HSM-based Performance Analysis of Cryptographic Algorithms

Guide: Indu Radhakrishnan

Name	**SRN**
Ruthvik V	PES1UG21EC227
S Sricharan	PES1UG21EC232
Samarth Jayanth	PES1UG21EC241

Abstract

In the realm of secure communications, Hardware Security Modules (HSMs) play a pivotal role by providing robust cryptographic operations and secure key management. HSMs are dedicated hardware devices designed to protect cryptographic keys and perform encryption and decryption within a secure environment, thereby ensuring the integrity and confidentiality of sensitive data. This project aims to implement an HSM using SoftHSM2 and OpenSC software, focusing on generating cryptographic keys and performing encryption and decryption operations. Our objective is to evaluate and compare the performance of three widely used cryptographic algorithms: Advanced Encryption Standard (AES), Triple Data Encryption Standard (DES3), and Rivest-Shamir-Adleman (RSA).

The evaluation criteria for this comparative analysis include encryption throughput, decryption throughput, key generation time, memory usage, and CPU usage. These metrics provide a comprehensive understanding of the efficiency and resource consumption of each algorithm within the HSM environment.

By integrating SoftHSM2, a software-based HSM, we simulate the functionality of a physical HSM, enabling secure key generation, storage, and cryptographic operations. Through methodical experimentation and analysis, this project will provide valuable insights into the performance trade-offs associated with each algorithm, thereby guiding the selection of appropriate cryptographic techniques for various security applications. The findings will contribute to the optimization of cryptographic operations in HSMs, enhancing the overall security and performance of encrypted communications.

Table of Contents

Introduction

In the current world where cyber threats and data breaches are on the rise, the need for robust security solutions has become paramount. As a result, the adoption of Hardware Security Modules (HSMs) has significantly increased due to their strong security features. An HSM is a tamper-resistant and intrusion-resistant, highly-trusted physical computing device designed to perform various cryptographic operations, such as key management, key exchange, encryption, and decryption. HSMs are built on specialized hardware, equipped with security-focused operating systems, and controlled access interfaces, making them highly secure and trusted components within a network.

HSMs play a crucial role in protecting cryptographic keys and ensuring that these keys remain secure throughout their lifecycle. One of the most critical functions of an HSM is key generation, which involves securely creating cryptographic keys used for encryption and decryption purposes. This is achieved through built-in true random number generators (TRNGs), ensuring the randomness and unpredictability of the keys. The keys generated and stored within an HSM are protected from unauthorized access and tampering, providing the highest level of security for sensitive data and cryptographic keys on the market.

The application of HSMs spans various industries due to their robust security capabilities. In the financial sector, HSMs are used to encrypt transaction data, manage payment card information, and secure online banking systems. Healthcare organizations use HSMs to safeguard patient records and ensure compliance with regulations like HIPAA. Government and defense industries rely on HSMs to encrypt classified information and secure communication channels. Additionally, e-

commerce platforms use HSMs to protect customer data and ensure secure online transactions. The deployment of HSMs ensures that cryptographic operations are performed in a secure and isolated environment, minimizing the risk of data breaches and unauthorized access.

HSMs are not only vital for ensuring the security of cryptographic operations but also for improving system performance. By offloading cryptographic tasks from the main servers to dedicated HSM devices, organizations can reduce the computational load on their systems, leading to improved performance and efficiency. HSMs also support a wide range of cryptographic algorithms, including AES, DES3, and RSA, making them versatile tools for various security applications.

In this project, we theoretically analyze cryptographic algorithms (AES, DES3, and RSA) and implement C++ programs to compare their performance based on metrics such as Encryption Throughput, Decryption Throughput, Key Generation Time, Memory Usage, and CPU Usage. This comparative study aims to provide insights into the efficiency and effectiveness of these algorithms when implemented using HSMs, particularly in terms of their application in real-world security scenarios.

Problem Statement

As cyber threats and data breaches become more sophisticated, ensuring the security of cryptographic operations and key management is paramount. Traditional software-based security systems are often vulnerable to tampering and unauthorized access, leading to potential compromises of sensitive data.

Hardware Security Modules (HSMs) offer a solution by providing a tamper-resistant, highly secure environment for executing critical cryptographic tasks. HSMs ensure that keys are generated, stored, and managed in a way that is isolated from the rest of the system, making them less susceptible to attacks.

However, while HSMs provide enhanced security, evaluating their performance across various cryptographic algorithms remains crucial. This evaluation helps determine the efficiency and effectiveness of HSMs when using different cryptographic algorithms, ensuring that the benefits of enhanced security do not come at the cost of significant performance degradation.

In this context, it becomes necessary to assess and compare the performance of widely used cryptographic algorithms—AES, DES3, and RSA—when implemented using HSMs. Each of these algorithms has its strengths and weaknesses in terms of encryption and decryption throughput, key generation time, memory usage, and CPU consumption.

Understanding these performance metrics is critical for organizations to make strategic decisions about which algorithm best suits their security needs while balancing performance requirements. This problem is particularly relevant for industries where security is paramount, such as finance, healthcare, and government, where data integrity and confidentiality are non-negotiable. Therefore, the analysis and comparison of these algorithms within the secure environment of HSMs will provide valuable insights into the trade-offs between security robustness and operational efficiency.

Background and Related Work

SoftHSM2 is an open source software implementation of a cryptographic store, using PKCS#11 APIs it emulates an HSM environment which is usable for development and testing. With SoftHSM2 being an emulation of the actual HSM, it is possible for developers and organizations to consider the advantages of using HSMs without the associated costs and logistical challenges.

The PKCS#11 (Public-Key Cryptography Standards #11) is a well known standard for a platform independent interface for cryptographic tokens like HSM and smart cards. This API is used to enable applications that require cryptographic services to run these without having to know the details of the underlying hardware. Thus, it serves as an interface for managing keys, as well as for encryption and decryption processes, which makes it a foundation for building secure applications.

In PKCS#11 framework there are two basic concepts that are slots and tokens. A slot can be a physical or logical reader that can hold a token. A token, on the other hand, is the physical entity or the gadget that can hold the cryptographic keys and can also process the cryptographic operations. In the case of SoftHSM2 the slots are virtual within the software and the tokens are also virtual which mimic the functionality of physical cryptographic tokens. This setup enables the developers to simulate the functionality of hardware HSMs and hence helps them to develop and test the applications in a more effective way.

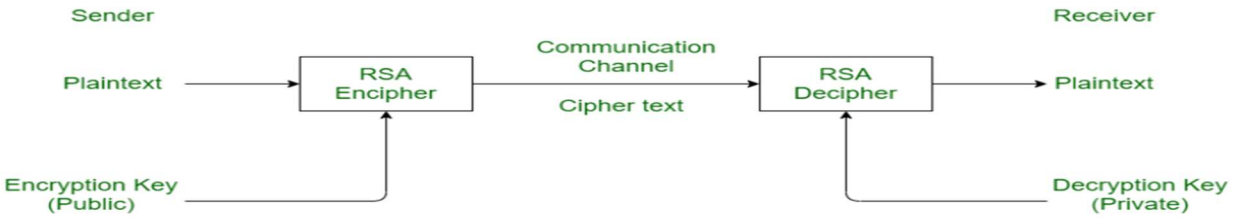
Confidentiality and integrity of data are crucial in communication to enhance security within the networks especially in the applications that involve personal information or any transaction. This process involves several critical components, with key generation and encryption being vital. Key generation involves securely creating cryptographic keys that are used for encryption and decryption purposes. These keys are so important in providing secure means for the exchange of communication to avoid security breaches.

In this context, virtual HSMs like SoftHSM2 play a critical role by securely managing cryptographic keys within a software-based environment. SoftHSM2 provides a flexible and cost-effective solution to generate, store, and manage keys securely, facilitating encryption and decryption processes while adhering to industry standards and compliance requirements. By integrating SoftHSM2 into their security infrastructure, organizations can enhance the confidentiality and integrity of their client-server communication. This integration helps mitigate risks associated with cyber threats, ensuring that sensitive information remains protected throughout its lifecycle.

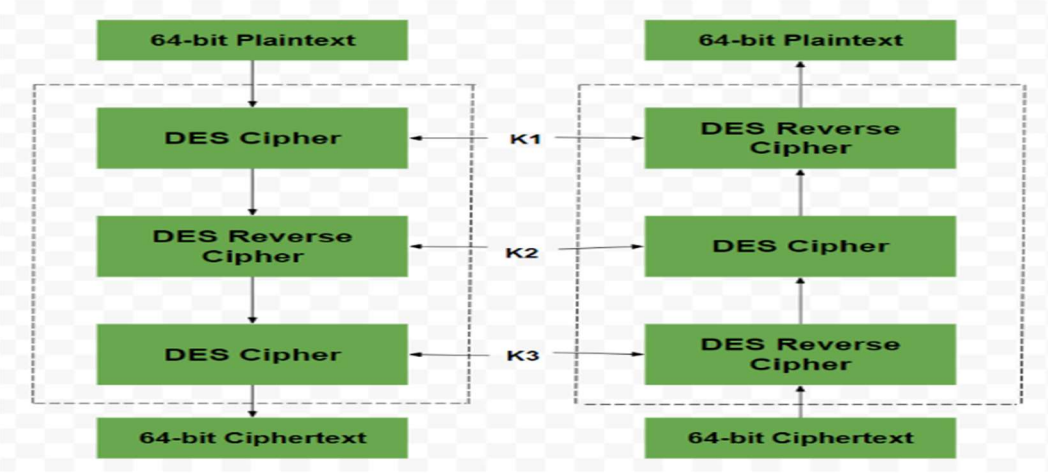
RSA stands for Rivest-Shamir-Adleman, an asymmetric-key encryption algorithm, first proposed in 1977, using a pair of keys: one public and the other private, for accomplishing the processes of encryption and its subsequent decryption. It often finds huge applications in secure data transmission, generating digital signatures, and key exchange. RSA generates a public key for encryption and a private key for its decryption. In this process, two large prime numbers p and q are selected, and their product n and Euler's totient function $\phi(n)$ are

calculated. A public exponent e is chosen, and a private exponent d is derived as the modular inverse of e modulo $\phi(n)$.

Encrypting plaintext M into ciphertext C = (m^e) mod n, while on the other hand, the decryption recovers M using M = (c^d) mod n. The block diagram for the RSA algorithm is shown below.

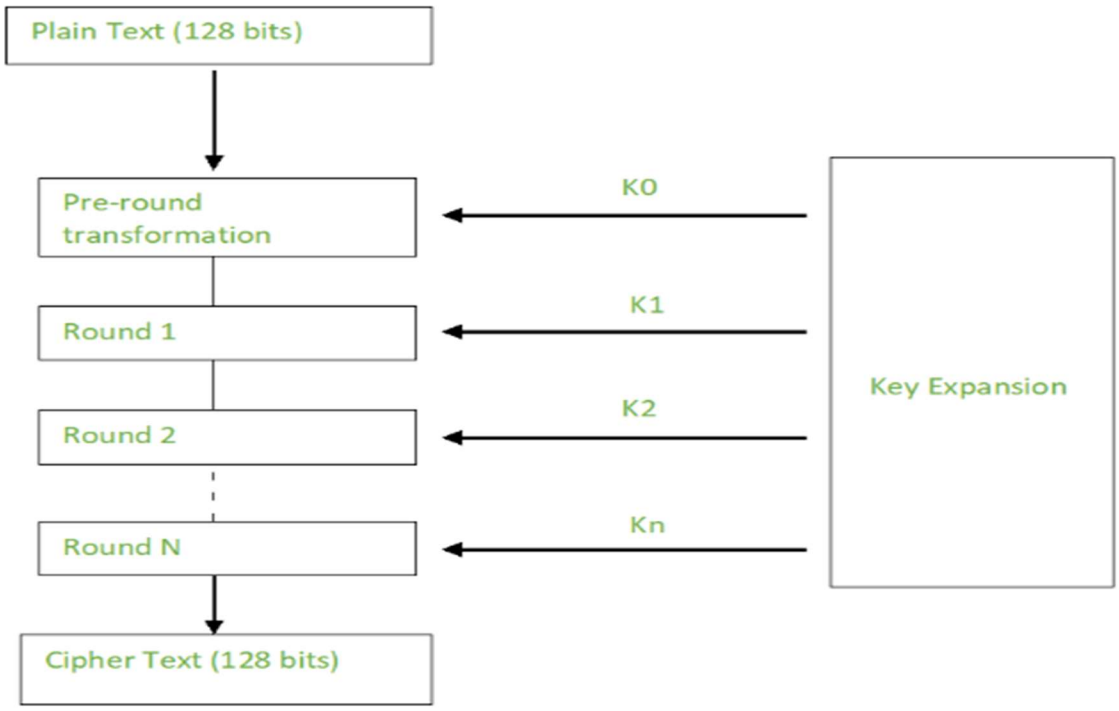


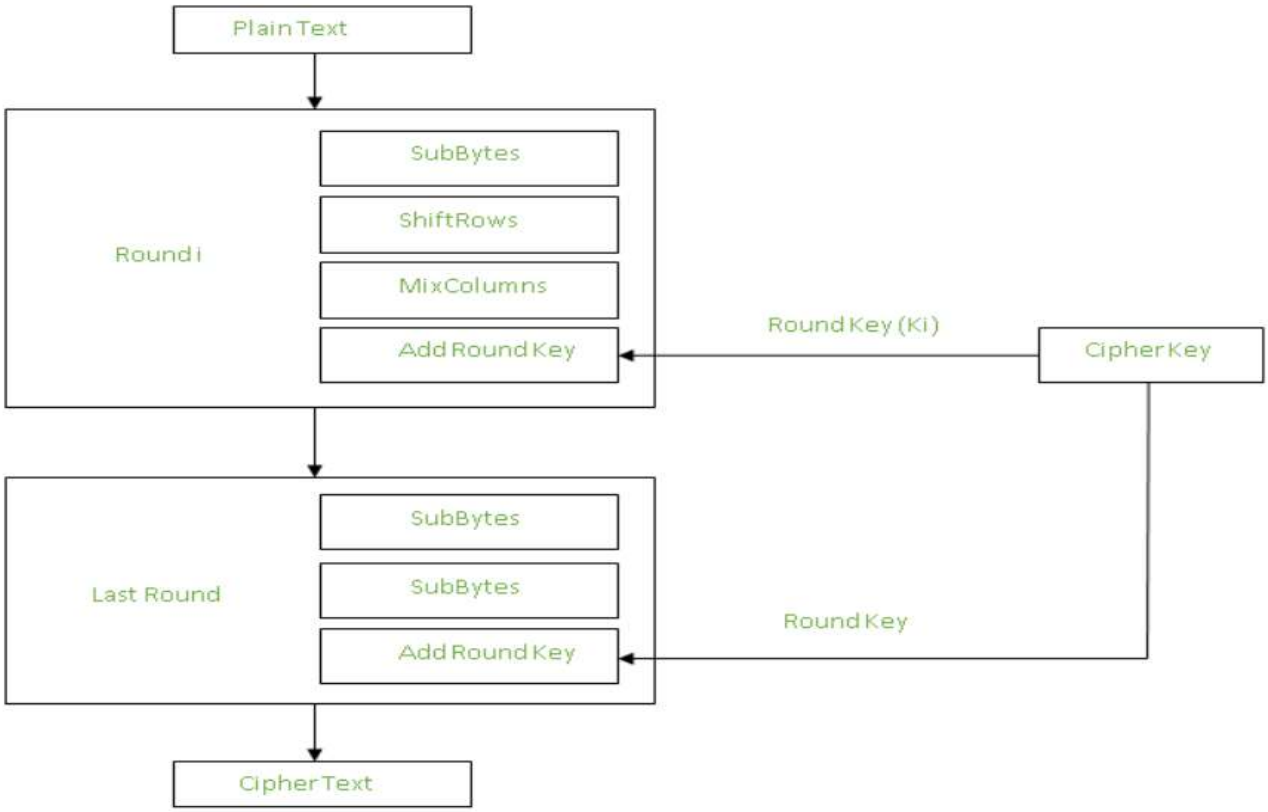
3DES is a symmetric key block cipher. The security of the original DES algorithm is improved by applying it three times to each block of data. Due to the additional safety provided by this method, it has been used in many applications where more security is needed than DES provides. DES3 triple-encrypts every block of plaintext using the DES cipher algorithm. Triple DES works with a packet of three 56-bit keys—that is, a total of 168 bits. The triple DES process undergoes three stages: an encryption step with the first key, followed by a decryption step with the second key, and finally another encryption with the third key. This triple application of DES makes 3DES much more secure compared to its predecessor and minimizes several vulnerabilities associated with the smaller key size of the single DES standard. The block diagram for the DES3 algorithm is shown below.



Triple DES

AES is the Advanced Encryption Standard, established in 2001, and it is also a symmetric key block cipher known for its speed and security with 128-bit block sizes and three key sizes of 128, 192, and 256 bits. It has wide applications in secure data encryption. AES represents a substitution of plaintext into the ciphertext under multiple substitution and permutation operations organized for several rounds according to the following conditions: 10 rounds for keys of 128 bits, 12 rounds for 192-bit keys, and finally 14 rounds for keys of 256 bits. Every round involves several steps: SubBytes that is byte substitution based on an S-box; ShiftRows for permutation by rows; MixColumns for column mixing; AddRoundKey for bitwise the operation of a round key, directly from the encryption key. Decryption is the inverse process. The block diagrams for the AES algorithm are shown below.


Plain Text (128 bits)
Pre-round transformation
Round 1
Round 2
Round N
Cipher Text (128 bits)
K0
K1
K2
Kn
Key Expansion


Plain Text
Round i
SubBytes
ShiftRows
MixColumns
Add Round Key
Round Key (Ki)
Cipher Key
Last Round
SubBytes
SubBytes
Add Round Key
Round Key
Cipher Text

When comparing these algorithms by examining their block diagrams and methodologies, certain observations can be made. AES is recognized for its impressive speed and robust security. It is more complex than some older algorithms, but its security, efficiency and flexibility with varying key lengths makes it widely applicable.
3DES, on the other hand, enhances security through triple encryption while retaining compatibility with DES, with a relatively small effective key length.
RSA stands out for its high security, yet it is slower than symmetric-key algorithms due to its extensive mathematical computations and its larger key sizes.

Methodology

In this section, we discuss the process followed to compare the cryptographic algorithms [RSA,DES3 and AES] based on performance metrics such as Encryption Throughput, Decryption Throughput, Key Generation Time, Memory Usage, CPU Usage using SoftHSM2. The steps involved are critical to ensuring the secure generation, management, and utilization of cryptographic keys and encryption-decryption process.

SoftHSM2 was employed to initialize slots and tokens. These slots and tokens are essential components in interacting with the virtual Hardware Security Module (HSM). Here, the slots represent logical entities that hold the tokens, which are the actual cryptographic modules responsible for storing and managing keys.

The process began with setting up the slots and tokens using SoftHSM2. Each slot was initialized to hold a token, which then simulated the functionality of a physical HSM. This setup allowed for a realistic emulation of hardware-based security within a software environment, making it possible to develop and test the cryptographic operations required for secure communication.

The C++ program was designed to perform several key tasks, leveraging the PKCS#11 API provided by SoftHSM2. The following steps outline the methodology:

Global Variables and Function Declarations :- At the top of the example, it declares several global variables: handles for the PKCS#11 library and session, slot ID, PIN for a slot, and buffers to hold encrypted and decrypted data. It also includes function declarations on loading the HSM library, freeing of resources, printing data in hexadecimal and plain text formats, connecting to and disconnecting from a slot, generating cryptographic keys, encrypting and decrypting data, measuring key generation, encryption, and decryption times, obtaining current memory and CPU usage, and calculating throughput for a given operation.

```
#include <iostream>
#include <cryptoki.h>
#include <windows.h>
#include <psapi.h>
#include <ctime>
#include <cstdlib>
#include <cstring>

using namespace std;

// Global variables
HINSTANCE libHandle = nullptr;
CK_FUNCTION_LIST_PTR p11Func = nullptr;
CK_SLOT_ID slotId = 0;
CK_SESSION_HANDLE hSession = 0;
CK_BYTE *slotPin = nullptr;
const char *libPath = nullptr;
CK_OBJECT_HANDLE objHandle = 0;
CK_BYTE IV[] = "1234567812345678";
unsigned char plainData[] = "Earth is the third planet of our Solar System.";
CK_BYTE *encryptedData = nullptr;
CK_BYTE *decryptedData = nullptr;
CK_ULONG encLen = 0;
CK_ULONG decLen = 0;
```

```
// Function declarations
void loadHSMLibrary();
void freeResource();
void printHex(CK_BYTE *bytes, int len);
void printData(const char* label, CK_BYTE *data, CK_ULONG len);
void checkOperation(CK_RV rv, const char *message);
void connectToSlot();
void disconnectFromSlot();
void generateAesKey();
void encryptData();
void decryptData();
void measureKeyGenerationTime();
void measureEncryptionTime();
void measureDecryptionTime();
size_t getCurrentMemoryUsage();
double getCurrentCpuUsage();
void calculateThroughput(void (*operation)(), const char* operationName, int numOperations)
```

Loading the HSM Library :- The loadHSMLibrary function loads the PKCS#11 library identified by the P11 LIB environment variable, gets a function list from the library, and saves it in the global p11Func variable. If the loading of the library or the function list fails, the function prints an error message and exits the program.

```
// Function to load PKCS#11 library
void loadHSMLibrary()
{
    libPath = getenv("P11_LIB");
    if (libPath == nullptr)
    {
        cout << "P11_LIB environment variable not set." << endl;
        exit(1);
    }

    libHandle = LoadLibrary(libPath);
    if (libHandle == nullptr)
    {
        cout << "Failed to load P11 library: " << libPath << endl;
        exit(1);
    }

    CK_C_GetFunctionList C_GetFunctionList = (CK_C_GetFunctionList)GetProcAddress(libHandle, "C_GetFunctionList");
    if (C_GetFunctionList == nullptr)
    {
        cout << "Failed to load P11 Functions." << endl;
        exit(1);
    }

    C_GetFunctionList(&p11Func);
    if (p11Func == nullptr)
    {
        cout << "Failed to get function list." << endl;
        exit(1);
    }
}
```

Free Resource :- This function is used to free allocated encrypted data and decrypted data, in order to prevent memory leaks.

```
// Function to free allocated resources
void freeResource()
{
    if (libHandle)
        FreeLibrary(libHandle);

    p11Func = nullptr;
    slotPin = nullptr;
    delete[] encryptedData;
    delete[] decryptedData;
}
```

PrintHex :- This function is used to print the text in hexadecimal format.

```
// Function to print byte array as hex string
void printHex(CK_BYTE *bytes, int len)
{
    for (int ctr = 0; ctr < len; ctr++)
    {
        printf("%02x", bytes[ctr]);
    }
    cout << endl;
}
```

PrintData :- This function is used to print the text in plaintext format.

```
// Function to print data (hexadecimal and plain text)
void printData(const char* label, CK_BYTE *data, CK_ULONG len)
{
    cout << label << " (Hex): ";
    printHex(data, len);

    cout << label << " (Plain Text): ";
    for (CK_ULONG i = 0; i < len; ++i)
    {
        cout << static_cast<char>(data[i]);
    }
    cout << endl;
}
```

CheckOperation :- This function checks if the operation fails and prints a message defining the reason for failure along with a return code.

```
// Function to check if PKCS#11 operation was successful
void checkOperation(CK_RV rv, const char *message)
{
    if (rv != CKR_OK)
    {
        cout << message << " failed with: " << rv << endl;
        freeResource();
        exit(1);
    }
}
```

Connection and disconnection of slots :- The connectToSlot function initializes the PKCS#11 library, opens a session with the given slot ID, and logs in with the given PIN. The disconnectFromSlot function logs out from the session, closes the session, and finalizes the PKCS#11 library.

```
// Function to connect to HSM
void connectToSlot()
{
    checkOperation(p11Func->C_Initialize(nullptr), "C_Initialize");
    checkOperation(p11Func->C_OpenSession(slotId, CKF_SERIAL_SESSION | CKF_RW_SESSION, nullptr, nullptr, &hSession), "C_OpenSession");
    checkOperation(p11Func->C_Login(hSession, CKU_USER, slotPin, strlen((const char*)slotPin)), "C_Login");
}

// Function to disconnect from HSM
void disconnectFromSlot()
{
    checkOperation(p11Func->C_Logout(hSession), "C_Logout");
    checkOperation(p11Func->C_CloseSession(hSession), "C_CloseSession");
    checkOperation(p11Func->C_Finalize(nullptr), "C_Finalize");
}
```

Generation of Cryptographic Keys :- It primarily contains functions for generating AES, DES3, and RSA keys. Consequently, the generateAesKey function generates an AES256 key through the CKM AES KEY GEN mechanism. It sets various attributes for the key, such as token, private, sensitive, encrypt, decrypt, and key size. Similarly, it offers the generateDes3Key and generateRsaKeyPair functions for the generation of the DES3 and RSA key pair, respectively.

```
// Function to generate AES-256 key and measure time
void generateAesKey()
{
    CK_MECHANISM mech = { CKM_AES_KEY_GEN, nullptr, 0 };
    CK_BBOOL yes = CK_TRUE;
    CK_BBOOL no = CK_FALSE;
    CK_UTF8CHAR label[] = "aes_key";
    CK_ULONG keySize = 32;

    CK_ATTRIBUTE attrib[] =
    {
        {CKA_TOKEN,          &no,        sizeof(CK_BBOOL)},
        {CKA_PRIVATE,        &yes,       sizeof(CK_BBOOL)},
        {CKA_SENSITIVE,      &yes,       sizeof(CK_BBOOL)},
        {CKA_ENCRYPT,        &yes,       sizeof(CK_BBOOL)},
        {CKA_DECRYPT,        &yes,       sizeof(CK_BBOOL)},
        {CKA_VALUE_LEN,      &keySize,   sizeof(CK_ULONG)}
    };
    CK_ULONG attribLen = sizeof(attrib) / sizeof(*attrib);

    clock_t start = clock();
    checkOperation(p11Func->C_GenerateKey(hSession, &mech, attrib, attribLen, &objHandle), "C_GenerateKey");
    clock_t end = clock();
    double elapsedTime = double(end - start) / CLOCKS_PER_SEC;
    cout << "AES-256 Key generation time: " << elapsedTime << " seconds" << endl;
}
```

Encryption and Decryption of Data :- The function encryptData initiates the necessary mechanism for encryption—for instance, CKM AES CBC PAD for AES encryption—and then encrypts the plain data in this buffer. The function decryptData initiates the very mechanism necessary for decryption, decrypts the encrypted data, and stores this in the decryptedData buffer. All the necessary PKCS#11 operations and error checks are handled by these functions.

```
// Function to encrypt data and measure time
void encryptData()
{
    CK_MECHANISM mech = { CKM_AES_CBC_PAD, IV, sizeof(IV)-1 };
    checkOperation(p11Func->C_EncryptInit(hSession, &mech, objHandle), "C_EncryptInit");
    checkOperation(p11Func->C_Encrypt(hSession, plainData, sizeof(plainData)-1, nullptr, &encLen), "C_Encrypt");
    encryptedData = new CK_BYTE[encLen];
    checkOperation(p11Func->C_Encrypt(hSession, plainData, sizeof(plainData)-1, encryptedData, &encLen), "C_Encrypt");
}

// Function to decrypt data and measure time
void decryptData()
{
    CK_MECHANISM mech = { CKM_AES_CBC_PAD, IV, sizeof(IV)-1 };
    checkOperation(p11Func->C_DecryptInit(hSession, &mech, objHandle), "C_DecryptInit");
    checkOperation(p11Func->C_Decrypt(hSession, encryptedData, encLen, nullptr, &decLen), "C_Decrypt");
    decryptedData = new CK_BYTE[decLen];
    checkOperation(p11Func->C_Decrypt(hSession, encryptedData, encLen, decryptedData, &decLen), "C_Decrypt");
}
```

Performance Metrics :-

Time Measurement :- In this regard, time measurement will be of paramount importance to help ascertain the performance of the cryptographic operations. The code measures the elapsed time for key generation, encryption, and decryption operations by calling the clock function. This function shall return the processor time that the program consumed, which in turn shall be converted into seconds. The performance functions will call the relevant cryptographic operations and capture the start and end times to return the duration.

```
// Function to measure key generation time
void measureKeyGenerationTime()
{
    generateAesKey();
}

// Function to measure encryption time
void measureEncryptionTime()
{
    encryptData();
}

// Function to measure decryption time
void measureDecryptionTime()
{
    decryptData();
}
```

Memory Usage :- Yet another important performance metric is memory usage. The getCurrentMemoryUsage function obtains the current memory usage of the process after calling GetProcessMemoryInfo. That function returns an PROCESS MEMORY INFO structure, which holds detailed information on how much memory is in use by a process, including the size of the working set, that is, memory in use, and the peak size of the working set in use. This function uses these functions to calculate the memory usage before and after cryptographic operations are performed, estimating the extra overhead in memory incurred by such operations.

```
// Function to get current memory usage
size_t getCurrentMemoryUsage()
{
    PROCESS_MEMORY_COUNTERS pmc;
    if (GetProcessMemoryInfo(GetCurrentProcess(), &pmc, sizeof(pmc)))
    {
        return pmc.WorkingSetSize;
    }
    else
    {
        cerr << "Failed to get memory usage information." << endl;
        return 0;
    }
}
```

CPU Usage :- CPU usage shows the computational load that the cryptographic operations exert on the processor. The function getCurrentCpuUsage returns the value of CPU usage based on the time elapsed. It captures the CPU time before and after the cryptographic operation was executed 4 and computes the difference. With this value, one gets an idea of the CPU load for the operation. This monitor detects operations that require high computational resources and hence need optimization.

```
// Function to get current CPU usage
double getCurrentCpuUsage()
{
    static LARGE_INTEGER lastCounter;
    static bool isFirstTime = true;

    LARGE_INTEGER currentCounter;
    QueryPerformanceCounter(&currentCounter);

    double elapsedTime = 0.0;
    if (!isFirstTime)
    {
        LARGE_INTEGER freq;
        QueryPerformanceFrequency(&freq);
        elapsedTime = double(currentCounter.QuadPart - lastCounter.QuadPart) / freq.QuadPart;
    }

    lastCounter = currentCounter;
    isFirstTime = false;

    // Assuming single core usage for simplicity
    const double processorUsage = elapsedTime * 100.0;

    return processorUsage;
}
```

Usage :- This function specifies the format in which input should be given for execution.

```
// Function to show usage of the executable
void usage(char exeName[30])
{
    cout << "Command usage: " << endl;
    cout << exeName << " <slotId> <slotPin>" << endl;
    exit(0);
}
```

Throughput :- Throughput is the number of performed operations per any chosen unit of time. The calculateThroughput function measures the throughput of a given operation by executing it repeatedly for the prescribed number of iterations and calculating the total elapsed time. Usually, the throughput is given in operations per second, which clearly indicates the capacity of the system to deal with the cryptographic task. High throughput is needed in many applications of cryptography, such as very fast and efficient cryptographic processing for secure communications and data encryptions.

```
// Function to calculate throughput for a given operation
void calculateThroughput(void (*operation)(), const char* operationName, int numOperations)
{
    clock_t start = clock();
    for (int i = 0; i < numOperations; ++i)
    {
        (*operation)();
    }
    clock_t end = clock();
    double elapsedTime = double(end - start) / CLOCKS_PER_SEC;
    double throughput = numOperations / elapsedTime;
    cout << "Throughput for " << operationName << ": " << throughput << " operations per second" << endl
}
```

Main Function :- It reads the command line arguments for obtaining the slot ID and the PIN. Further, it loads the HSM library, then connects to the slot, measures the performance of the key generation, encryption, and decryption operations, and evaluates the initial and final memory and CPU usage. Afterwards, it prints the results. At the very end, it will print in hexadecimal and plain text both the plain data, the encrypted data, and the decrypted data, disconnecting from the slot and freeing the allocated resources.

```
int main(int argc, char **argv)
{
    if (argc != 3)
    {
        usage(argv[0]);
    }
    else
    {
        slotId = atoi(argv[1]);
        slotPin = new CK_BYTE[strlen(argv[2]) + 1];
        strcpy((char*)slotPin, argv[2]);
    }

    loadHSMLibrary();
    cout << "P11 library loaded." << endl;
    connectToSlot();
    cout << "Connected via session: " << hSession << endl;
```

```
    size_t initialMemoryUsage = getCurrentMemoryUsage();
    double initialCpuUsage = getCurrentCpuUsage();

    measureKeyGenerationTime();
    measureEncryptionTime();
    measureDecryptionTime();

    // Calculate throughput for encryption and decryption
    calculateThroughput(&encryptData, "Encryption", 100); // Adjust numOperations as needed
    calculateThroughput(&decryptData, "Decryption", 100); // Adjust numOperations as needed

    size_t finalMemoryUsage = getCurrentMemoryUsage();
    double finalCpuUsage = getCurrentCpuUsage();

    cout << "Initial Memory Usage: " << initialMemoryUsage << " bytes" << endl;
    cout << "Final Memory Usage: " << finalMemoryUsage << " bytes" << endl;
    cout << "Memory Usage Change: " << (finalMemoryUsage - initialMemoryUsage) << " bytes" << endl;

    cout << "Initial CPU Usage: " << initialCpuUsage << "%" << endl;
    cout << "Final CPU Usage: " << finalCpuUsage << "%" << endl;
    cout << "CPU Usage Change: " << (finalCpuUsage - initialCpuUsage) << "%" << endl;

    // Print plain data, encrypted data, and decrypted data
    printData("Plain Data", plainData, sizeof(plainData) - 1);
    printData("Encrypted Data", encryptedData, encLen);
    printData("Decrypted Data", decryptedData, decLen);

    disconnectFromSlot();
    freeResource();

    delete[] slotPin;
    return 0;
}
```

Results

RSA :-

Compiling :-

```
C:\Users\S Sricharan>cd C:\C\C++\Cryptography\Application\Comparision

C:\C\C++\Cryptography\Application\Comparision>g++ rsa.cpp -o rsa -Iinclude -lpsapi
```

Execution :-

```
C:\C\C++\Cryptography\Application\Comparision>rsa.exe
Command usage:
rsa.exe <slotId> <slotPin>

C:\C\C++\Cryptography\Application\Comparision>rsa.exe 1591203078 1234
P11 library loaded.
Connected via session: 1
RSA Key Pair generation time: 0.179 seconds
Throughput for Encryption: 5882.35 operations/second
Throughput for Decryption: 224.215 operations/second
Initial Memory Usage: 7921664 bytes
Final Memory Usage: 8241152 bytes
Memory Usage Change: 319488 bytes
Initial CPU Usage: 0%
Final CPU Usage: 64.9762%
CPU Usage Change: 64.9762%
Plain Data (Hex): 4561727468206973207468652074686972642070616e6574206f66206f757220536f6c61722053797374656d2e
Plain Data (Plain Text): Earth is the third planet of our Solar System.
Encrypted Data (Hex): c6e8a1e00e9f83bfb63b4f5a503f72f33d476cf78177362d6d1f939991effb2f6a36bc91c8148930c2fe36970ea947909fcf874c10cc08b4bd9a80274143e5f2b22d9c
6098a79fb8a9b25db003c882fece8c4b998df0f9750b3dce5fd4ad11e99a07e52c1633bcbc3ee5bf508e194c90cb42e3d24dee72381518e342602da9faae99f4901716c9c953027c9c0d76a9c820
5c0d196596734fdd8f2ccb25189e795ea3f7c3f4cdd2cb0e1846f5bfdb17d11a3d22f417e027bf75a374d56ef4f7c1f948cdd60fc74d6f860129902227ec427876ecf4a45d9deea819f6108312b1
559407766bf068500323b6cbeb41927d61ca88c2fa7961eaf7757658a9b96b7b59
Encrypted Data (Plain Text): ╞Φíαƒâ┐║;OZP?r≤=Gl≈üw6-môÖæn√/j6╜æ╚ë0┬■6ù¬GÉƒ╧çL┤╜ÜÇ'ACσ≥▒-£`ÿ°ƒ┐¬▒]▒╚é■╬îKÖì≡•u
eûsO▌Å,╦%₧y^ú≈├⌠=╦╤F⌡┐█╤ƒ="⌠α'┐uút╤n⌠≈┴•H=╖║Moå)É"'∞Bxv∞⌠ñ]¥ε¿÷â▒Uövk≡hP#║╤δAÆ}a╧ê┬·yaΩ≈uvX¬╣k{Y
Decrypted Data (Hex): 4561727468206973207468652074686972642070616e6574206f66206f757220536f6c61722053797374656d2e
Decrypted Data (Plain Text): Earth is the third planet of our Solar System.
```

DES3 :-

Compiling :-

```
C:\Users\S Sricharan>cd C:\C\C++\Cryptography\Application\Comparision

C:\C\C++\Cryptography\Application\Comparision>g++ des3.cpp -o des3 -Iinclude -lpsapi
```

Execution :-

```
C:\C\C++\Cryptography\Application\Comparision>g++ des3.cpp -o des3 -Iinclude -lpsapi

C:\C\C++\Cryptography\Application\Comparision>des3.exe
Command usage:
des3.exe <slotId> <slotPin>

C:\C\C++\Cryptography\Application\Comparision>des3.exe 1591203078 1234
P11 library loaded.
Connected via session: 1
Key generation throughput: 25006.3 operations per second
Key generation time: 0.003999 seconds for 100 operations
Encryption throughput: 99206.3 operations per second
Decryption throughput: 99800.4 operations per second
Initial Memory Usage: 7950336 bytes
Final Memory Usage: 8888320 bytes
Memory Usage Change: 937984 bytes
Initial CPU Usage: 0%
Final CPU Usage: 0.85778%
CPU Usage Change: 0.85778%
Plain Data (Hex): 456172746820697320746865207468697264207 06c616e6574206f66206f757220536f6c617220537973746 56d2e
Plain Data (Plain Text): Earth is the third planet of our Solar System.
Encrypted Data (Hex): df3a4ab314db836608845145292255 4c01d81bc25133b6a288d91541f1525bba2d22b98e4ce95b86a18f0d666f3aa7e4
fo:ºΣpted Data (Plain Text): ▀:J|▌âäQE)"UL╪Q3╢óé┘A±R[╟-"╣ÄLΘ[åíÅ
Decrypted Data (Hex): 456172746820697320746865207468697264207 06c616e6574206f66206f757220536f6c617220537973746 56d2e
Decrypted Data (Plain Text): Earth is the third planet of our Solar System.
```

AES :-

Compiling :-

```
C:\Users\S Sricharan>cd C:\C\C++\Cryptography\Application\Comparision

C:\C\C++\Cryptography\Application\Comparision>g++ aes.cpp -o aes -Iinclude -lpsapi
```

Execution :-

```
C:\C\C++\Cryptography\Application\Comparision>aes.exe
Command usage:
aes.exe <slotId> <slotPin>

C:\C\C++\Cryptography\Application\Comparision>aes.exe 1591203078 1234
P11 library loaded.
Connected via session: 1
AES-256 Key generation time: 0.001 seconds
Throughput for Encryption: 100000 operations per second
Throughput for Decryption: 50000 operations per second
Initial Memory Usage: 7905280 bytes
Final Memory Usage: 8056832 bytes
Memory Usage Change: 151552 bytes
Initial CPU Usage: 0%
Final CPU Usage: 0.34657%
CPU Usage Change: 0.34657%
Plain Data (Hex): 45617274682069732074686520746869726420706c616e6574206f66206f757220536f6c617220537973746574656d2e
Plain Data (Plain Text): Earth is the third planet of our Solar System.
Encrypted Data (Hex): a120abbecd8d01835dafae165529c0d85f654544997ba94b4e4ab6c3e6e691105ca6007e913ad1c3ff95cce8294c774d
Encrypted Data (Plain Text): í ¾╡=ìâ]»«U)└╪_eED0{⌐KNJ╢├µµæ\ª~æ:╤├ ò╠®)LwM
Decrypted Data (Hex): 45617274682069732074686520746869726420706c616e6574206f66206f757220536f6c617220537973746574656d2e
Decrypted Data (Plain Text): Earth is the third planet of our Solar System.
```

Table-1 contains the performance comparison of Throughput and Key Generation Time for Encryption-Decryption with different Algorithms such as AES, DES3, RSA.

Algorithm	Encryption Throughput (Operations/sec)	Decryption Throughput (Operations/sec)	Key Generation Time (Seconds)
AES	100000	50000	0.001
DES3	99206.3	99800.4	0.003999
RSA	5882.35	224.215	0.179

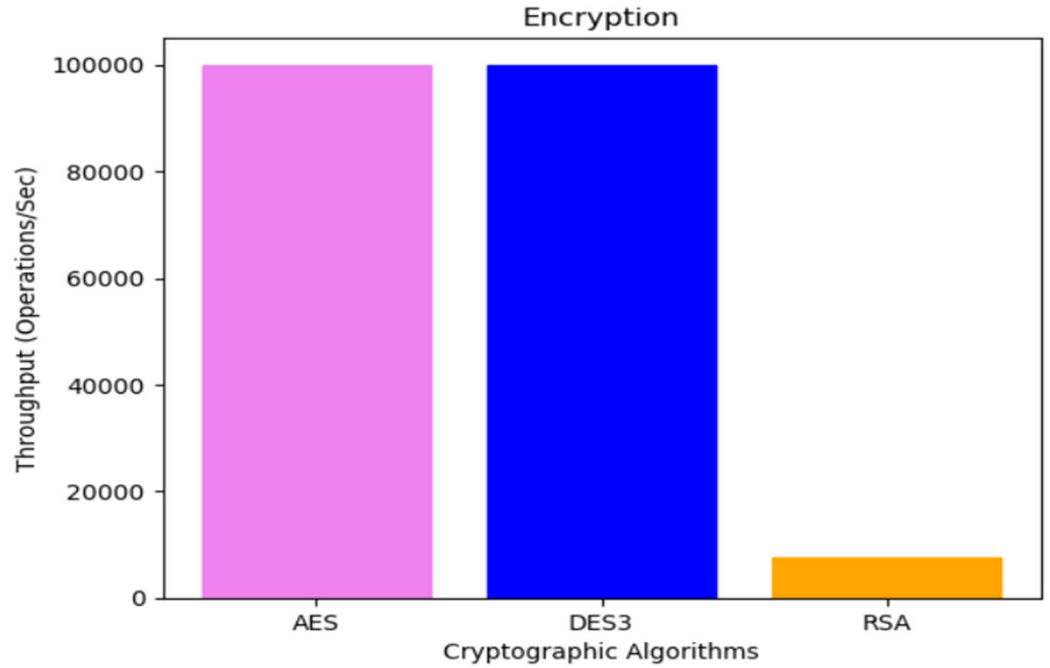
Table 1

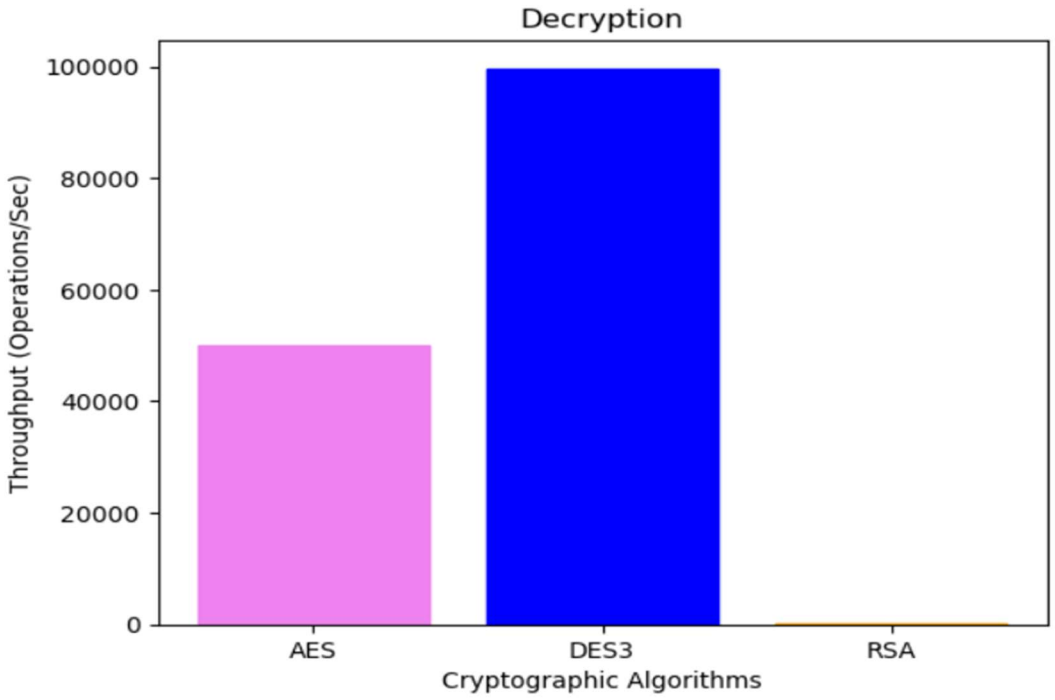
Table-2 contains the performance comparison of CPU usage and Memory Usage for Encryption-Decryption with different Algorithms such as AES, DES3, RSA.

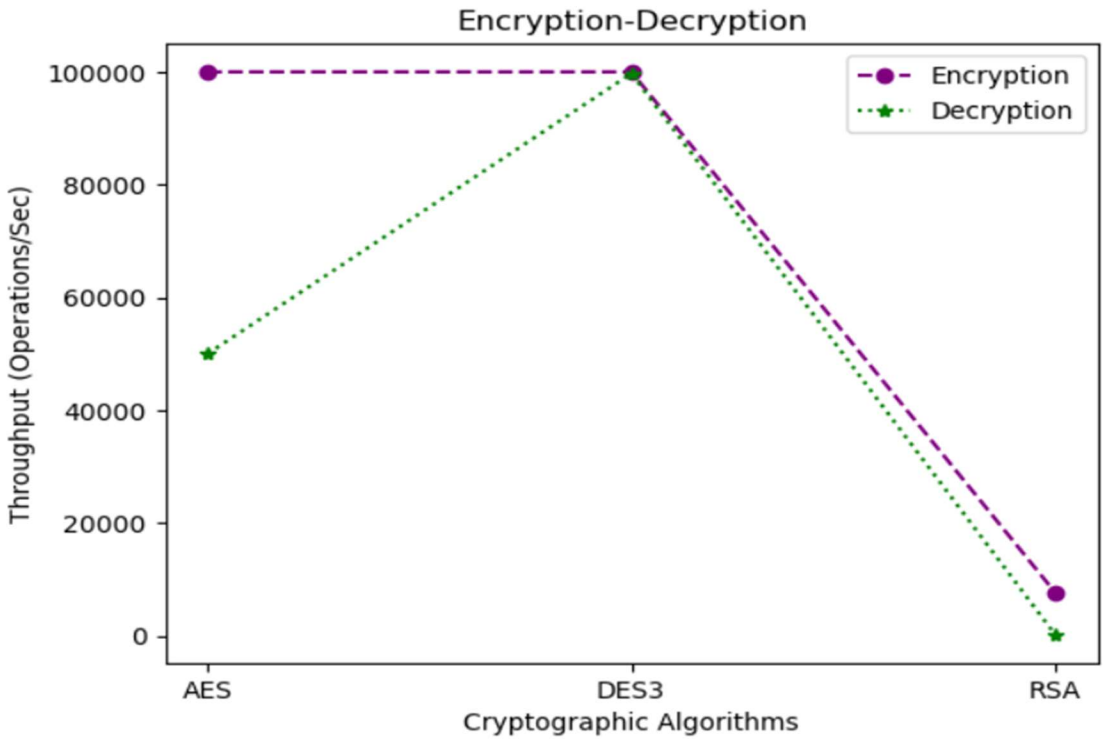
Algorithm	CPU Usage (%)	Memory Usage (Bytes)
AES	0.34657	151552
DES3	0.85778	937984
RSA	64.9762	319488

Table 2

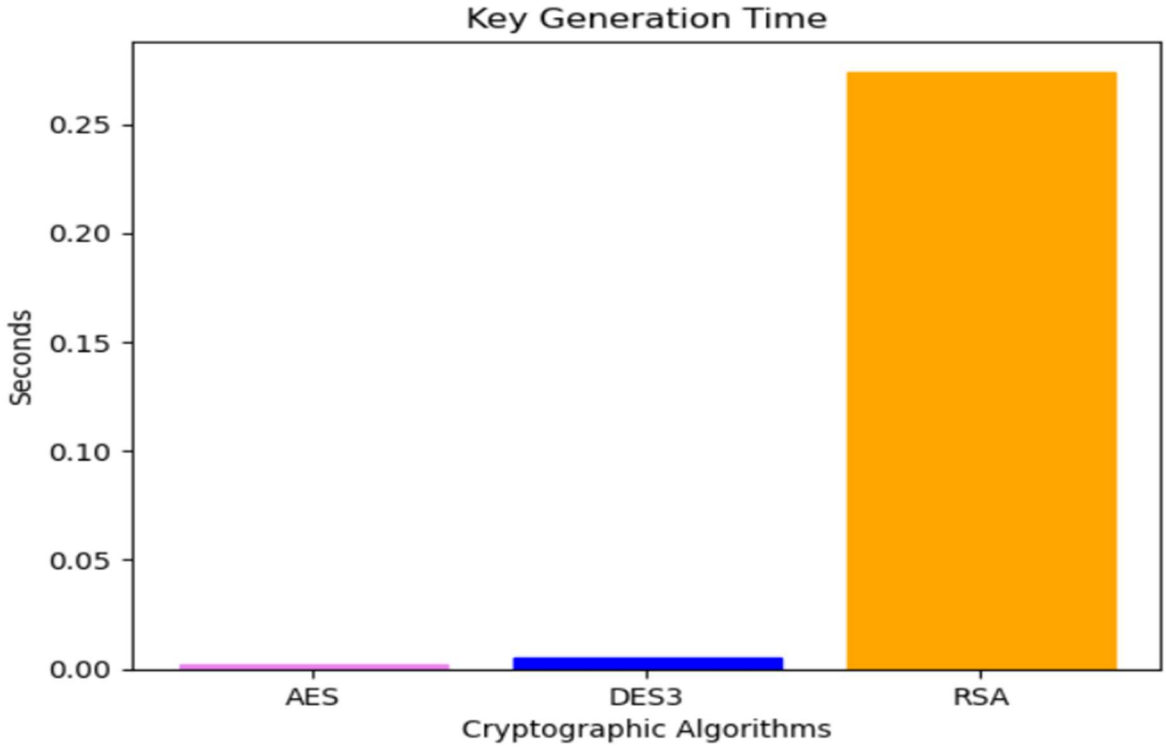
We show the plots for Encryption Throughput, Decryption Throughput, and the comparison below.

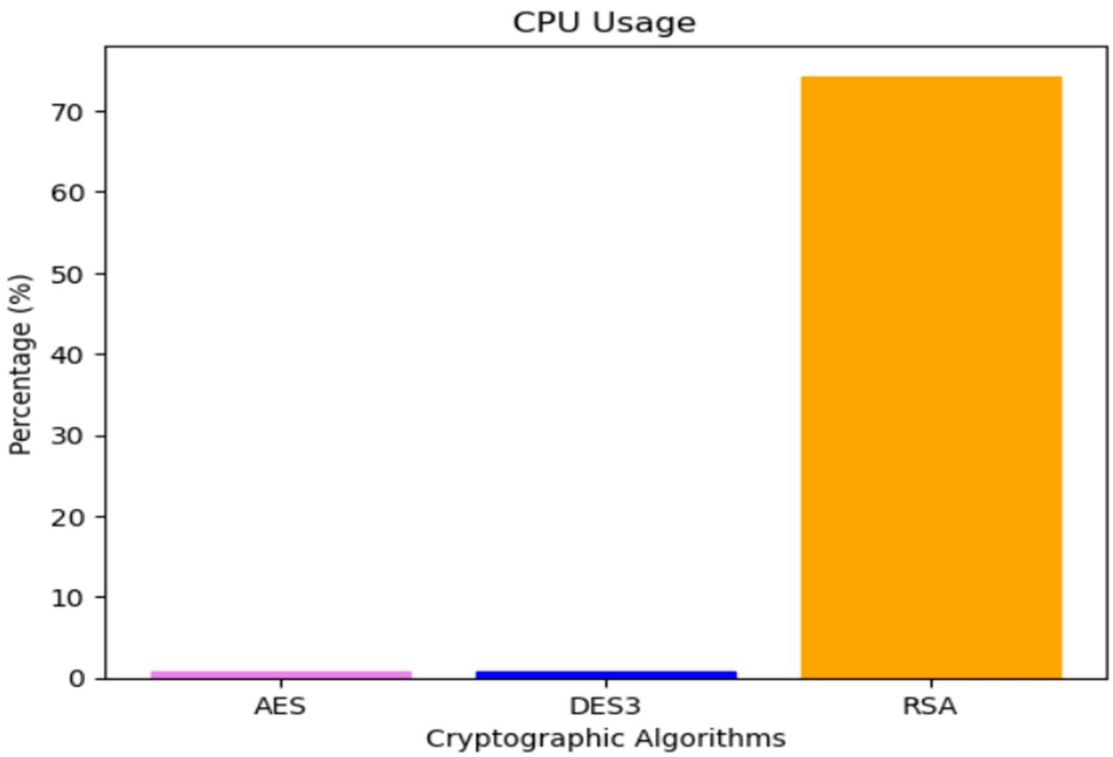


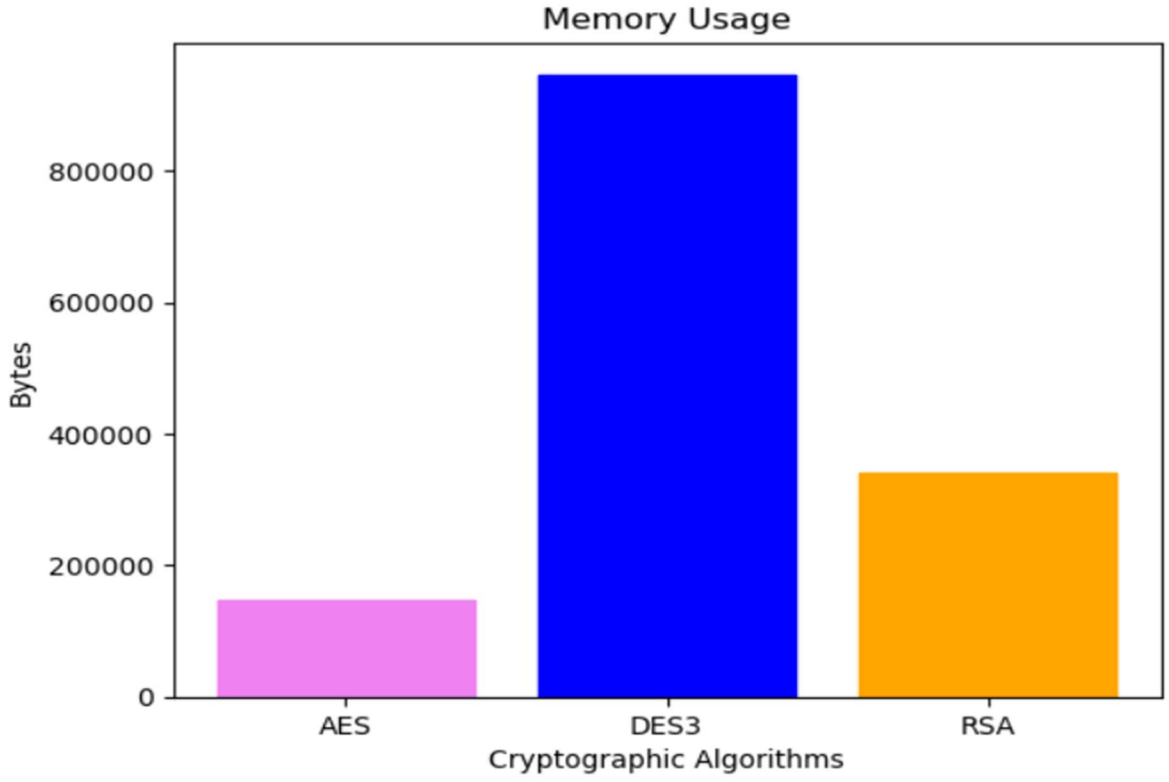




We show the plots for Key Generation Time,CPU Usage,and Memory Usage below.




CPU Usage
Percentage (%)
0
10
20
30
40
50
60
70
AES
DES3
RSA
Cryptographic Algorithms


Memory Usage
Bytes
0
200000
400000
600000
800000
AES
DES3
RSA
Cryptographic Algorithms

Discussion

In this project, we utilize SoftHSM2, an open-source emulation of a physical Hardware Security Module (HSM), to perform basic cryptographic operations and evaluate the performance of three widely used algorithms: AES, 3DES and RSA. SoftHSM2 provides a controlled environment to analyze and compare these algorithms based on various performance metrics.

After implementation we find that,
AES has the best rate among block ciphers because of its speed, security and flexibility, including support for key lengths of 128, 192, and 256 bits, with no known practical attacks against it when it is used correctly. Thus making it highly suitable for modern cryptographic applications.

3DES is more secure than DES, given that it is triple-encrypted. At the same time, the advantages include the compatibility it retains with DES. Despite this, it is slower and less efficient compared to AES, having a relatively small effective key length.

RSA is the most preferred asymmetric encryption algorithm which offers the advantage of high security but is slower than symmetric-key algorithms, specifically when processing large volumes because it does a significant amount of mathematical computing and requires larger key sizes for equivalent security.

The appropriateness of each encryption algorithm ultimately depends on the specific use case:

AES is preferred for securing sensitive data such as encrypting databases and files, securing communications in VPNs and messaging platforms. It is also employed in streaming services for real-time encryption, securing wireless networks according to standards like the wi-fi Protected Access security standard (WPA2), and implementing secure cloud storage solutions.

3DES remains relevant for legacy systems and specific scenarios where backward compatibility with DES is required, such as certain financial transactions and older enterprise applications. However, due to its slower performance and reduced efficiency compared to modern algorithms, it is generally phased out.

RSA is suitable for asymmetric encryption tasks, such as secure key exchange and digital signatures. It is widely used in securing web communications through protocols like HTTPS and in establishing secure email communications. RSA is also employed in public key infrastructure (PKI) systems to manage and validate digital certificates.

Conclusion

In this work, we set up a virtual Hardware Security Module environment using SoftHSM2, a software-based HSM emulator that allows us to simulate and test cryptographic operations in a secure manner. Using SoftHSM2, we implemented C++ programs to compare cryptographic algorithms such as RSA, DES3, and AES based on several performance metrics, including encryption throughput, decryption throughput, key generation time, memory usage, and CPU usage.

Through our experimental and theoretical comparison, we have demonstrated the varying efficiencies of these algorithms in different contexts. Our findings show that AES outperforms the other algorithms in terms of overall efficiency, particularly in encryption and decryption throughput. DES3, while secure, shows significantly slower performance and higher resource consumption. RSA, as an asymmetric algorithm, also shows slower performance in throughput compared to AES but is essential for secure key exchanges.

From these insights, we conclude that AES is the most optimal algorithm for environments where high performance and low resource usage are critical. 3DES is a virtually phased out algorithm which is only suited for legacy systems and other scenarios that require backward compatibility with DES. RSA, despite its slower performance, remains indispensable for tasks requiring secure key exchanges and digital signatures due to its high security and widespread adoption in public key infrastructures and secure communication protocols.
This setup underscores the importance of HSMs in securely managing cryptographic keys and operations, making them indispensable in scenarios where security and performance are paramount.

References

[1] “What is a Hardware Security Module (HSM)? - Utimaco,” utimaco.com, Apr. 05, 2022. Accessed: June. 15, 2024. [Online.] Available: https://utimaco.com/service/knowledge-base/ hardware-security-modules/what-hardware-security-module-hsm

[2] “What is an HSM? — What are the benefits to using an HSM? — Encryption Consulting,” Sep. 23, 2020. Accessed: June. 15, 2024. [Online.] Available: https://www.encryptionconsulting.com/ education-center/what-is-an-hsm/

[3] A. Lance, “What is a Hardware Security Module? — Sidechain Security,” Jun. 28, 2024. Accessed: Jun. 15, 2024. [Online.] Available: https://sidechainsecurity.com/what-is-a-hardware-security-module/

[4] W. Stallings, Cryptography and network security : principles and practice : William Stallings. Upper Saddle River, N.J.: Pearson/Prentice Hall, 2006. pp. 27-372.

[5] P. Kumar Tiwari, V. Choudhary and S. Raj Aman, ”Analysis and Comparison of DES, AES, RSA Encryption Algorithms,” 2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N), Greater Noida, India, 2022, pp. 1913-1918, doi: 10.1109/ICAC3N56670.2022.10073996 .

[6] E. Davies and C. McKenzie, “hardware security module (HSM),” SearchSecurity, Jul. 28, 2021. Accessed: Jun. 21, 2024. [Online.] Available: https://www.techtarget.com/searchsecurity/definition/ hardware-security-module-HSM

[7] “What is Hardware Security Module (HSM)? — Fortinet,” Fortinet. Accessed: Jun. 21, 2024. [Online.] Available:https://www.fortinet.com/ resources/cyberglossary/hardware-security-module

[8] C. Paar , J. Pelzl, “The RSA Cryptosystem.” in: Understanding Cryptography : A Textbook for Students and Practitioners. Berlin, Heidelberg.: Springer Science & Business Media, 2010.
[9] “Hardware security module,” Wikipedia, Jul. 12, 2024.Accessed: June. 21, 2024. [Online.] Available: https://en.wikipedia.org/wiki/Hardware\ security\ module
[10] “SoftHSM v2-OpenDNSSEC.” Available: https://wiki.opendnssec.org/softhsm2/
[11] “OpenSC, Wikipedia” Jun. 30, 2024. . Accessed: Jun. 02, 2024. [Online.] Available: https://en.wikipedia.org/wiki/OpenSC
[12] “PKCS#11”. Available: https://cyberhashira.github.io/notes/pkcs11/pkcs-11v2-20.pdf
[13] “Difference between RSA algorithm and DSA, GeeksforGeeks”. May 22, 2020. Available: https://www.geeksforgeeks.org/ difference-between-rsa-algorithm-and-dsa/
[14] “Triple DES (3DES), GeeksforGeeks”. Mar. 06, 2024. Available: https: //www.geeksforgeeks.org/triple-des-3des/
[15] “Advanced Encryption Standard (AES), GeeksforGeeks”. May 22, 2023. Available: https://www.geeksforgeeks.org/ advanced-encryption-standard-aes/
[16] “Key Generation.” Available: https://github.com/CyberHashira/PKCS-11-Tutorials/tree/main/samples/object management/generating keys
[17] “Encryption Decryption”. Available: https://github.com/CyberHashira/ PKCS-11-Tutorials/tree/main/samples/crypto operations/encryption

[18] R. Andriani, S. E. Wijayanti and F. W. Wibowo, ”Comparision Of AES 128, 192 And 256 Bit Algorithm For Encryption And Description File,” 2018 3rd International Conference on Information Technology, Information System and Electrical Engineering (ICITISEE), Yogyakarta, Indonesia, 2018, pp. 120-124, doi: 10.1109/ICITISEE.2018.8720983. [19] A. Hamza and B. Kumar, ”A Review Paper on DES, AES, RSA Encryption Standards,” 2020 9th International Conference System Modeling and Advancement in Research Trends (SMART), Moradabad, India, 2020, pp. 333-338, doi: 10.1109/SMART50582.2020.9336800.
[20] K. Srinivasan, A. Akash, P. Jaiganesh and M. Mahizhan, ”A VLSI Perspective on Encryption Algorithm Analysis,” 2024 International Conference on Communication, Computing and Internet of Things (IC3IoT), Chennai, India, 2024, pp. 1-6, doi: 10.1109/IC3IoT60841.2024.10550324.
[21] P. Parikh, N. Patel, D. Patel, P. Modi and H. Kaur, ”Ciphering the Modern World: A Comprehensive Analysis of DES, AES, RSA and DHKE,” 2024 11th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 2024, pp. 838-842, doi: 10.23919/INDIACom61295.2024.10498330.
[22] K. Patel, “Performance analysis of AES, DES and Blowfish cryptographic algorithms on small and large data files,” International journal of Information Technology, vol. 11, pp. 813–819, January 2019. doi: 10.1007/s41870-018-0271-4
[23] M. Paradesi Priyanka et al., ”A Comparative Review between Modern Encryption Algorithms viz. DES, AES, and RSA,” 2022 International Conference on Computational Intelligence and Sustainable Engineering Solutions (CISES), Greater Noida, India, 2022, pp. 295-300, doi: 10.1109/CISES54857.2022.9844393.

[24] M. B. Yassein, S. Aljawarneh, E. Qawasmeh, W. Mardini and Y. Khamayseh, ”Comprehensive study of symmetric key and asymmetric key encryption algorithms,” 2017 International Conference on Engineering and Technology (ICET), Antalya, Turkey, 2017, pp. 1-7, doi: 10.1109/ICEngTechnol.2017.8308215.