacy_provider = OSSL_PROVIDER_load(NULL,
"default");
    legacy_provider = OSSL_PROVIDER_load(NULL, "legacy");
    if (legacy_provider == NULL){
        fprintf(stderr, "Failed to load legacy
provider.\n");
        return 1;
    }
    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_iv[MAX_IV_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];
    unsigned char decrypted_message[MAX_SIZE];

    memset(received_message, 0, MAX_SIZE);
    memset(decrypted_message, 0, MAX_SIZE);

    // Loading the private key 1
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key =
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    // Loading the private key 2
    FILE* privKeyFile_iv = fopen("private2.pem", "rb");
    if (!privKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_iv =
PEM_read_PrivateKey(privKeyFile_iv, NULL, NULL, NULL);
    fclose(privKeyFile_iv);
    if (!privKey_iv) ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(privKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_iv_len = sizeof(received_iv);
    size_t received_message_len =
strlen(received_message);
    size_t decrypted_key_len, decrypted_iv_len,
decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (read(fd, received_iv, sizeof(received_iv)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, NULL,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key =
malloc(decrypted_key_len);
    unsigned char* decrypted_iv =
malloc(decrypted_iv_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, decrypted_iv,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    // Printing key and iv on terminal (will be deleted
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Received iv is: ");
    for(int i=0; i<received_iv_len;i++){
        printf("%02X ", received_iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted key is: ");
    for(int i=0; i<decrypted_key_len;i++){
        printf("%02X ", decrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted iv is: ");
    for(int i=0; i<decrypted_iv_len;i++){
        printf("%02X ", decrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // DES Decryption
    EVP_CIPHER_CTX *ctx_des = EVP_CIPHER_CTX_new();

    if (!ctx_des) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    //sleep(2);
    int len = read(fd, received_message,
sizeof(received_message));
    if (len > 0){
        printf("Received text: ");
        for(int i = 0; i < sizeof(received_message); i++)
            printf("%X%X ", (received_message[i] >> 4) &
0xf, received_message[i] & 0xf);
        printf("\n\n");

        //
        // For DES (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_des, EVP_des_ecb(),
NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }
        //

        /*/
        // For DES (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }
        /*/

        if (!EVP_DecryptUpdate(ctx_des,
decrypted_message, &plaintext_len, received_message,
len)) {
            ERR_print_errors_fp(stderr);
        }

        if (!EVP_DecryptFinal_ex(ctx_des,
decrypted_message + plaintext_len, &final_len)) {
            ERR_print_errors_fp(stderr);
        }

        plaintext_len += final_len;

        decrypted_message[plaintext_len] = '\0'; //
Ensure null terminator to treat as C string
        printf("Length of the message:
%d\n",plaintext_len);
        printf("Decrypted plain-text: %s\n",
decrypted_message);
    }
    EVP_CIPHER_CTX_free(ctx_des);
    OSSL_PROVIDER_unload(legacy_provider);
    EVP_PKEY_free(privKey_key);
    EVP_PKEY_free(privKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    free(decrypted_key);
    free(decrypted_iv);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    return 0;
}
```

A.7. Final form of Ceasar writer process

```
// Ceasar_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_ceasar"
#define MAX_SIZE 256 // maximum message size

char* encrypt(const char *plaintext, int shift) {
    char *encrypted = (char *)malloc(strlen(plaintext) +
1); // Allocate memory for the encrypted string

    for(int i = 0; plaintext[i] != '\0'; ++i) {
        char ch = plaintext[i];

        if(ch >= 'a' && ch <= 'z') {
            ch = ((ch - 'a') + shift) % 26 + 'a';
        }
        else if(ch >= 'A' && ch <= 'Z') {
            ch = ((ch - 'A') + shift) % 26 + 'A';
        }
        encrypted[i] = ch;
    }
    encrypted[strlen(plaintext)] = '\0'; // Null-
terminate the encrypted string

    return encrypted;
}

void append_to_csv(const char *filename, long double
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);

    fclose(fp);
}

int main(){
    long double start_rsa_enc_time, end_rsa_enc_time,
start_ceasar_enc_time, end_ceasar_enc_time,
elapsed_rsa_enc_time, elapsed_ceasar_enc_time,
total_duration_1, total_duration_2, total_duration_3,
total_duration_4, total_duration_5, total_duration_6,
total_duration_all;
    total_duration_all = 0;
    total_duration_1 = clock();

    int fd;
    srand(time(NULL));
    int shift_key = rand() % 26;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char key[16];

    const char *filename_1 =
"tables/td_rsakey_enc_ceasar_writer.csv";
    const char *filename_2 =
"tables/td_enc_ceasar_writer.csv";
    const char *filename_3 =
"tables/td_ceasar_writer.csv";

    sprintf(key, "%d", shift_key);

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    start_rsa_enc_time = clock();
    EVP_PKEY_CTX* ctx_ceasar_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_ceasar_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_ceasar_key) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len;

    if (EVP_PKEY_encrypt(ctx_ceasar_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);

    if (EVP_PKEY_encrypt(ctx_ceasar_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_enc_time = clock();

    elapsed_rsa_enc_time = (end_rsa_enc_time -
start_rsa_enc_time)/CLOCKS_PER_SEC;

    // Printing key on terminal (will be deleted after)
    printf("Randomly Generated Ceasar Shift Key: %d\n\n",
shift_key);
    //sleep(2);
    printf("Encrypted key is: ");
    for (size_t i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");

    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    start_ceasar_enc_time = clock();

    // Ceasar Encryption
    unsigned char* encrypted_ceasar = encrypt(basemsg,
shift_key);

    end_ceasar_enc_time = clock();

    elapsed_ceasar_enc_time = (end_ceasar_enc_time -
start_ceasar_enc_time)/CLOCKS_PER_SEC;

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: %s\n\n", encrypted_ceasar);
    printf("RSA Key Encryption Duration: %Lf us.\n\n",
elapsed_rsa_enc_time*1000000);
    printf("Ceasar Encryption Duration: %Lf us.\n\n",
elapsed_ceasar_enc_time*1000000);

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 -
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;

    if(write(fd, encrypted_ceasar,
strlen(encrypted_ceasar)) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_5 = clock();

    close(fd);
    EVP_PKEY_free(pubKey_key);
    free(encrypted_key);
    EVP_cleanup();
    ERR_free_strings();

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 -
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n",
total_duration_all*1000000);

    append_to_csv(filename_1,
elapsed_rsa_enc_time*1000000);
    append_to_csv(filename_2,
elapsed_ceasar_enc_time*1000000);
    append_to_csv(filename_3,
total_duration_all*1000000);

    return 0;
}
```

A.8. Final form of Ceasar reader process

```
// Ceasar_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_ceasar"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256

char* decrypt(const char *ciphertext, int shift) {
    char *decrypted = (char *)malloc(strlen(ciphertext) +
1); // Allocate memory for the decrypted string

    for(int i = 0; ciphertext[i] != '\0'; ++i) {
        char ch = ciphertext[i];

        if(ch >= 'a' && ch <= 'z') {
            ch = ((ch - 'a') - shift + 26) % 26 + 'a';
        }
        else if(ch >= 'A' && ch <= 'Z') {
            ch = ((ch - 'A') - shift + 26) % 26 + 'A';
        }
        decrypted[i] = ch;
    }
    decrypted[strlen(ciphertext)] = '\0'; // Null-
terminate the decrypted string

    return decrypted;
}

void append_to_csv(const char *filename, long double
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);

    fclose(fp);
}

int main(){
    long double start_rsa_dec_time, end_rsa_dec_time,
start_ceasar_dec_time, end_ceasar_dec_time,
elapsed_rsa_dec_time, elapsed_ceasar_dec_time,
total_duration_1, total_duration_2, total_duration_3,
total_duration_4, total_duration_5, total_duration_6,
total_duration_all;
    total_duration_all = 0;
    total_duration_1 = clock();

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];

    const char *filename_1 =
"tables/td_rsakey_dec_ceasar_reader.csv";
    const char *filename_2 =
"tables/td_dec_ceasar_reader.csv";
    const char *filename_3 =
"tables/td_ceasar_reader.csv";

    memset(received_message, 0, MAX_SIZE);

    // Loading the private key
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key =
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_message_len =
strlen(received_message);
    size_t decrypted_key_len, decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);

    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    //sleep(2);

    start_rsa_dec_time = clock();

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key =
malloc(decrypted_key_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_dec_time = clock();

    elapsed_rsa_dec_time = (end_rsa_dec_time -
start_rsa_dec_time)/CLOCKS_PER_SEC;

    // Printing key and iv on terminal (will be deleted
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    int decrypted_integer_key = atoi(decrypted_key);
    printf("Decrypted key is: %d\n\n"
,decrypted_integer_key);
    //sleep(2);

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 -
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;

    if (read(fd, received_message,
sizeof(received_message)) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_5 = clock();

    start_ceasar_dec_time = clock();
    char* decrypted_message = decrypt(received_message,
decrypted_integer_key);

    end_ceasar_dec_time = clock();

    elapsed_ceasar_dec_time = (end_ceasar_dec_time -
start_ceasar_dec_time)/CLOCKS_PER_SEC;

    printf("Received text: %s\n\n", received_message);
    printf("Decrypted plain-text: %s\n\n",
decrypted_message);

    printf("RSA Key Decryption Duration: %Lf us.\n\n",
elapsed_rsa_dec_time*1000000);
    printf("Ceasar Decryption Duration: %Lf us.\n\n",
elapsed_ceasar_dec_time*1000000);

    EVP_PKEY_free(privKey_key);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    free(decrypted_key);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 -
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n",
total_duration_all*1000000);

    append_to_csv(filename_1,
elapsed_rsa_dec_time*1000000);
    append_to_csv(filename_2,
elapsed_ceasar_dec_time*1000000);
    append_to_csv(filename_3,
total_duration_all*1000000);

    return 0;
}
```

A.9. Final form of DES (CBC) writer process

```
// DES_RSA_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <openssl/provider.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_des"
#define MAX_SIZE 256 // maximum message size

void append_to_csv(const char *filename, long double
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);
    fclose(fp);
}

int main(){
    long double start_rsa_enc_time, end_rsa_enc_time,
start_des_enc_time, end_des_enc_time,
elapsed_rsa_enc_time, elapsed_des_enc_time,
total_duration_1, total_duration_2, total_duration_3,
total_duration_4, total_duration_5, total_duration_6,
total_duration_all;
    total_duration_all = 0;
    total_duration_1 = clock();

    OSSL_PROVIDER *legacy_provider = NULL;
    legacy_provider = OSSL_PROVIDER_load(NULL,
"default");
    legacy_provider = OSSL_PROVIDER_load(NULL, "legacy");
    if (legacy_provider == NULL){
        fprintf(stderr, "Failed to load legacy
provider.\n");
        return 1;
    }

    // For DES
    // unsigned char key[8] = {0x01, 0x23, 0x45, 0x67,
0x89, 0xAB, 0xCD, 0xEF};

    // For CBC mode
    // unsigned char iv[8] = {0x01, 0x23, 0x45, 0x67,
0x89, 0xAB, 0xCD, 0xEF};


    // DES random key and iv generation
    unsigned char key[8];
    unsigned char iv[8];
    srand(time(NULL));
    if (!RAND_bytes(key, sizeof(key)) || !RAND_bytes(iv,
sizeof(iv))) {
        perror("RAND_bytes failed.\n");
        return EXIT_FAILURE;
    }


    int len, fd;
    int ciphertext_len;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char encrypted_des[MAX_SIZE];

    const char *filename_1 =
"tables/td_rsakey_enc_des_cbc_writer.csv";
    const char *filename_2 =
"tables/td_enc_des_cbc_writer.csv";
    const char *filename_3 =
"tables/td_des_cbc_writer.csv";

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    FILE* pubKeyFile_iv = fopen("public2.pem", "rb");
    if (!pubKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_iv = PEM_read_PUBKEY(pubKeyFile_iv,
NULL, NULL, NULL);
```

```
    fclose(pubKeyFile_iv);
    if (!pubKey_iv) ERR_print_errors_fp(stderr);

    start_rsa_enc_time = clock();

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    //sleep(2);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(pubKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len, encrypted_iv_len;
    size_t iv_len = sizeof(iv);

    if (EVP_PKEY_encrypt(ctx_rsa_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, NULL,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //leep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);
    unsigned char* encrypted_iv =
malloc(encrypted_iv_len);

    if (EVP_PKEY_encrypt(ctx_rsa_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, encrypted_iv,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_enc_time = clock();

    elapsed_rsa_enc_time = (end_rsa_enc_time -
start_des_enc_time)/CLOCKS_PER_SEC;

    // Printing key and iv on terminal (will be deleted
after)
    printf("Randomly generated DES Key: ");
    for (int i=0; i<key_len; i++){
        printf("%02X ", key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Randomly generated DES IV: ");
    for (int i=0; i<iv_len; i++){
        printf("%02X ", iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted key is: ");
    for (int i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted iv is: ");
    for (int i = 0; i < encrypted_iv_len; i++) {
        printf("%02X ", encrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    if (write(fd, encrypted_iv, encrypted_iv_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    // DES Encryption

    start_des_enc_time = clock();

    EVP_CIPHER_CTX *ctx_des = EVP_CIPHER_CTX_new();

    if (!ctx_des) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //
    // For DES (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    //

    /*
    // For DES (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, key, iv)) {
        EVP_CIPHER_CTX_free(ctx_des);
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }
    /*/
    if (1 != EVP_EncryptUpdate(ctx_des, encrypted_des,
&len, (unsigned char*)basemsg, strlen(basemsg))) {
        EVP_CIPHER_CTX_free(ctx_des);
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    ciphertext_len = len;

    if (1 != EVP_EncryptFinal_ex(ctx_des, encrypted_des +
len, &len)) {
        perror("EVP_EncryptFinal_ex failed.\n");
        EVP_CIPHER_CTX_free(ctx_des);
        close(fd);
        return EXIT_FAILURE;
    }

    ciphertext_len += len;

    end_des_enc_time = clock();

    elapsed_des_enc_time = (end_des_enc_time -
start_des_enc_time)/CLOCKS_PER_SEC;

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: ");
    for(size_t i = 0; i < sizeof(encrypted_des); i++)
        printf("%X%X ", (encrypted_des[i] >> 4) & 0xf,
encrypted_des[i] & 0xf);
    printf("\n\n");

    printf("RSA Key Encryption Duration: %Lf us.\n\n",
elapsed_rsa_enc_time*1000000);
    printf("DES Encryption Duration: %Lf us.\n\n",
elapsed_des_enc_time*1000000);

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 -
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;

    if(write(fd, encrypted_des, ciphertext_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_5 = clock();

    close(fd);
    EVP_PKEY_free(pubKey_key);
    EVP_PKEY_free(pubKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    EVP_CIPHER_CTX_free(ctx_des);
    OSSL_PROVIDER_unload(legacy_provider);
    free(encrypted_key);
    free(encrypted_iv);
    EVP_cleanup();
    ERR_free_strings();

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 -
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n",
total_duration_all*1000000);

    append_to_csv(filename_1,
elapsed_rsa_enc_time*1000000);
    append_to_csv(filename_2,
elapsed_des_enc_time*1000000);
    append_to_csv(filename_3,
total_duration_all*1000000);

    return 0;
}
```

A.10. Final form of DES (CBC) reader process

```
// DES_RSA_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <openssl/provider.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_des"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256
#define MAX_IV_SIZE_MODE 256

void append_to_csv(const char *filename, long double
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);

    fclose(fp);
}

int main(){
    long double start_rsa_dec_time, end_rsa_dec_time,
start_des_dec_time, end_des_dec_time,
elapsed_rsa_dec_time, elapsed_des_dec_time,
total_duration_1, total_duration_2, total_duration_3,
total_duration_4, total_duration_5, total_duration_6,
total_duration_all;
    total_duration_all = 0;
    total_duration_1 = clock();

    OSSL_PROVIDER *legacy_provider = NULL;
    legacy_provider = OSSL_PROVIDER_load(NULL,
"default");
    legacy_provider = OSSL_PROVIDER_load(NULL, "legacy");
    if (legacy_provider == NULL){
        fprintf(stderr, "Failed to load legacy
provider.\n");
        return 1;
    }

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_iv[MAX_IV_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];
    unsigned char decrypted_message[MAX_SIZE];

    const char *filename_1 =
"tables/td_rsakey_dec_des_cbc_reader.csv";
    const char *filename_2 =
"tables/td_dec_des_cbc_reader.csv";
    const char *filename_3 =
"tables/td_des_cbc_reader.csv";

    memset(received_message, 0, MAX_SIZE);
    memset(decrypted_message, 0, MAX_SIZE);

    // Loading the private key 1
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key =
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    // Loading the private key 2
    FILE* privKeyFile_iv = fopen("private2.pem", "rb");
    if (!privKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_iv =
PEM_read_PrivateKey(privKeyFile_iv, NULL, NULL, NULL);
    fclose(privKeyFile_iv);
    if (!privKey_iv) ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_iv_len = sizeof(received_iv);
    size_t received_message_len =
strlen(received_message);
    size_t decrypted_key_len, decrypted_iv_len,
decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);

    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    if (read(fd, received_iv, sizeof(received_iv)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    start_rsa_dec_time = clock();

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(privKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, NULL,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key =
malloc(decrypted_key_len);
    unsigned char* decrypted_iv =
malloc(decrypted_iv_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, decrypted_iv,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_dec_time = clock();

    elapsed_rsa_dec_time = (end_rsa_dec_time -
start_rsa_dec_time)/CLOCKS_PER_SEC;

    // Printing key and iv on terminal (will be deleted
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Received iv is: ");
    for(int i=0; i<received_iv_len;i++){
        printf("%02X ", received_iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted key is: ");
    for(int i=0; i<decrypted_key_len;i++){
        printf("%02X ", decrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted iv is: ");
    for(int i=0; i<decrypted_iv_len;i++){
        printf("%02X ", decrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // DES Decryption
    EVP_CIPHER_CTX *ctx_des = EVP_CIPHER_CTX_new();

    if (!ctx_des) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 -
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;
```

```
    int len = read(fd, received_message,
sizeof(received_message));

    total_duration_5 = clock();

    if (len > 0){
        printf("Received text: ");
        for(int i = 0; i < sizeof(received_message); i++)
            printf("%X%X ", (received_message[i] >> 4) &
0xf, received_message[i] & 0xf);
        printf("\n\n");

        start_des_dec_time = clock();

        //
        // For DES (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }
        //

        /*/
        // For DES (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_des, EVP_des_cbc(),
NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }
        /*/

        if (!EVP_DecryptUpdate(ctx_des,
decrypted_message, &plaintext_len, received_message,
len)) {
            ERR_print_errors_fp(stderr);
        }

        if (!EVP_DecryptFinal_ex(ctx_des,
decrypted_message + plaintext_len, &final_len)) {
            ERR_print_errors_fp(stderr);
        }

        plaintext_len += final_len;

        decrypted_message[plaintext_len] = '\0'; //
Ensure null terminator to treat as C string

        end_des_dec_time = clock();

        elapsed_des_dec_time = (end_des_dec_time -
start_des_dec_time)/CLOCKS_PER_SEC;

        printf("Length of the message:
%d\n\n",plaintext_len);
        printf("Decrypted plain-text: %s\n\n",
decrypted_message);

        printf("RSA Key Decryption Duration: %Lf
us.\n\n", elapsed_rsa_dec_time*1000000);
        printf("DES Decryption Duration: %Lf us.\n\n",
elapsed_des_dec_time*1000000);
    }
    EVP_CIPHER_CTX_free(ctx_des);
    OSSL_PROVIDER_unload(legacy_provider);
    EVP_PKEY_free(privKey_key);
    EVP_PKEY_free(privKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    free(decrypted_key);
    free(decrypted_iv);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 -
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n",
total_duration_all*1000000);

    append_to_csv(filename_1,
elapsed_rsa_dec_time*1000000);
    append_to_csv(filename_2,
elapsed_des_dec_time*1000000);
    append_to_csv(filename_3,
total_duration_all*1000000);


    return 0;
}
```

A.11. Final form of AES-128 (CBC) writer process

```
// AES_writer.c
#include <fcntl.h>
#include <stdio.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_aes"
#define MAX_SIZE 256 // maximum message size

void append_to_csv(const char *filename, long double
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);

    fclose(fp);
}

int main(){
    long double start_rsa_enc_time, end_rsa_enc_time,
start_aes_enc_time, end_aes_enc_time,
elapsed_rsa_enc_time, elapsed_aes_enc_time,
total_duration_1, total_duration_2, total_duration_3,
total_duration_4, total_duration_5, total_duration_6,
total_duration_all;

    total_duration_all = 0;
    total_duration_1 = clock();

    // For AES-128
    // unsigned char key[16] = {0x00, 0x01, 0x02, 0x03,
0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C,
0x0D, 0x0E, 0x0F};

    /****************************************************/
    /*
    // For AES-192
    unsigned char key[24] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17
    };
    */
    /****************************************************/
    /*
    // For AES-256
    unsigned char key[32] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F
    };
    */

    // For CBC mode
    // unsigned char iv[16] = {0xA0, 0xA1, 0xA2, 0xA3,
0xA4, 0xA5, 0xA6, 0xA7, 0xA8, 0xA9, 0xAA, 0xAB, 0xAC,
0xAD, 0xAE, 0xAF};

    int fd;
    int len;
    int aes_len;
    char basemsg[MAX_SIZE] = "This is a plain-text.";
    unsigned char encrypted_aes[MAX_SIZE];

    const char *filename_1 =
"tables/td_rsakey_enc_aes_128cbc_writer.csv";
    const char *filename_2 =
"tables/td_enc_aes_128cbc_writer.csv";
    const char *filename_3 =
"tables/td_aes_128cbc_writer.csv";

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();


    // AES random key and iv generation
    // AES-128: 16
    // AES-192: 24
    // AES-256: 32
    unsigned char key[16];
    unsigned char iv[16];
    srand(time(NULL));
    if (!RAND_bytes(key, sizeof(key)) || !RAND_bytes(iv,
sizeof(iv))) {
        perror("RAND_bytes failed.\n");
        return EXIT_FAILURE;
    }


    // Loading the public key
    FILE* pubKeyFile_key = fopen("public.pem", "rb");
    if (!pubKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_key =
PEM_read_PUBKEY(pubKeyFile_key, NULL, NULL, NULL);
    fclose(pubKeyFile_key);
    if (!pubKey_key) ERR_print_errors_fp(stderr);

    FILE* pubKeyFile_iv = fopen("public2.pem", "rb");
    if (!pubKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* pubKey_iv = PEM_read_PUBKEY(pubKeyFile_iv,
NULL, NULL, NULL);
    fclose(pubKeyFile_iv);
    if (!pubKey_iv) ERR_print_errors_fp(stderr);

    start_rsa_enc_time = clock();
```

```
    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(pubKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    //sleep(2);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(pubKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_encrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    // Encrypting key and iv
    size_t key_len = sizeof(key);
    size_t encrypted_key_len, encrypted_iv_len;
    size_t iv_len = sizeof(iv);

    if (EVP_PKEY_encrypt(ctx_rsa_key, NULL,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, NULL,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //leep(2);

    unsigned char* encrypted_key =
malloc(encrypted_key_len);
    unsigned char* encrypted_iv =
malloc(encrypted_iv_len);

    if (EVP_PKEY_encrypt(ctx_rsa_key, encrypted_key,
&encrypted_key_len, key, key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_encrypt(ctx_rsa_iv, encrypted_iv,
&encrypted_iv_len, iv, iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_enc_time = clock();

    elapsed_rsa_enc_time = (end_rsa_enc_time -
start_rsa_enc_time)/CLOCKS_PER_SEC;

    // Printing key and iv on terminal (will be deleted
after)
    printf("Randomly generated AES Key: ");
    for (int i=0; i<key_len; i++){
        printf("%02X ", key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Randomly generated AES IV: ");
    for (int i=0; i<iv_len; i++){
        printf("%02X ", iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted key is: ");
    for (int i = 0; i < encrypted_key_len; i++) {
        printf("%02X ", encrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Encrypted iv is: ");
    for (int i = 0; i < encrypted_iv_len; i++) {
        printf("%02X ", encrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // Creating named pipe if it does not exist
    mkfifo(FIFO_FILE, 0640);

    // Writing key and iv to named pipe
    fd = open(FIFO_FILE, O_WRONLY);

    if (fd == -1){
        ERR_print_errors_fp(stderr);
        return EXIT_FAILURE;
    }

    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (write(fd, encrypted_key, encrypted_key_len) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    //sleep(2);

    if (write(fd, encrypted_iv, encrypted_iv_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    // AES Encryption

    start_aes_enc_time = clock();

    EVP_CIPHER_CTX *ctx_aes = EVP_CIPHER_CTX_new();
    if (!ctx_aes) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    /*
    // For AES-128 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_128_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /*****************************************************/
    /*
    // For AES-192 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_192_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /*****************************************************/
    /*
    // For AES-256 (in ECB mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_256_ecb(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /*****************************************************/

    // For AES-128 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_128_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    /*****************************************************/
    /*
    // For AES-192 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_192_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */
    /*****************************************************/
    /*
    // For AES-256 (in CBC mode)
    if (1 != EVP_EncryptInit_ex(ctx_aes,
EVP_aes_256_cbc(), NULL, key, iv)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }
    */

    if (1 != EVP_EncryptUpdate(ctx_aes, encrypted_aes,
&len, (unsigned char*)basemsg, strlen(basemsg))) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    aes_len = len;

    if (1 != EVP_EncryptFinal_ex(ctx_aes, encrypted_aes +
len, &len)) {
        ERR_print_errors_fp(stderr);
        EVP_CIPHER_CTX_free(ctx_aes);
        close(fd);
        return EXIT_FAILURE;
    }

    aes_len += len;

    end_aes_enc_time = clock();

    elapsed_aes_enc_time = (end_aes_enc_time -
start_aes_enc_time)/CLOCKS_PER_SEC;

    printf("Plain text: %s\n\n", basemsg);
    printf("Encrypted text: ");
    for(size_t i = 0; i < aes_len; i++)
```

```
        printf("%X%X ", (encrypted_aes[i] >> 4) & 0xf, 
encrypted_aes[i] & 0xf);
    printf("\n\n");

    printf("RSA Key Encryption Duration: %Lf us.\n\n", 
elapsed_rsa_enc_time*1000000);
    printf("AES Encryption Duration: %Lf us.\n\n", 
elapsed_aes_enc_time*1000000);

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 - 
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;

    if(write(fd, encrypted_aes, aes_len) == -1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_5 = clock();

    close(fd);
    EVP_PKEY_free(pubKey_key);
    EVP_PKEY_free(pubKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    EVP_CIPHER_CTX_free(ctx_aes);
    free(encrypted_key);
    free(encrypted_iv);
    EVP_cleanup();
    ERR_free_strings();

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 - 
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n", 
total_duration_all*1000000);

    append_to_csv(filename_1, 
elapsed_rsa_enc_time*1000000);
    append_to_csv(filename_2, 
elapsed_aes_enc_time*1000000);
    append_to_csv(filename_3, 
total_duration_all*1000000);

    return 0;
}
```

A.12. Final form of AES-128 (CBC) reader process

```
// AES_reader.c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <openssl/evp.h>
#include <openssl/rand.h>
#include <openssl/pem.h>
#include <openssl/err.h>
#include <unistd.h>
#include <string.h>
#include <time.h>

#define FIFO_FILE "myfifo_aes"
#define MAX_SIZE 256 // maximum message size
#define MAX_KEY_SIZE_MODE 256
#define MAX_IV_SIZE_MODE 256
```

```
void append_to_csv(const char *filename, long double 
value){
    FILE *fp = fopen(filename, "a");
    if (!fp){
        ERR_print_errors_fp(stderr);
    }

    fprintf(fp, "%Lf\n", value);

    fclose(fp);
}

int main(){
    long double start_rsa_dec_time, end_rsa_dec_time, 
start_aes_dec_time, end_aes_dec_time, 
elapsed_rsa_dec_time, elapsed_aes_dec_time, 
total_duration_1, total_duration_2, total_duration_3, 
total_duration_4, total_duration_5, total_duration_6, 
total_duration_all;
    total_duration_all = 0;
    total_duration_1 = clock();

    // Initializing OpenSSL
    ERR_load_crypto_strings();
    OpenSSL_add_all_algorithms();

    int fd;
    int plaintext_len, final_len;

    unsigned char received_key[MAX_KEY_SIZE_MODE];
    unsigned char received_iv[MAX_IV_SIZE_MODE];
    unsigned char received_message[MAX_SIZE];
    unsigned char decrypted_message[MAX_SIZE];

    const char *filename_1 = 
"tables/td_rsakey_dec_aes_128cbc_reader.csv";
    const char *filename_2 = 
"tables/td_dec_aes_128cbc_reader.csv";
    const char *filename_3 = 
"tables/td_aes_128cbc_reader.csv";

    memset(received_message, 0, MAX_SIZE);
    memset(decrypted_message, 0, MAX_SIZE);

    // Loading the private key 1
    FILE* privKeyFile_key = fopen("private.pem", "rb");
    if (!privKeyFile_key) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_key = 
PEM_read_PrivateKey(privKeyFile_key, NULL, NULL, NULL);
    fclose(privKeyFile_key);
    if (!privKey_key) ERR_print_errors_fp(stderr);

    // Loading the private key 2
    FILE* privKeyFile_iv = fopen("private2.pem", "rb");
    if (!privKeyFile_iv) ERR_print_errors_fp(stderr);
    EVP_PKEY* privKey_iv = 
PEM_read_PrivateKey(privKeyFile_iv, NULL, NULL, NULL);
    fclose(privKeyFile_iv);
    if (!privKey_iv) ERR_print_errors_fp(stderr);

    // Encrypted key and iv from the writer
    size_t received_key_len = sizeof(received_key);
    size_t received_iv_len = sizeof(received_iv);
    size_t received_message_len = 
strlen(received_message);
    size_t decrypted_key_len, decrypted_iv_len, 
decrypted_message_len;

    fd = open(FIFO_FILE, O_RDONLY);
    if (fd == -1) ERR_print_errors_fp(stderr);
    total_duration_2 = clock();
    total_duration_all = (total_duration_2 -
total_duration_1)/CLOCKS_PER_SEC + total_duration_all;

    if (read(fd, received_key, sizeof(received_key)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    if (read(fd, received_iv, sizeof(received_iv)) == -
1){
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_3 = clock();

    start_rsa_dec_time = clock();

    EVP_PKEY_CTX* ctx_rsa_key =
EVP_PKEY_CTX_new(privKey_key, NULL);
    if (!ctx_rsa_key) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_key) <= 0)
ERR_print_errors_fp(stderr);

    EVP_PKEY_CTX* ctx_rsa_iv =
EVP_PKEY_CTX_new(privKey_iv, NULL);
    if (!ctx_rsa_iv) ERR_print_errors_fp(stderr);
    if (EVP_PKEY_decrypt_init(ctx_rsa_iv) <= 0)
ERR_print_errors_fp(stderr);

    if (EVP_PKEY_decrypt(ctx_rsa_key, NULL,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, NULL,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    unsigned char* decrypted_key =
malloc(decrypted_key_len);
    unsigned char* decrypted_iv =
malloc(decrypted_iv_len);

    if (EVP_PKEY_decrypt(ctx_rsa_key, decrypted_key,
&decrypted_key_len, received_key, received_key_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);
    if (EVP_PKEY_decrypt(ctx_rsa_iv, decrypted_iv,
&decrypted_iv_len, received_iv, received_iv_len) <= 0)
ERR_print_errors_fp(stderr);
    //sleep(2);

    end_rsa_dec_time = clock();

    elapsed_rsa_dec_time = (end_rsa_dec_time -
start_rsa_dec_time)/CLOCKS_PER_SEC;

    // Printing key and iv on terminal (will be deleted
after)
    printf("Received key is: ");
    for(int i=0; i<received_key_len;i++){
        printf("%02X ", received_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Received iv is: ");
    for(int i=0; i<received_iv_len;i++){
        printf("%02X ", received_iv[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted key is: ");
    for(int i=0; i<decrypted_key_len;i++){
        printf("%02X ", decrypted_key[i]);
    }
    printf("\n\n");
    //sleep(2);
    printf("Decrypted iv is: ");
    for(int i=0; i<decrypted_iv_len;i++){
        printf("%02X ", decrypted_iv[i]);
    }
    printf("\n\n");
    //sleep(2);

    // AES Decryption

    EVP_CIPHER_CTX *ctx_aes = EVP_CIPHER_CTX_new();

    if (!ctx_aes) {
        ERR_print_errors_fp(stderr);
        close(fd);
        return EXIT_FAILURE;
    }

    total_duration_4 = clock();
    total_duration_all = (total_duration_4 -
total_duration_3)/CLOCKS_PER_SEC + total_duration_all;

    int len = read(fd, received_message,
sizeof(received_message));

    total_duration_5 = clock();

    if (len > 0){
        //if (len>0){
        printf("Received text: ");
        for(int i = 0; i < sizeof(received_message); i++)
            printf("%X%X ", (received_message[i] >> 4) &
0xf, received_message[i] & 0xf);
        printf("\n\n");

        start_aes_dec_time = clock();

        /*
        // For AES-128 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_128_ecb(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/
        /*
        // For AES-192 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_192_ecb(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /*************************************************
***/
        /*
        // For AES-256 (in ECB mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_256_ecb(), NULL, decrypted_key, decrypted_iv)) {
```

```
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /************************************************
***/

        // For AES-128 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_128_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
        }

        /************************************************
***/
        /*
        // For AES-192 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_192_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */
        /************************************************
***/
        /*
        // For AES-256 (in CBC mode)
        if (!EVP_DecryptInit_ex(ctx_aes,
EVP_aes_256_cbc(), NULL, decrypted_key, decrypted_iv)) {
            ERR_print_errors_fp(stderr);
            close(fd);
        }
        */

        if (!EVP_DecryptUpdate(ctx_aes,
decrypted_message, &plaintext_len, received_message,
len)) {
            ERR_print_errors_fp(stderr);
        }

        if (!EVP_DecryptFinal_ex(ctx_aes,
decrypted_message + plaintext_len, &final_len)) {
            ERR_print_errors_fp(stderr);
        }

        plaintext_len += final_len;

        decrypted_message[plaintext_len] = '\0'; //
Ensure null terminator to treat as C string

        end_aes_dec_time = clock();

        elapsed_aes_dec_time = (end_aes_dec_time -
start_aes_dec_time)/CLOCKS_PER_SEC;

        printf("Length of the message:
%d\n\n",plaintext_len);
        printf("Decrypted plain-text: %s\n\n",
decrypted_message);

        printf("RSA Key Decryption Duration: %Lf
us.\n\n", elapsed_rsa_dec_time*1000000);
        printf("AES Decryption Duration: %Lf us.\n\n",
elapsed_aes_dec_time*1000000);
    }

    EVP_CIPHER_CTX_free(ctx_aes);
    EVP_PKEY_free(privKey_key);
    EVP_PKEY_free(privKey_iv);
    EVP_PKEY_CTX_free(ctx_rsa_key);
    EVP_PKEY_CTX_free(ctx_rsa_iv);
    free(decrypted_key);
    free(decrypted_iv);
    EVP_cleanup();
    ERR_free_strings();
    close(fd);

    total_duration_6 = clock();
    total_duration_all = (total_duration_6 -
total_duration_5)/CLOCKS_PER_SEC + total_duration_all;
    printf("Total Duration: %Lf us.\n\n",
total_duration_all*1000000);

    append_to_csv(filename_1,
elapsed_rsa_dec_time*1000000);
    append_to_csv(filename_2,
elapsed_aes_dec_time*1000000);
    append_to_csv(filename_3,
total_duration_all*1000000);

    return 0;
}
```

A.13. Bash script for timing analysis

```
#!/bin/bash
sleep 1
# Loop for running each pair of writer-reader scripts
1000 times
for i in {1..1000}; do
    ./ceasar_w &
    ./ceasar_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes128_cbc_w &
    ./aes128_cbc_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes128_ecb_w &
    ./aes128_ecb_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes192_cbc_w &
    ./aes192_cbc_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes192_ecb_w &
    ./aes192_ecb_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes256_cbc_w &
    ./aes256_cbc_r
    # sleep 1
done

for i in {1..1000}; do
    ./aes256_ecb_w &
    ./aes256_ecb_r
    # sleep 1
done
```

```
for i in {1..1000}; do
    ./des_cbc_w &
    ./des_cbc_r
    # sleep 1
done

for i in {1..1000}; do
    ./des_ecb_w &
    ./des_ecb_r
    # sleep 1
done

sleep 1
# Run the averager script after all the loops are done
./averager
sleep 1
python3 plotter.py
```

A.14. C code for averaging timing datas

```
#include <stdio.h>
#include <stdlib.h>

// Function to compute the average of values in a CSV
file.
long double compute_average_from_csv(const char*
filename) {
    FILE* file;
    long double value;
    long double count = 0.0;
    long double sum = 0.0;

    // Open the CSV file for reading.
    file = fopen(filename, "r");
    if (file == NULL) {
        perror("Unable to open the file");
        return EXIT_FAILURE;
    }

    // Read values from the file, assuming they're
separated by commas or newlines.
    while (fscanf(file, "%Lf,", &value) == 1) {
        sum += value;
        count++;
    }

    fclose(file);

    // Compute the average. Avoid division by zero.
    if (count == 0) {
        return 0.0;
    }

    return sum / count;
}

void append_to_csv(const char *filename, const char
*category1, const char *category2, const char *category3,
int value) {
    FILE *file = fopen(filename, "a"); // Open the file
in append mode
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }
    fprintf(file, "%s,%s,%s,%d\n", category1, category2,
category3, value); // Write category and value to the
file
    fclose(file); // Close the file
}

int main() {
    // Ceasar
    const char* filename_1 =
"tables/td_rsakey_enc_ceasar_writer.csv";
    const char* filename_2 =
"tables/td_rsakey_dec_ceasar_reader.csv";
    const char* filename_3 =
"tables/td_enc_ceasar_writer.csv";
    const char* filename_4 =
"tables/td_dec_ceasar_reader.csv";
    const char* filename_5 =
"tables/td_ceasar_writer.csv";
    const char* filename_6 =
"tables/td_ceasar_reader.csv";

    // AES-128 (CBC)
    const char* filename_7 =
"tables/td_rsakey_enc_aes_128cbc_writer.csv";
    const char* filename_8 =
"tables/td_rsakey_dec_aes_128cbc_reader.csv";
    const char* filename_9 =
"tables/td_enc_aes_128cbc_writer.csv";
    const char* filename_10 =
"tables/td_dec_aes_128cbc_reader.csv";
    const char* filename_11 =
"tables/td_aes_128cbc_writer.csv";
    const char* filename_12 =
"tables/td_aes_128cbc_reader.csv";

    // AES-128 (ECB)
    const char* filename_13 =
"tables/td_rsakey_enc_aes_128ecb_writer.csv";
    const char* filename_14 =
"tables/td_rsakey_dec_aes_128ecb_reader.csv";
    const char* filename_15 =
"tables/td_enc_aes_128ecb_writer.csv";
    const char* filename_16 =
"tables/td_dec_aes_128ecb_reader.csv";
    const char* filename_17 =
"tables/td_aes_128ecb_writer.csv";
    const char* filename_18 =
"tables/td_aes_128ecb_reader.csv";

    // AES-192 (CBC)
    const char* filename_19 =
"tables/td_rsakey_enc_aes_192cbc_writer.csv";
    const char* filename_20 =
"tables/td_rsakey_dec_aes_192cbc_reader.csv";
    const char* filename_21 =
"tables/td_enc_aes_192cbc_writer.csv";
    const char* filename_22 =
"tables/td_dec_aes_192cbc_reader.csv";
    const char* filename_23 =
"tables/td_aes_192cbc_writer.csv";
    const char* filename_24 =
"tables/td_aes_192cbc_reader.csv";

    // AES-192 (ECB)
    const char* filename_25 =
"tables/td_rsakey_enc_aes_192ecb_writer.csv";
    const char* filename_26 =
"tables/td_rsakey_dec_aes_192ecb_reader.csv";
    const char* filename_27 =
"tables/td_enc_aes_192ecb_writer.csv";
    const char* filename_28 =
"tables/td_dec_aes_192ecb_reader.csv";
    const char* filename_29 =
"tables/td_aes_192ecb_writer.csv";
    const char* filename_30 =
"tables/td_aes_192ecb_reader.csv";
```

```
    // AES-256 (CBC)
    const char* filename_31 =
"tables/td_rsakey_enc_aes_256cbc_writer.csv";
    const char* filename_32 =
"tables/td_rsakey_dec_aes_256cbc_reader.csv";
    const char* filename_33 =
"tables/td_enc_aes_256cbc_writer.csv";
    const char* filename_34 =
"tables/td_dec_aes_256cbc_reader.csv";
    const char* filename_35 =
"tables/td_aes_256cbc_writer.csv";
    const char* filename_36 =
"tables/td_aes_256cbc_reader.csv";

    // AES-256 (ECB)
    const char* filename_37 =
"tables/td_rsakey_enc_aes_256ecb_writer.csv";
    const char* filename_38 =
"tables/td_rsakey_dec_aes_256ecb_reader.csv";
    const char* filename_39 =
"tables/td_enc_aes_256ecb_writer.csv";
    const char* filename_40 =
"tables/td_dec_aes_256ecb_reader.csv";
    const char* filename_41 =
"tables/td_aes_256ecb_writer.csv";
    const char* filename_42 =
"tables/td_aes_256ecb_reader.csv";

    // DES (CBC)
    const char* filename_43 =
"tables/td_rsakey_enc_des_cbc_writer.csv";
    const char* filename_44 =
"tables/td_rsakey_dec_des_cbc_reader.csv";
    const char* filename_45 =
"tables/td_enc_des_cbc_writer.csv";
    const char* filename_46 =
"tables/td_dec_des_cbc_reader.csv";
    const char* filename_47 =
"tables/td_des_cbc_writer.csv";
    const char* filename_48 =
"tables/td_des_cbc_reader.csv";

    // DES (ECB)
    const char* filename_49 =
"tables/td_rsakey_enc_des_cbc_writer.csv";
    const char* filename_50 =
"tables/td_rsakey_dec_des_cbc_reader.csv";
    const char* filename_51 =
"tables/td_enc_des_cbc_writer.csv";
    const char* filename_52 =
"tables/td_dec_des_cbc_reader.csv";
    const char* filename_53 =
"tables/td_des_cbc_writer.csv";
    const char* filename_54 =
"tables/td_des_cbc_reader.csv";

    // Ceasar          1-6
    // AES-128 (CBC)   7-12
    // AES-128 (ECB)   13-18
    // AES-192 (CBC)   19-24
    // AES-192 (ECB)   25-30
    // AES-256 (CBC)   31-36
    // AES-256 (ECB)   37-42
    // DES (CBC)       43-48
    // DES (ECB)       49-54

    long double average_1 =
compute_average_from_csv(filename_1);
    long double average_2 =
compute_average_from_csv(filename_2);
    long double average_3 =
compute_average_from_csv(filename_3);
    long double average_4 =
compute_average_from_csv(filename_4);
    long double average_5 =
compute_average_from_csv(filename_5);
    long double average_6 =
compute_average_from_csv(filename_6);
    long double average_7 =
compute_average_from_csv(filename_7);
    long double average_8 =
compute_average_from_csv(filename_8);
    long double average_9 =
compute_average_from_csv(filename_9);
    long double average_10 =
compute_average_from_csv(filename_10);
    long double average_11 =
compute_average_from_csv(filename_11);
    long double average_12 =
compute_average_from_csv(filename_12);
    long double average_13 =
compute_average_from_csv(filename_13);
    long double average_14 =
compute_average_from_csv(filename_14);
    long double average_15 =
compute_average_from_csv(filename_15);
    long double average_16 =
compute_average_from_csv(filename_16);
    long double average_17 =
compute_average_from_csv(filename_17);
    long double average_18 =
compute_average_from_csv(filename_18);
    long double average_19 =
compute_average_from_csv(filename_19);
    long double average_20 =
compute_average_from_csv(filename_20);
    long double average_21 =
compute_average_from_csv(filename_21);
    long double average_22 =
compute_average_from_csv(filename_22);
    long double average_23 =
compute_average_from_csv(filename_23);
    long double average_24 =
compute_average_from_csv(filename_24);
    long double average_25 =
compute_average_from_csv(filename_25);
    long double average_26 =
compute_average_from_csv(filename_26);
    long double average_27 =
compute_average_from_csv(filename_27);
    long double average_28 =
compute_average_from_csv(filename_28);
    long double average_29 =
compute_average_from_csv(filename_29);
    long double average_30 =
compute_average_from_csv(filename_30);
    long double average_31 =
compute_average_from_csv(filename_31);
    long double average_32 =
compute_average_from_csv(filename_32);
    long double average_33 =
compute_average_from_csv(filename_33);
    long double average_34 =
compute_average_from_csv(filename_34);
    long double average_35 =
compute_average_from_csv(filename_35);
    long double average_36 =
compute_average_from_csv(filename_36);
    long double average_37 =
compute_average_from_csv(filename_37);
```

```
    long double average_38 =
compute_average_from_csv(filename_38);
    long double average_39 =
compute_average_from_csv(filename_39);
    long double average_40 =
compute_average_from_csv(filename_40);
    long double average_41 =
compute_average_from_csv(filename_41);
    long double average_42 =
compute_average_from_csv(filename_42);
    long double average_43 =
compute_average_from_csv(filename_43);
    long double average_44 =
compute_average_from_csv(filename_44);
    long double average_45 =
compute_average_from_csv(filename_45);
    long double average_46 =
compute_average_from_csv(filename_46);
    long double average_47 =
compute_average_from_csv(filename_47);
    long double average_48 =
compute_average_from_csv(filename_48);
    long double average_49 =
compute_average_from_csv(filename_49);
    long double average_50 =
compute_average_from_csv(filename_50);
    long double average_51 =
compute_average_from_csv(filename_51);
    long double average_52 =
compute_average_from_csv(filename_52);
    long double average_53 =
compute_average_from_csv(filename_53);
    long double average_54 =
compute_average_from_csv(filename_54);

    append_to_csv("tables/data.csv", "Ceasar", "None",
"RSA_key_enc", average_1);
    append_to_csv("tables/data.csv", "Ceasar", "None",
"RSA_key_dec", average_2);
    append_to_csv("tables/data.csv", "Ceasar", "None",
"enc", average_3);
    append_to_csv("tables/data.csv", "Ceasar", "None",
"dec", average_4);
    append_to_csv("tables/data.csv", "Ceasar", "None",
"writer", average_5);
    append_to_csv("tables/data.csv", "Ceasar", "None",
"reader", average_6);

    append_to_csv("tables/data.csv", "AES-128", "CBC",
"RSA_key_enc", average_7);
    append_to_csv("tables/data.csv", "AES-128", "CBC",
"RSA_key_dec", average_8);
    append_to_csv("tables/data.csv", "AES-128", "CBC",
"enc", average_9);
    append_to_csv("tables/data.csv", "AES-128", "CBC",
"dec", average_10);
    append_to_csv("tables/data.csv", "AES-128", "CBC",
"writer", average_11);
    append_to_csv("tables/data.csv", "AES-128", "CBC",
"reader", average_12);

    append_to_csv("tables/data.csv", "AES-128", "ECB",
"RSA_key_enc", average_13);
    append_to_csv("tables/data.csv", "AES-128", "ECB",
"RSA_key_dec", average_14);
    append_to_csv("tables/data.csv", "AES-128", "ECB",
"enc", average_15);
    append_to_csv("tables/data.csv", "AES-128", "ECB",
"dec", average_16);
    append_to_csv("tables/data.csv", "AES-128", "ECB",
"writer", average_17);
    append_to_csv("tables/data.csv", "AES-128", "ECB",
"reader", average_18);

    append_to_csv("tables/data.csv", "AES-192", "CBC",
"RSA_key_enc", average_19);
    append_to_csv("tables/data.csv", "AES-192", "CBC",
"RSA_key_dec", average_20);
    append_to_csv("tables/data.csv", "AES-192", "CBC",
"enc", average_21);
    append_to_csv("tables/data.csv", "AES-192", "CBC",
"dec", average_22);
    append_to_csv("tables/data.csv", "AES-192", "CBC",
"writer", average_23);
    append_to_csv("tables/data.csv", "AES-192", "CBC",
"reader", average_24);

    append_to_csv("tables/data.csv", "AES-192", "ECB",
"RSA_key_enc", average_25);
    append_to_csv("tables/data.csv", "AES-192", "ECB",
"RSA_key_dec", average_26);
    append_to_csv("tables/data.csv", "AES-192", "ECB",
"enc", average_27);
    append_to_csv("tables/data.csv", "AES-192", "ECB",
"dec", average_28);
    append_to_csv("tables/data.csv", "AES-192", "ECB",
"writer", average_29);
    append_to_csv("tables/data.csv", "AES-192", "ECB",
"reader", average_30);

    append_to_csv("tables/data.csv", "AES-256", "CBC",
"RSA_key_enc", average_31);
    append_to_csv("tables/data.csv", "AES-256", "CBC",
"RSA_key_dec", average_32);
    append_to_csv("tables/data.csv", "AES-256", "CBC",
"enc", average_33);
    append_to_csv("tables/data.csv", "AES-256", "CBC",
"dec", average_34);
    append_to_csv("tables/data.csv", "AES-256", "CBC",
"writer", average_35);
    append_to_csv("tables/data.csv", "AES-256", "CBC",
"reader", average_36);

    append_to_csv("tables/data.csv", "AES-256", "ECB",
"RSA_key_enc", average_37);
    append_to_csv("tables/data.csv", "AES-256", "ECB",
"RSA_key_dec", average_38);
    append_to_csv("tables/data.csv", "AES-256", "ECB",
"enc", average_39);
    append_to_csv("tables/data.csv", "AES-256", "ECB",
"dec", average_40);
    append_to_csv("tables/data.csv", "AES-256", "ECB",
"writer", average_41);
    append_to_csv("tables/data.csv", "AES-256", "ECB",
"reader", average_42);

    append_to_csv("tables/data.csv", "DES", "CBC",
"RSA_key_enc", average_43);
    append_to_csv("tables/data.csv", "DES", "CBC",
"RSA_key_dec", average_44);
    append_to_csv("tables/data.csv", "DES", "CBC", "enc",
average_45);
    append_to_csv("tables/data.csv", "DES", "CBC", "dec",
average_46);
    append_to_csv("tables/data.csv", "DES", "CBC",
"writer", average_47);
    append_to_csv("tables/data.csv", "DES", "CBC",
"reader", average_48);

    append_to_csv("tables/data.csv", "DES", "ECB",
"RSA_key_enc", average_49);
```

```
    append_to_csv("tables/data.csv", "DES", "ECB",
"RSA_key_dec", average_50);
    append_to_csv("tables/data.csv", "DES", "ECB", "enc",
average_51);
    append_to_csv("tables/data.csv", "DES", "ECB", "dec",
average_52);
    append_to_csv("tables/data.csv", "DES", "ECB",
"writer", average_53);
    append_to_csv("tables/data.csv", "DES", "ECB",
"reader", average_54);

    return 0;
}
```

A.15. Python script for timing plots

```
import pandas as pd
import matplotlib.pyplot as plt

# Read data from CSV
df = pd.read_csv("tables/data.csv", header=None)

# Define the parameters and their indexes for subplots (2
rows and 3 columns of plots)
parameters = ["RSA_key_enc", "RSA_key_dec", "enc", "dec",
"writer", "reader"]
n_rows = 2
n_cols = 3

# Create a figure with subplots
fig, axs = plt.subplots(n_rows, n_cols, figsize=(15, 10))

# Flatten the array of axes for easy iteration
axs = axs.flatten()

for i, param in enumerate(parameters):
    # Filter data for each parameter
    param_data = df[df.iloc[:, 2] == param]  # Selecting
the 3rd column
    # Combine method and mode for x-axis labels
    x_labels = param_data.iloc[:, 0].astype(str) + " (" +
param_data.iloc[:, 1].astype(str) + ")"
    # Plot on the i-th subplot
    axs[i].bar(x_labels, param_data.iloc[:, 3])
    axs[i].set_title(f"{param} Duration Comparison")
    axs[i].set_xlabel("Encryption Method (Mode)")
    axs[i].set_ylabel("Duration (microseconds)")
    axs[i].tick_params(labelrotation=90)  # Rotate the x-
axis labels for readability

# Adjust the layout so the subplots fit into the figure
nicely
plt.tight_layout()
# Save the figure with all subplots
plt.savefig('all_parameters_duration_comparison.png')
plt.close()  # Close the figure after saving to file
```

A.16. Bash script for CPU analysis

```
#!/bin/bash

# Function to capture CPU utilization and append to file
capture_cpu_utilization() {
    local command="$1"
    local file="$2"
    taskset -c 0 perf stat $command 2>&1 | awk -F'[ %]+'
'/CPUs utilized/ {gsub(",", ".", $6); print $6}' >>
"$file"
}
```

```
for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./ceasar_w"
"tables/cpu_ceasar_writer.txt" &
    capture_cpu_utilization "./ceasar_r"
"tables/cpu_ceasar_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes128_cbc_w"
"tables/cpu_aes128_cbc_writer.txt" &
    capture_cpu_utilization "./aes128_cbc_r"
"tables/cpu_aes128_cbc_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes128_ecb_w"
"tables/cpu_aes128_ecb_writer.txt" &
    capture_cpu_utilization "./aes128_ecb_r"
"tables/cpu_aes128_ecb_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes192_cbc_w"
"tables/cpu_aes192_cbc_writer.txt" &
    capture_cpu_utilization "./aes192_cbc_r"
"tables/cpu_aes192_cbc_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes192_ecb_w"
"tables/cpu_aes192_ecb_writer.txt" &
    capture_cpu_utilization "./aes192_ecb_r"
"tables/cpu_aes192_ecb_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes256_cbc_w"
"tables/cpu_aes256_cbc_writer.txt" &
    capture_cpu_utilization "./aes256_cbc_r"
"tables/cpu_aes256_cbc_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./aes256_ecb_w"
"tables/cpu_aes256_ecb_writer.txt" &
    capture_cpu_utilization "./aes256_ecb_r"
"tables/cpu_aes256_ecb_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./des_cbc_w"
"tables/cpu_des_cbc_writer.txt" &
    capture_cpu_utilization "./des_cbc_r"
"tables/cpu_des_cbc_reader.txt"
done

for i in {1..1000}; do
    echo "Iteration $i"
    capture_cpu_utilization "./des_ecb_w"
"tables/cpu_des_ecb_writer.txt" &
    capture_cpu_utilization "./des_ecb_r"
"tables/cpu_des_ecb_reader.txt"
```

```
done

echo "All iterations completed."

sleep 1
./cpu_averager

sleep 1
python3 cpu_plotter.py
```

A.17. C code for averaging CPU utilization datas

```
#include <stdio.h>
#include <stdlib.h>

// Function to compute the average of values in a CSV
file.
long double compute_average_from_txt(const char*
filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        perror("Unable to open the file");
        return -1; // Using -1 to indicate an error
condition
    }

    long double value, sum = 0.0;
    int count = 0;
    char line[128]; // Buffer to hold each line

    // Read each line and extract the CPU utilization
value
    while (fgets(line, sizeof(line), file)) {
        value = atof(line);
        if (value > 2){
            value = 0.5;
        }
        sum += value;
        count++;
    }

    fclose(file);

    if (count == 0) {
        fprintf(stderr, "No valid data found in the
file.\n");
        return 0;
    }

    return 1000*sum/count; // Return the average
}

void append_to_csv(const char *filename, const char
*category1, const char *category2, const char *category3,
int value) {
    FILE *file = fopen(filename, "a"); // Open the file
in append mode
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }
    fprintf(file, "%s,%s,%s,%d\n", category1, category2,
category3, value); // Write category and value to the
file
    fclose(file); // Close the file
}

int main() {
    // Ceasar
    const char* filename_1 =
"tables/cpu_ceasar_writer.txt";
    const char* filename_2 =
"tables/cpu_ceasar_reader.txt";
    // AES-128 (CBC)
    const char* filename_3 =
"tables/cpu_aes128_cbc_writer.txt";
    const char* filename_4 =
"tables/cpu_aes128_cbc_reader.txt";
    // AES-128 (ECB)
    const char* filename_5 =
"tables/cpu_aes128_ecb_writer.txt";
    const char* filename_6 =
"tables/cpu_aes128_ecb_reader.txt";
    // AES-192 (CBC)
    const char* filename_7 =
"tables/cpu_aes192_cbc_writer.txt";
    const char* filename_8 =
"tables/cpu_aes192_cbc_reader.txt";
    // AES-192 (ECB)
    const char* filename_9 =
"tables/cpu_aes192_ecb_writer.txt";
    const char* filename_10 =
"tables/cpu_aes192_ecb_reader.txt";
    // AES-256 (CBC)
    const char* filename_11 =
"tables/cpu_aes256_cbc_writer.txt";
    const char* filename_12 =
"tables/cpu_aes256_cbc_reader.txt";
    // AES-256 (ECB)
    const char* filename_13 =
"tables/cpu_aes256_ecb_writer.txt";
    const char* filename_14 =
"tables/cpu_aes256_ecb_reader.txt";
    // DES (CBC)
    const char* filename_15 =
"tables/cpu_des_cbc_writer.txt";
    const char* filename_16 =
"tables/cpu_des_cbc_reader.txt";
    // DES (ECB)
    const char* filename_17 =
"tables/cpu_des_ecb_writer.txt";
    const char* filename_18 =
"tables/cpu_des_ecb_reader.txt";

    long double average_1 =
compute_average_from_txt(filename_1);
    long double average_2 =
compute_average_from_txt(filename_2);
    long double average_3 =
compute_average_from_txt(filename_3);
    long double average_4 =
compute_average_from_txt(filename_4);
    long double average_5 =
compute_average_from_txt(filename_5);
    long double average_6 =
compute_average_from_txt(filename_6);
    long double average_7 =
compute_average_from_txt(filename_7);
    long double average_8 =
compute_average_from_txt(filename_8);
    long double average_9 =
compute_average_from_txt(filename_9);
    long double average_10 =
compute_average_from_txt(filename_10);
    long double average_11 =
compute_average_from_txt(filename_11);
    long double average_12 =
compute_average_from_txt(filename_12);
    long double average_13 =
compute_average_from_txt(filename_13);
```

```
    long double average_14 =
compute_average_from_txt(filename_14);
    long double average_15 =
compute_average_from_txt(filename_15);
    long double average_16 =
compute_average_from_txt(filename_16);
    long double average_17 =
compute_average_from_txt(filename_17);
    long double average_18 =
compute_average_from_txt(filename_18);


    append_to_csv("tables/cpu_data.csv", "Ceasar",
"None", "writer", average_1);
    append_to_csv("tables/cpu_data.csv", "Ceasar",
"None", "reader", average_2);

    append_to_csv("tables/cpu_data.csv", "AES-128",
"CBC", "writer", average_3);
    append_to_csv("tables/cpu_data.csv", "AES-128",
"CBC", "reader", average_4);

    append_to_csv("tables/cpu_data.csv", "AES-128",
"ECB", "writer", average_5);
    append_to_csv("tables/cpu_data.csv", "AES-128",
"ECB", "reader", average_6);

    append_to_csv("tables/cpu_data.csv", "AES-192",
"CBC", "writer", average_7);
    append_to_csv("tables/cpu_data.csv", "AES-192",
"CBC", "reader", average_8);

    append_to_csv("tables/cpu_data.csv", "AES-192",
"ECB", "writer", average_9);
    append_to_csv("tables/cpu_data.csv", "AES-192",
"ECB", "reader", average_10);

    append_to_csv("tables/cpu_data.csv", "AES-256",
"CBC", "writer", average_11);
    append_to_csv("tables/cpu_data.csv", "AES-256",
"CBC", "reader", average_12);

    append_to_csv("tables/cpu_data.csv", "AES-256",
"ECB", "writer", average_13);
    append_to_csv("tables/cpu_data.csv", "AES-256",
"ECB", "reader", average_14);

    append_to_csv("tables/cpu_data.csv", "DES", "CBC",
"writer", average_15);
    append_to_csv("tables/cpu_data.csv", "DES", "CBC",
"reader", average_16);

    append_to_csv("tables/cpu_data.csv", "DES", "ECB",
"writer", average_17);
    append_to_csv("tables/cpu_data.csv", "DES", "ECB",
"reader", average_18);

    return 0;
}
```

A.18. Python script for CPU utilization plots

```
import pandas as pd
import matplotlib.pyplot as plt

# Read data from CSV
df = pd.read_csv("tables/cpu_data.csv", header=None)

# Define the parameters and their indexes for subplots (2
rows and 3 columns of plots)
parameters = ["writer", "reader"]
n_rows = 1
n_cols = 2

# Create a figure with subplots
fig, axs = plt.subplots(n_rows, n_cols, figsize=(15, 10))

# Flatten the array of axes for easy iteration
axs = axs.flatten()

for i, param in enumerate(parameters):
    # Filter data for each parameter
    param_data = df[df.iloc[:, 2] == param]  # Selecting
the 3rd column
    # Combine method and mode for x-axis labels
    x_labels = param_data.iloc[:, 0].astype(str) + " (" +
param_data.iloc[:, 1].astype(str) + ")"
    # Plot on the i-th subplot
    axs[i].bar(x_labels, param_data.iloc[:, 3]/10)
    axs[i].set_title(f"{param} CPU Utilization")
    axs[i].set_xlabel("Encryption Method (Mode)")
    axs[i].set_ylabel("CPU Utilization (%)")
    axs[i].tick_params(labelrotation=90)  # Rotate the x-
axis labels for readability

# Adjust the layout so the subplots fit into the figure
nicely
plt.tight_layout()
# Save the figure with all subplots
plt.savefig('all_parameters_cpu_utilization_comparison.pn
g')
plt.close()  # Close the figure after saving to file
```

NOTE: 128, 192, 256, ECB and CBC modes can be varied by changing only the inner function definitions in the EVP_EncryptInit_ex(.) or EVP_DecryptInit_ex(.) functions, So there is no need to add all variations of C codes to the Appendix.