Nome: Guilherme Augusto Amorim Terrell

Prof: Julio López

Disciplina: MO442

2º Semestre 2024

Avaliação do Impacto do Método de Geração e Compartilhamento de Chave no Tempo de Execução de um Esquema Criptográfico Simétrico Desenvolvido em Python com a Biblioteca Cryptography

1. Resumo

O propósito desse trabalho é avaliar o quão mais rápido ou mais lento um esquema criptográfico simétrico fica a depender do método de compartilhamento de segredo adotado utilizando uma biblioteca de criptografia implementada em *python* chamada *cryptography*. Buscou-se também avaliar a viabilidade do uso de bibliotecas prontas como a já citada para a prototipagem e implementação de aplicações em criptografia.

Como método de avaliação foram implementados os seguintes esquemas criptográficos simétricos:

- AES-GCM com x25519 para compartilhamento da chave simétrica
- AES-GCM com x448 para compartilhamento da chave simétrica
- AES-GCM com RSA para compartilhamento da chave simétrica

Os tempos de execução de cada uma das etapas que envolvem o esquema criptográfico foram medidos e, a partir da comparação constata-se claramente que os mecanismos de troca de chave baseados em curvas elípticas são significativamente mais rápidos, sendo o x25519 o mais rápido dentre os testados nesse trabalho. Além disso, concluiu-se ser muito eficaz o emprego da biblioteca *cryptography* para o desenvolvimento das aplicações aqui propostas, uma vez que a referida biblioteca é muito bem documentada e possui uma sintaxe simples, abstraindo do usuário toda a complexidade matemática envolvida nos processos de geração e compartilhamento de chaves e de encriptação e decriptação, fornecendo ao desenvolvedor uma grande quantidade de exemplos de código. Além disso, essa biblioteca abrange uma vasta gama de funcionalidades como criptografia assimétrica e simétrica com possibilidade de escolha de diferentes modos de operação, assinatura digital, entre outros. Contudo, há limitações. A biblioteca, em alguns casos, não é muito flexível em termos dos parâmetros que servem como input das funções, dessa forma o usuário fica limitado aos valores padrão. Vale ainda ressaltar que a biblioteca *cryptography* não foi desenvolvida para uma arquitetura em específica, então, a depender da máquina onde os *scripts* são executados (processador, sistema operacional, etc), o tempo de execução pode ser diferente dos que foram apresentados aqui.

2. Introdução

Dentro da criptografia simétrica é imprescindível que as partes que se comunicam possam compartilhar essa chave de forma rápida, eficiente e segura, sem vazar nenhuma informação sobre o segredo em questão. Existem várias técnicas que permitem esse compartilhamento seguro, porém algumas requerem mais recursos computacionais como processamento e memória do que outras, sendo assim, um esquema criptográfico pode ficar mais ou menos eficiente do ponto de vista de tempo de execução e consumo de memória e tempo de CPU a depender de qual mecanismo de troca de chave foi escolhido. O objetivo desse trabalho não é propor uma implementação nova dos mecanismos de troca de chave já existentes, mas sim a validação do custo computacional de alguns desses mecanismos a partir da implementação de uma aplicação de criptografia simétrica que utiliza métodos de compartilhamento de segredo para estabelecer uma chave simétrica em comum aos elementos que estão se trocando mensagens a partir do uso de bibliotecas que já estão prontas e podem rodar em qualquer plataforma, o que facilita a verificação e revisão dos resultados aqui apresentados. Tem-se também como objetivo o estudo da biblioteca adotada para elaboração dos códigos, avaliando a viabilidade do uso desse tipo de ferramenta no desenvolvimento de aplicações criptográficas, e mitigando os prós e contras da utilização de uma plataforma padrão em detrimento do desenvolvimento de um código específico para a máquina que executará o programa.

3. Conceitos Importantes

Conforme apresentado na introdução o foco desse trabalho é o estudo de mecanismos de compartilhamento de segredos associados a criptografia simétrica. Os seguintes conceitos se fazem importantes para o entendimento desse trabalho:

- Chave simétrica: String aleatória de tamanho fixo utilizada tanto pelo emissor da mensagem quanto pelo receptor para cifrar e decifrar mensagens
- Cifradores de bloco: Algorítmo criptográfico que opera (cifrar ou decifrar) sobre uma quantidade fixa de bytes de uma mensagem (ou seja, um bloco) utilizando uma chave simétrica. Nesse trabalho foi utilizado o algorítmo AES.
- Modos de operação: Esquema que utiliza um cifrador de bloco para encriptar ou decriptar um conjunto de bytes maior do que o tamanho de bytes que aquele cifrador pode operar de uma única vez. Nesse trabalho utilizou-se o AES-GCM.
- Criptografia simétrica: Técnica criptográfica na qual emissor e receptor utilizam uma única chave compartilhada entre ambos (chave simétrica) para encriptar e decriptar mensagens.
- Criptografia assimétrica: Técnica criptográfica na qual emissor e receptor geram 2 pares de chave, sendo uma privada e uma pública. Nesse esquema gera-se uma chave compartilhada entre ambos a partir de operações matemáticas que combinam a chave privada de um com a chave pública do outro
- Compartilhamento de segredo: Mecanismo criptográfico que permite que emissor e receptor estabeleçam uma chave (segredo) em comum para cifrar e decifrar mensagens usando o esquema de criptografia simétrica. Os seguintes mecanismos de troca de chave foram estudados nesse trabalho:
    - X25519: Algorítmo de troca de chaves baseado no modelo de Diffie-Hellman construído sobre a curva elíptica curve25519
    - X448: Algorítmo de troca de chaves baseado no modelo de Diffie-Hellman construído sobre a curve elíptica curve448
    - RSA (esquema híbrido): Algorítmo de criptografia assimétrica que nesse trabalho foi utilizado para cifrar uma chave simétrica para utilizar o esquema simétrico de criptografia

4. Metodologia

Utilizando a biblioteca *cryptography*, implementou-se em python os seguintes esquemas:

- AES-GCM com x25519 para compartilhamento da chave simétrica
- AES-GCM com x448 para compartilhamento da chave simétrica
- AES-GCM com RSA para compartilhamento da chave simétrica

Comparou-se os tempos de execução de cada etapa e, com isso, obtém-se uma idea de qual esquema é mais computacionalmente custoso. Os *scripts* em *python* possuem a mesma estrutura (diferenciando-se um do outro apenas no que se refere ao mecanismo de troca de chave). Os algorítmos em *python* implementados têm as seguintes etapas:

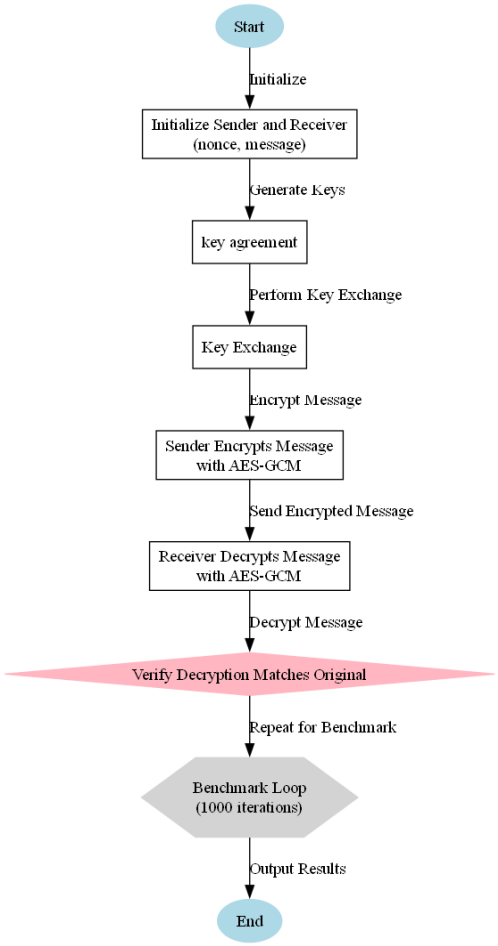


Figura 1: Flow chart com a descrição do algorítmo implementado em python para encriptação utilizando AES-GCM e um mecanismo de troca de chaves

Nesse trabalho a rotina "AES_<keyExchangeMechanism>_benchmark" que implementa todo o esquema criptográfico será executada 1000 vezes, onde "keyExchangeMechanism" refere-se ao mecanismo de troca de chave que foi adotado naquela implementação (por exemplo, "AES_x25519_benchmark" utiliza AES-GCM com troca de chave implementada utilizando x25519). A cada nova execução do código será gerado uma nova mensagem, sempre com tamanho1MB através da rotina "*os.urand(1024*1024)*", bem como um novo *nonce* também utilizando "*os.urand()*", mas os tempos de geração da mensagem e do *nonce* não são computados pois esses são os inputs do esquema criptográfico. Os seguintes tempos serão coletados utilizando o método *time.time()* do module *Time*, que está disponível por padrão ao instalar o *python:*

- Tempo para geração da chave (*key generation*)
- Tempo para troca de chave (*key exchange*)
- Tempo para aquele que envia a mensagem (*"Sender"*) encriptar o conteúdo utilizando AES-GCM e a chave previamete acordada (*AES-GCM encrypt*)
- Tempo para aquele que recebe o conteúdo ("Receiver") decriptar a mensagem utilizando AES-GCM e a chave previamente acordada (AES-GCM decrypt)

Os tempos de execução são salvos em um array e, caso não haja nenhum erro, isto é, o texto decifrado ("*plaintext*") é sempre igual a mensagem original que foi enviada, então é computada a média nos tempos de execução em cada uma das etapas descritas acima. A tabela abaixo indica as ferramentas utilizadas para implementação e execução dos scritps, bem como descreve o ambiente no qual os scripts foram executados:

Python v3.12.4
Pip v24.3.1
Cryptography v43.0.3
Time (built in)
Os (built in)
Sys (built in)
GNU Bash v5.2.26
Visual Studio Code v1.95

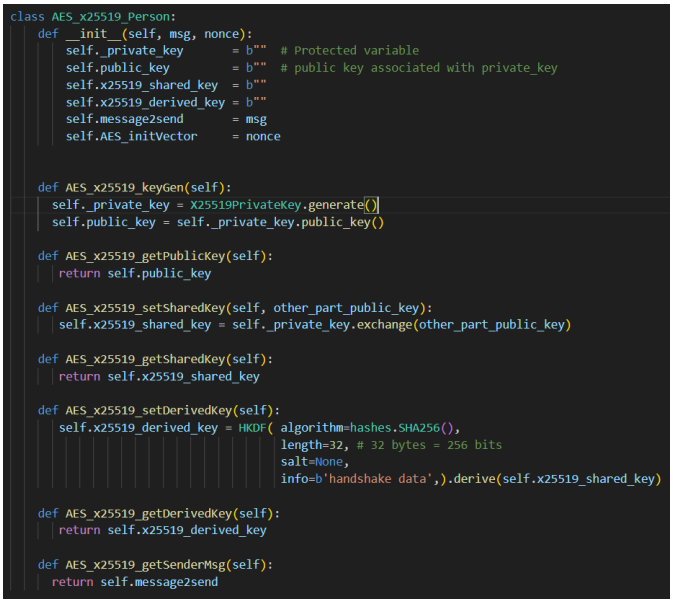
Processador Intel(R) Core(TM) i5-12450H 12th Gen, 2 GH, x86
Memória RAM 8GB, SSD 512 GB
Sist Op Windows 11 Home Single Language, v23H2, 64 bits

Tabela 1: Lista de ferramentas e descrição do ambiente utilizadas (o)

Para garantir que os dados do *"Sender"* e do *"Receiver"* não fiquem visiveis globalmente no código foram criadas classes, sendo *"Sender" e "Receiver"* objetos instanciados dessas classes. Cada informação como chave privada, chave pública são atributos de cada objeto e não ficam disponíveis globalmente no arquivo, requerendo métodos específicos da classe para acessar informações específicas de um objeto, dessa forma conseguimos isolar informações privadas de cada uma das partes comunicantes a fim de emular um ambiente real de comunicação (a classe não fornece nenhum método para obter a chave privada, por exemplo, portanto do ponto de vista dessa implementação não há como uma parte saber a chave privada da outra). Veja abaixo a implementação da classe "*AES_x448_Person*" que pode ser utilizada tanto por quem vai enviar a mensagem quanto por quem a recebe (para o esquema que utiliza x25519 a classe é exatamente igual a não ser pelos nomes que possuem x25519 ao invés de x448). A classe foi implementada dessa forma pois nesse esquema as "obrigações" daquele que envias e daquele que recebe são praticamente as mesmas (ambos têm que gerar um par de chaves, ambos devem coletar a chave pública do outro e calcular a chave compartilhada, etc).

```
class AES_x448_Person:
    def __init__(self, msg, nonce):
        self._private_key     = b""  # Protected variable
        self.public_key       = b""  # public key associated with private_key
        self.x448_shared_key  = b""
        self.x448_derived_key = b""
        self.message2send     = msg
        self.AES_initVector   = nonce


    def AES_x448_keyGen(self):
      self._private_key = X448PrivateKey.generate()
      self.public_key = self._private_key.public_key()

    def AES_x448_getPublicKey(self):
        return self.public_key

    def AES_x448_setSharedKey(self, other_part_public_key):
        self.x448_shared_key = self._private_key.exchange(other_part_public_key)

    def AES_x448_getSharedKey(self):
        return self.x448_shared_key

    def AES_x448_setDerivedKey(self):
        self.x448_derived_key = HKDF( algorithm=hashes.SHA256(),
                                      length=32, # 32 bytes = 256 bits
                                      salt=None,
                                      info=b'handshake data',).derive(self.x448_shared_key)

    def AES_x448_getDerivedKey(self):
        return self.x448_derived_key

    def AES_x448_getSenderMsg(self):
        return self.message2send
```

. Figura 2: Classe com atributos e métodos relativos a aplicação AES-GCM com x448 para troca de segredo



```python
class AES_x25519_Person:
    def __init__(self, msg, nonce):
        self._private_key       = b""  # Protected variable
        self.public_key         = b""  # public key associated with private_key
        self.x25519_shared_key  = b""
        self.x25519_derived_key = b""
        self.message2send       = msg
        self.AES_initVector     = nonce


    def AES_x25519_keyGen(self):
        self._private_key = X25519PrivateKey.generate()
        self.public_key = self._private_key.public_key()

    def AES_x25519_getPublicKey(self):
        return self.public_key

    def AES_x25519_setSharedKey(self, other_part_public_key):
        self.x25519_shared_key = self._private_key.exchange(other_part_public_key)

    def AES_x25519_getSharedKey(self):
        return self.x25519_shared_key

    def AES_x25519_setDerivedKey(self):
        self.x25519_derived_key = HKDF( algorithm=hashes.SHA256(),
                                        length=32, # 32 bytes = 256 bits
                                        salt=None,
                                        info=b'handshake data',).derive(self.x25519_shared_key)

    def AES_x25519_getDerivedKey(self):
        return self.x25519_derived_key

    def AES_x25519_getSenderMsg(self):
        return self.message2send
```

Figura 3: Classe com atributos e métodos relativos a aplicação AES-GCM com x25519 para troca de segredo

Abaixo encontram-se tabelas descrevendo brevemente cada um dos objetos e métodos da biblioteca *cryptography* que foram utilizados nos esquemas AES-GCM com x25519 para compartilhamento de chave e AES-GCM com x448 para compartilhamento de chave:

Método	generate()
Classe	cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey
Descrição	Gerar uma privada aleatória baseada na curva elíptica curve448. Para obter a chave pública associada a chave privada basta aplicar o método .public() na chave privada
Input	void
Output	Chave privada
Documentação	https://cryptography.io/en/latest/hazmat/primitives/asymmetric/x448/#cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey.generate

Tabela 2: Descrição do método generate

Método	exchange()
Classe	cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey
Descrição	Gerar uma chave compartilhada de tamanho fixo, no caso do x448 são 56 bytes. Implementa as operações matemáticas que combinam a chave privada do que envia com a chave pública do que recebe para gerar uma chave compartilhada
Input	peer_public_key: Chave pública do outro elemento com que se está comunicando
Output	Chave compartilhada "crua" (raw shared secret). Ainda não está pronta para ser utilizada para cifrar ou decifrar
Documentação	https://cryptography.io/en/latest/hazmat/primitives/asymmetric/x448/#cryptography.hazmat.primitives.asymmetric.x448.X448PrivateKey.exchange

Tabela 3: Descrição do método exchange

Método	HKDF()
Classe	cryptography.hazmat.primitives.kdf.hkdf
Descrição	Inicializar um objeto KDF que será utilizado para gerar a chave derivada
Input	Chave pública do outro elemento com que se está comunicando algorithm: Algorítmo de hash utilizado internamente lenght: Tamanho desejado da chave compartilhada salt: Valor aleatório que garante que a chave derivada é única, mesmo que a mesma chave compartilhada "crua" (método exchange) seja utilizada. Geralmente tem 16 bytes info: Distingue uma chave das outras chaves derivadas atribuindo-lhe um propósito único
Output	Objeto KDF
Documentação	https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/#cryptography.hazmat.primitives.kdf.hkdf.HKDF

Tabela 4: Descrição do método HKDF

Método	derive()
Classe	cryptography.hazmat.primitives.kdf.hkdf
Descrição	Derivar uma chave compartilhada a partir da hash gerada com o método exchange e com o objeto inicializado em HKDF
Input	void
Output	Chave compartilhada
Documentação	https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions/#cryptography.hazmat.primitives.kdf.hkdf.HKDF.derive

Tabela 5: Descrição do método derive

Já no esquema AES-GCM com RSA como método de compartilhamento de chave, emissor e receptor tem “obrigações” distintas, portanto entendeu-se no contexto desse trabalho que seria pertinente que as classes que “emulam” cada uma das partes tem alguns atributos e métodos distintos, conforme imagens abaixo:

```python
class AES_RSA_sender:
    def __init__(self, AES_initVector, message):
        self.AES_initVector     = AES_initVector
        self._AES_key           = b""
        self.message2send       = message

    def AES_RSA_AESkeyGen(self):
        self._AES_key = AESGCM.generate_key(bit_length=256)

    def AES_RSA_encryptSimetricKey(self, receiver_public_rsa_key):
        return receiver_public_rsa_key.encrypt(self._AES_key, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
                                                                           algorithm=hashes.SHA256(),
                                                                           label=None) )

    def AES_RSA_encryptMessage(self):
        return AESGCM(self._AES_key).encrypt(self.AES_initVector, self.message2send, None)

    def AES_RSA_getSenderMessage(self):
        return self.message2send
```

Figura 4: Classe com atributos e métodos do elemento que envia relativos a aplicação AES-GCM com RSA para troca de segredo

.

```python
class AES_RSA_receiver:
    def __init__(self, AES_initVector):
        self._rsa_private_key   = b"" # Protected variable
        self.public_key         = b"" # public key associated with private_key
        self._AES_key           = b""
        self.AES_initVector     = AES_initVector

    def AES_RSA_RSAkeyGen(self):
        self._rsa_private_key = rsa.generate_private_key( public_exponent=65537,key_size=2048,) # 2048 bits RSA key size
        self.public_key = self._rsa_private_key.public_key()

    def AES_RSA_getReceiverPublicKey(self):
        return self.public_key

    def AES_RSA_decryptAndGetAESkey(self, encrypted_AES_key):
        self._AES_key = self._rsa_private_key.decrypt(encrypted_AES_key, padding.OAEP(
                                                                        mgf=padding.MGF1(algorithm=hashes.SHA256()),
                                                                        algorithm=hashes.SHA256(),
                                                                        label=None))
    def AES_RSA_getPlainText(self, cipheredText):
        return AESGCM(self._AES_key).decrypt(self.AES_initVector, cipheredText, None)
```

Figura 5: Classe com atributos e métodos do elemento que recebe relativos a aplicação AES-GCM com RSA para troca de segredo

Abaixo encontram-se tabelas descrevendo brevemente cada um dos objetos e métodos da biblioteca *cryptography* que foram utilizados nos esquemas AES-GCM com RSA para compartilhamento de chave:

Método	generate_key()
Classe	from cryptography.hazmat.primitives.ciphers.aead import AESGCM
Descrição	Gera uma chave AES-GCM
Input	void
Output	Chave gerada
Documentação	https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM

Tabela 6: Descrição do método generate_key() para AES-GCM

Método	encrypt()
Classe	cryptography.hazmat.primitives.asymmetric
Descrição	Encriptção RSA
Input	<table><tr><td>message</td><td>Conteúdo a ser cifrado</td></tr><tr><td>padding</td><td>Método de padding</td></tr></table>
Output	Texto cifrado
Documentação	https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#encryption

Tabela 7: Descrição do método encrypt() para RSA

Método	decrypt()
Classe	cryptography.hazmat.primitives.asymmetric
Descrição	Decriptção RSA
Input	<table><tr><td>ciphertext</td><td>Texto cifrado</td></tr><tr><td>padding</td><td>Método de padding</td></tr></table>
Output	Texto original (plaintext)
Documentação	https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#decryption

Tabela 8: Descrição do método decrypt() para RSA

As tabelas abaixo descrevem métodos e objetos relacionados a criptografia simétrica com AES-GCM, que foram utilizados nos 3 esquemas aqui discutidos.

Método	AESGCM	
Classe	cryptography.hazmat.primitives.ciphers.aead.AESGCM	
Descrição	Inicializar o cifrador AES-GCM	
Input		
	Key	Chave simétrica de 128, 192 ou 256 bits que será usada para cifrar e decifrar
Output	Objeto que representa um cifrador AES-GCM	
Documentação	https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM	

Tabela 9: Descrição do método AESGCM

Método	encrypt()	
Classe	cryptography.hazmat.primitives.ciphers.aead.AESGCM	
Descrição	Encriptar um texto plano. Pode cifrar no máximo 2^32 – 1 bytes	
Input		
	nonce	Sequência aleatória de 12 bytes
	data	Mensagem a ser encriptada
	associated_data	Será utilizado para autenticar a mensagem. Pode ser None
Output	Mensagem cifrada	
Documentação	https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM.encrypt	

Tabela 10: Descrição do método encrypt() para AES-GCM

Método	decrypt()	
Classe	cryptography.hazmat.primitives.ciphers.aead.AESGCM	
Descrição	Descriptar um texto cifrado. Pode decifrar no máximo 2^32 – 1 bytes	
Input		
	nonce	Sequência aleatória de 12 bytes
	data	Mensagem cifrada
	associated_data	Será utilizado para autenticar a mensagem. Pode ser None
Output	Mensagem original (*plaintext*)	
Documentação	https://cryptography.io/en/latest/hazmat/primitives/aead/#cryptography.hazmat.primitives.ciphers.aead.AESGCM.decrypt	

Tabela 11: Descrição do método decrypt para AES-GCM

5. Resultados e discussões

A execução dos 3 scripts que implementam individualmente os esquemas de criptografia simétrica AES-GCM com x25519, AES-GCM com x448 e AES-GCM com RSA (híbrido), com os seguintes ambientes e parâmetros:

- Computador desconectado da bateria;
- Algorítmo hash para obter chave derivada x25519 e x448: SHA256
- Salt para obter a chave derivada x25519 e x448: 32 bytes
- Encriptação e decriptação da chave RSA feita com padding OAEP, SHA256

Esquema	Tempo médio key gen (s)	Tempo médio key exchange (s)	Tempo médio encrypt (s)	Tempo médio decrypt (s)	Tempo médio total
AES-GCM com x25519	0,000165	0,000143	0,000605	0,000595	0,001508
AES-GCM com x448	0,000966	0,001300	0,000656	0,000620	0,003542
AES-GCM com RSA	0,065638	0,001603	0,000667	0,000629	0,068537

Tabela 12: Tabela de média dos tempos de execução de cada etapa do esquema de acordo com os parâmetros listados acima

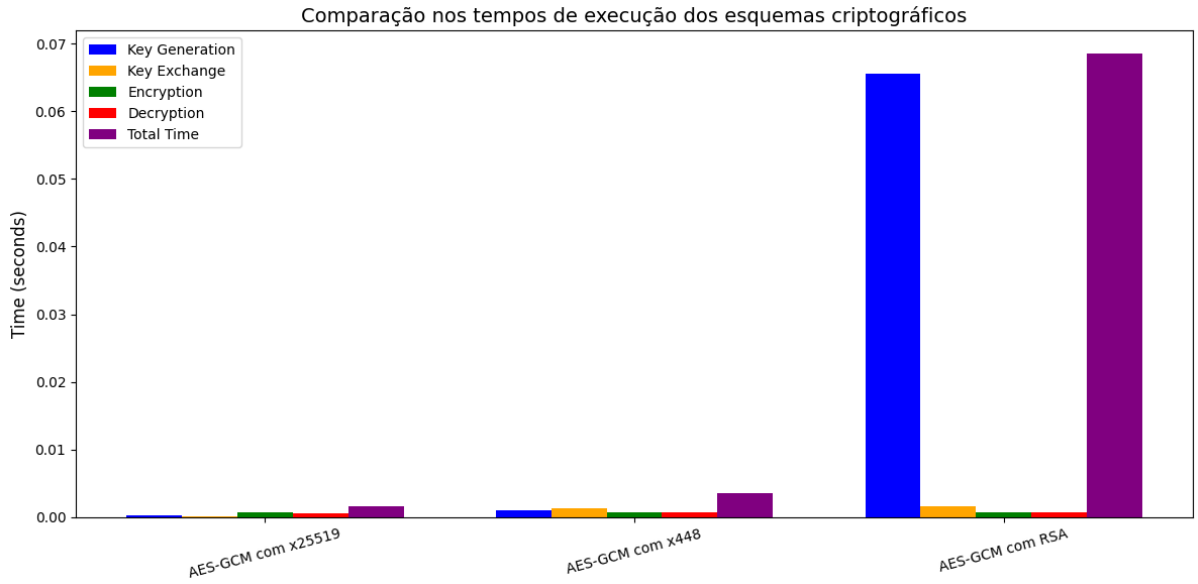


Figura 6: Média dos tempos de execução de acordo com a tabela 12

- Computador conectado a bateria
- Algorítmo hash para obter chave derivada x25519 e x448: SHA256
- Salt para obter a chave derivada x25519 e x448: 32 bytes
- Encriptação e decriptação da chave RSA feita com padding OAEP, SHA256

Esquema	Tempo médio key gen (s)	Tempo médio key exchange (s)	Tempo médio encrypt (s)	Tempo médio decrypt (s)	Tempo médio total
AES-GCM com x25519	0,000085	0,000076	0,000364	0,000362	0,000887
AES-GCM com x448	0,000485	0,000645	0,000359	0,000340	0,001829
AES-GCM com RSA	0,034282	0,000835	0,000394	0,000355	0,035866

Tabela 13: Tabela de média dos tempos de execução de cada etapa do esquema de acordo com os parâmetros listados acima

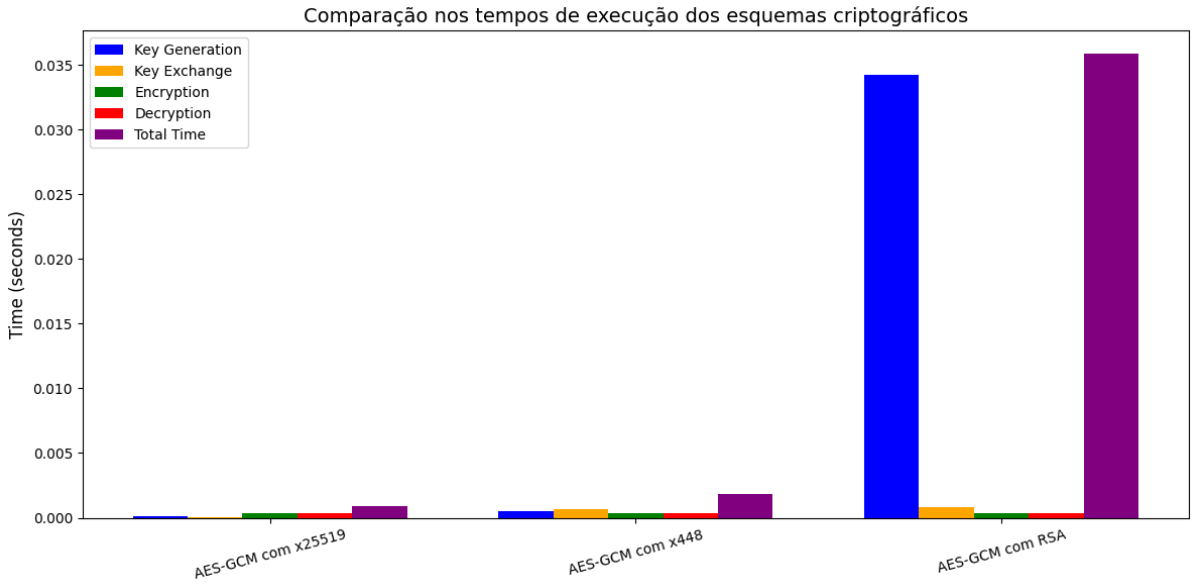


Figura 7: Média dos tempos de execução de acordo com a tabela 13

6. Conclusão

No que se refere a comparação dos esquemas, pode-se validar que os mecanismos de troca de chaves baseados em curvas elítpticas são muito mais efeicientes do que o mecanismo híbrido com RSA, especialmente o x25519, que foi quase 400 vezes mais rápido. Isso significa que o computador foi menos exigido em termos de recursos como tempo de CPU e memória durante a execução dos cálculos, e portanto corrobora a ideia vista em aula de que mecanismos de troca de chaves baseados em curvas elítpticas são os mais indicados para sistestemas com poucos recursos computacionais, como os embarcados por exemplo. Contudo, embora os resultados obtidos nesse trabalho com a medições dos tempos esteja de acordo com o esperado, é possível que a execução desse mesmo benchmark em arquiteruras distintas resulte em tempos deferentes, uma vez que a biblioteca utilizada na implementação dos esquemas é de uso geral, e não está otimizada para uma arquitetura em específico.

Conclui-se também ser bastate válido e muito útil o uso da biblioteca *cryptography* para implementação dos esquemas criptográficos, principlamente devido a facilidade e simplicidade de implementação por ela proporcionados, abstraindo do usuário todo o rigor matemático que está por trás da teoria criptográfica. Isso vale também para projetos mais sofisticados do que os que foram apresentados aqui. A única ressalva é se a eficiência do algorítmo for um fator determinante do projeto, nesse caso é possível que o desempenho do esquema implementado utilizando a biblioteca *cryptography* seja inferior ao de uma aplicação desenvolvida para uma arquitetura em específico.

7. Referências bibliográficas

Python Cryptographic Authority. (2024). Cryptography (Version 43.0.3)[Computer software]. Disponível em: https://cryptography.io/en/latest/

HAYATO, Fujii., ARANHA, Diego. “Curve25519 for the Cortex-M4 and beyond”.Disponível em: https://www.lasca.ic.unicamp.br/media/publications/paper39.pdf

HAYATO, Fujii., ARANHA, Diego. “Efficient Curve25519 Implementation for ARM Microcontrollers”. Disponível em https://sol.sbc.org.br/index.php/sbseg_estendido/article/view/4142/4071

MICROCHIP. “RSA vs. ECC comparison for Embedded Systems”. Disponível em https://ww1.microchip.com/downloads/en/DeviceDoc/00003442A.pdf

STALLINGS, William. Other Public Key Cryptosystems. *In*: STALLINGS, William.

Cryptography and Network Security Principles and Practice. England: PEARSON, 2017.p.313- 336.

Etiris Magazine. (n.d.). *Best Python cryptography libraries for secure data encryption*. Medium. Disponível em: https://medium.com/@etirismagazine/best-python-cryptography-libraries-for-secure-data-encryption-71b132f47d74