



---

# Criptografia Pós-Quântica

## Comunicação segura usando OpenSSL

<b>Elaborador:</b>	Israel de Oliveira Silva
<b>Orientadora:</b>	Mariana Godoy Vazquez Miano

---

**Israel de Oliveira Silva**

## **Criptografia Pós-Quântica**

### **Comunicação segura usando OpenSSL**

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Segurança da Informação na área de concentração em segurança da informação.

Este trabalho corresponde a Relatório Técnico apresentado por Israel de Oliveira Silva e orientado pela Profa. Doutora Mariana Godoy Vazquez Miano

**Americana, SP**

**2024**

---

## FICHA CATALOGRÁFICA – Biblioteca Fatec Americana Ministro Ralph Biasi- CEETEPS Dados Internacionais de Catalogação-na-fonte

SILVA, Israel de Oliveira

Criptografia Pós-Quântica: Comunicação segura usando  
OpenSSL. / Israel de Oliveira SILVA – Americana, 2024.

52f.

Relatório técnico (Curso Superior de Tecnologia em  
Segurança da Informação) - - Faculdade de Tecnologia de  
Americana Ministro Ralph Biasi – Centro Estadual de Educação  
Tecnológica Paula Souza

Orientadora: Profa. Dra. Mariana Godoy Vazquez MIANO

1. Criptografia 2. Segurança em sistemas de informação 3.  
VPN – rede de computadores. I. SILVA, Israel de Oliveira II. MIANO,  
Mariana Godoy Vazquez III. Centro Estadual de Educação  
Tecnológica Paula Souza – Faculdade de Tecnologia de Americana  
Ministro Ralph Biasi

CDU: 681.518.5

681.518.5

681.519VPN

Elaborada pelo autor por meio de sistema automático gerador de  
ficha catalográfica da Fatec de Americana Ministro Ralph Biasi.

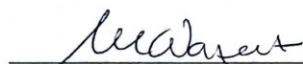
Israel de Oliveira Silva


**Criptografia Pós-Quântica**  
**Comunicação segura usando OpenSSL**


Trabalho de graduação apresentado como exigência parcial para obtenção do título de Tecnólogo em Curso Superior de Tecnologia em Segurança da Informação pelo Centro Paula Souza – FATEC Faculdade de Tecnologia de Americana – Ralph Biasi.  
Área de concentração: Segurança da Informação

Americana, 20 de junho de 2024

**Banca Examinadora:**

  
Mariana Godoy Vazquez Miano (Presidente)  
Pós-Doutorado em Engenharia de Produção  
Universidade Federal de São Carlos, UFSCar, Brasil

  
Ana Lúcia Spigolon (Membro)  
Pós graduação em Gestão de Projetos e Processos Organizacionais  
Centro Estadual de Educação Tecnológica Paula Souza, CEETEPS, Brasil.

  
Wagner Siqueira Cavalcante (Membro)  
Mestrado em Ciência da Computação.  
Universidade Federal de São Carlos, UFSCar, Brasil

---

## LISTA DE FIGURAS

---

Figura 1: VM – Configuração do Sistema .....	11
Figura 2: VM – Configuração do Monitor .....	12
Figura 3: Configuração da Rede .....	12
Figura 4: Instalação OpenSSL .....	15
Figura 5: Configuração da biblioteca liboqs .....	16
Figura 6: Instalação da biblioteca liboqs .....	17
Figura 7: Configuração oqsprovider .....	18
Figura 8: Instalação do oqsprovider .....	19
Figura 9: Configurado no OpenSSL o provedor oqsprovider .....	20
Figura 10: Export das variáveis padrão para novo diretório .....	21
Figura 11: Certificado RSA .....	23
Figura 12: Certificado Elliptic Curve P-256 .....	24
Figura 13: Certificado Dilithium3 .....	24
Figura 14: Gráfico do tempo total .....	25
Figura 15: Tempo de CPU do usuário .....	26
Figura 16: Gráfico de falhas de página .....	26
Figura 17: Gráfico das trocas de contexto .....	27
Figura 18: Perf do certificado RSA .....	30
Figura 19: Perf do Elliptic Curve P-256 .....	31
Figura 20: Perf do Dilithium3 .....	32
Figura 21: Gráfico do total de eventos por objeto .....	33
Figura 22: Gráfico distribuição dos objetos .....	33
Figura 23: Velocidade RSA .....	34

---

Figura 24: Velocidade Elliptic Curve P-256 .....	35
Figura 25: Velocidade Dilithium3 .....	35
Figura 26: Gráficos do teste de velocidade.....	35
Figura 27: Configuração de rede server.....	38
Figura 28: Configuração de rede client .....	38
Figura 29: Abrindo conexão com servidor <i>web</i> .....	39
Figura 30: Conexão com servidor <i>web</i> .....	39
Figura 31: Negociando criptografia AES256SHA384 da conexão com servidor <i>web</i> .....	40
Figura 32: Sessão da conexão com servidor <i>web</i> .....	41
Figura 33: Execução do Charon .....	43
Figura 34: Configuração swanctl.....	45
Figura 35: Configuração do swactl no cliente .....	45
Figura 36: Erro ao carregar chave Dilithium.....	46
Figura 37: Chave Dilithium carregada com sucesso .....	47
Figura 38: IPsec estabelecido .....	48

---

## SUMÁRIO

<b>1</b>	<b>OBJETIVO</b> .....	<b>5</b>
<b>2</b>	<b>REVISÃO BIBLIOGRÁFICA</b> .....	<b>6</b>
<b>3</b>	<b>DESENVOLVIMENTO</b> .....	<b>11</b>
3.1	Preparando o ambiente .....	11
3.1.1	Instalações .....	13
3.1.2	Configuração dos repositórios.....	14
3.2	Performance.....	21
3.2.1	Geração de Certificados Digitais .....	21
3.2.2	Registros .....	27
3.2.3	Velocidade no OpenSSL .....	34
<b>4</b>	<b>RESULTADOS</b> .....	<b>36</b>
<b>5</b>	<b>CONSIDERAÇÕES FINAIS DO DESEMPENHO</b> .....	<b>37</b>
<b>6</b>	<b>IMPLEMENTAÇÃO DO DILITHIUM 3</b> .....	<b>37</b>
6.1	Servidor <i>Web</i> com Dilithium3 .....	38
6.2	IPsec com certificado Dilithium3 .....	42
	<b>REFERÊNCIAS</b> .....	<b>50</b>



## 1 OBJETIVO

O objetivo central desse trabalho é explorar e apresentar estratégias eficazes para a transição gradual de algoritmos criptográficos convencionais para técnicas pós-quânticas no âmbito da comunicação pela Internet usando *Virtual Private Networks* (VPNs) e Aplicações *Web*. Esta proposta de pesquisa visa abordar a necessidade premente de adaptação da segurança da informação frente aos avanços nos estudos sobre computação quântica, destacando o papel crucial dos algoritmos de criptografia pós-quânticos na garantia de comunicações seguras.

A presente pesquisa é de cunho quantitativo, a partir da simulação de uma comunicação fictícia entre dois *hosts*, a fim de analisar de forma específica e comparativa dois tipos de criptografia de sistema assimétrico: o algoritmo *Elliptic Curve Cryptography* (ECC) ou Rivest-Shamir-Adleman (RSA) e o sistema pós-quântico utilizando *Crystal-Dilithium*, que foi escolhido como padrão de chave pública pós-quântica para criação de certificados pelo Instituto Nacional de Padrões e Tecnologia (NIST). O ECC e o RSA são comumente utilizados no protocolo *Internet Key Exchange* (IKE) para troca de chaves de forma segura, utilizando os certificados *Elliptic Curve Digital Signature Algorithm* (ECDSA) e do RSA. Assim, será feita a comparação de sua *performance* frente a uma possível e gradativa migração no *Internet Protocol Security* (IPsec) para compartilhar chaves secretas no túnel de forma autêntica e não vulnerável a ataques de algoritmos quânticos como o de Shor.

Será utilizada a aplicação OpenSSL com repositórios criados pelo projeto *Open Quantum Safe* (OQS) no GitHub com a biblioteca *oqsprovider* usada no OpenSSL v. 3.x. Esse é um projeto de código aberto com a finalidade de dar suporte para migração de tecnologias pós-quânticas seguras. Serão ainda utilizadas ferramentas de *performance* como *perf* e *OpenSSL-speed*.

Por fim, será realizada a simulação de servidor *web* simples, utilizando o certificado digital de criptografia pós-quântica entre duas máquinas, cliente e servidor, e realizada uma comunicação ponto a ponto, utilizando a aplicação *Strongswan* para estabelecer a comunicação pela VPN.



## 2 REVISÃO BIBLIOGRÁFICA

A criptografia é dividida em dois sistemas: chave privada e chave pública. Um sistema de criptografia (*cryptosystem*) é a transformação de um texto inteligível em um texto não inteligível (cifra), por meio de uma chave, e a decifragem da mensagem, pela mesma chave (simétrica) ou outra (assimétrica), pelo receptor. (Brassard, 1988).

O sistema simétrico foi o primeiro utilizado e comumente introduzido pela analogia de comunicação entre Alice e Bob. Quando Alice quer transmitir uma informação ao Bob sem que terceiros tenham conhecimento, ela usa uma chave secreta conhecida por ambos. Essa chave é utilizada tanto para cifrar quanto para decifrar a mensagem. Neste caso, o sistema é uma transformação invertível, ou seja, existe uma outra que, quando aplicada na mensagem cifrada, retorna a mensagem original que Alice quis transmitir (Stinson e Paterson, 2018).

Apesar de parecer uma ótima ideia manter uma comunicação segura, esse sistema tem um grande problema: Alice e Bob precisam trocar a sua chave secreta de forma estritamente privada além de mantê-la em segredo por todo o tempo da conversa, pois qualquer um em posse dela fará com que a confidencialidade seja perdida.

Assim, foi criado o mais conhecido *cryptosystem* de chave pública por Rivest, Shamir e Adleman (RSA) (Stinson e Paterson, 2019). O sistema de encriptação RSA é baseado na geração de dois grandes números primos que, quando multiplicados, facilmente é obtido o resultado. Entretanto a fatoração para obter um dos primos anteriores é inversamente proporcional em dificuldade (Hellman, 1978). Uma evolução para esse tipo de criptografia RSA foi a cifra através de Curva Elíptica, que também utiliza sistema assimétrico, que requer um tamanho de chave muito menor mantendo o nível de segurança (NSA, 2009). Esse tipo de cifragem é baseado em estrutura algébrica de corpos finitos formados por determinadas curvas elípticas na matemática.

No caso do método de Curva Elíptica, ao invés de usar dois números primos, sendo um deles chave pública e outra privada como é no RSA, são utilizados dois pontos  $A$  e  $B$  na curva como chaves públicas e um inteiro  $m$  como chave privada, sendo  $B = Am$ . Com isso, escolhendo um número aleatório  $k$ ,  $Bk$  será usado para cifrar o texto a ser transmitido (Stinson e Paterson 2018).

Apesar do novo sistema ser inovador e eficaz, existe a necessidade de criar um canal seguro para compartilhar chaves públicas, pois como o remetente saberá que a chave que estiver usando pertence ao seu destinatário confidente? Poderia um intermediário gerar outro par de chaves, compartilhar a pública com Alice, e ela, acreditando que sua mensagem será decriptada por Bob, poderá ser vista por quem estiver no meio da transmissão? Logo, será necessário criar autenticidade das chaves usando alguns métodos mais eficazes, como por exemplo, assinaturas digitais.

O sistema de assinaturas digitais, analogamente no contexto de chave pública de Alice e Bob, pode ser definido como: Bob precisa saber se a chave pública de Alice pertence de fato a ela. Assim, Alice cifraria sua mensagem com a chave privada que só pode ser descriptada por sua chave pública. Logo, Bob poderá garantir que a mensagem é de fato dela, já que ele conseguiria decifrar a mensagem com essa chave pública de Alice.

O objetivo do uso desses algoritmos é criar uma comunicação segura. Devido ao crescimento da estrutura de redes no mundo, as informações ficaram suscetíveis às más intenções de terceiros. Um dos avanços nesse tipo de comunicação deu início à VPN (*Virtual Private Networks*) que é a uma rede virtual através da internet, que permite comunicação privada. É uma espécie de túnel no meio de comunicação, em que somente pessoas autorizadas conseguem ver o conteúdo, ou seja, são os dados criptografados na rede. Além da criptografia dos dados, também são utilizados certificados digitais entre os *hosts* para garantir que a comunicação seja autêntica.

Um dos exemplos de aplicação dessas VPN é o IPsec, que é um padrão aberto, mantido pelo *Internet Engineering Task Force* (IETF) com o propósito de estabelecer um conjunto seguro de serviços para o tráfego na camada *Internet Protocol* (IP), abrangendo, tanto o IPv4, quanto o IPv6 (Cheng et al., 1998).

Esse protocolo é subdividido em três componentes: *Authentication Header* (AH), *Encapsulation Security Payload* (ESP) e *Internet Security Association e Key Management Protocol* (ISAKMP). É amplamente empregado em conexões de redes virtuais privadas (VPNs) para criar uma comunicação segura entre dois *hosts* (denominados *gateways*), assegurando que



os dados não sejam interceptados por terceiros. Mesmo se houver tentativas de coleta de dados por esses terceiros, as informações permanecerão criptografadas (Kent e Seo, 2005).

Para estabelecer uma comunicação segura entre dois computadores, é necessário compartilhar chaves pública e privada entre os *hosts* na camada de rede, além de definir parâmetros em comum, como algoritmos de criptografia, método de integridade dos dados e *time to live*, que é a vida útil da comunicação. O protocolo *Internet Key Exchange* (IKE) é o responsável pela troca de chaves entre os *hosts*, que foi aprimorado no IPsec do *ikev1* para o *ikev2*, corrigindo fraquezas e vulnerabilidades.

Para Preskill (2018), por muitos anos esse tipo de comunicação teve garantia de segurança para empresas e clientes da *internet* em geral. Entretanto, com o grande avanço da ciência e influência dos princípios de grandes físicos na mecânica quântica, e necessidade de otimização para processar alta quantidade de dados, foram desenvolvidos os computadores quânticos. Esses computadores têm a capacidade de realizar cálculos de alto processamento, utilizando conceitos avançados como superposição e emaranhamento. O emaranhamento é a relação intrínseca entre dois *bits* quânticos, onde o estado de um está relacionado com o do outro. Já a superposição é a capacidade do *bit* quântico poder ter dois estados ao mesmo tempo, permitindo realizar cálculos simultâneos, com alto poder de processamento e armazenamento de dados.

Como cita Preskill (2018), para compreender a fundo o funcionamento desses sistemas, é importante definir o que é um *qubit* e entender seu papel no processamento de dados quânticos. Enquanto um *bit* clássico é, de maneira simplificada, uma variável que pode assumir os estados '0' ou '1', permitindo operações lógicas booleanas (XOR, AND, OR), no contexto quântico, os *qubits* transcendem essa dualidade, podendo existir nos dois estados simultaneamente. Isso não apenas amplia as possibilidades de processamento, mas também adiciona complexidade e sutileza à computação quântica.

Uma das aplicações mais notáveis dos computadores quânticos reside na sua eficiente capacidade de fatoração de números primos. Por meio de algoritmos quânticos avançados, como o de Shor, esses dispositivos podem desvendar criptografias de chaves públicas, como as utilizadas nos sistemas RSA ou ECC. Essa habilidade representa uma potencial ameaça à



segurança de dados confidenciais, uma vez que as técnicas criptográficas tradicionais podem ser comprometidas (Shor, 1994).

Assim, houve uma busca para solucionar o problema através da criptografia pós-quântica, que tem como foco desenvolver métodos seguros contra os ataques de computadores quânticos. São algoritmos matemáticos desenvolvidos para não serem resolvidos até mesmo com a potência de um computador quântico.

Dentre eles, os principais foram escolhidos pelo NIST numa maratona de padronização das criptografias pós-quântica, lançada em 2016. O algoritmo Dilithium, por exemplo, é aplicado usando o esquema de assinatura digital. Ele é baseado na estrutura matemática de *lattice*, que é um espaço vetorial de vetores discretos, além do uso de técnicas de *hash*, que pega qualquer dado e o converte em um código fixo único, garantindo que os dados não sejam alterados e protegendo sua integridade. Essa estrutura matemática tem propriedades interessantes para criptografia, como o ‘problema do vetor mais curto’, onde dado um conjunto de vetores que geram um reticulado (*lattice*), deve-se encontrar uma combinação linear do vetor não nulo tal que nenhum vetor é menor que ele. Um problema tão difícil de resolver, torna inviável a prática computacional de encontrar esse vetor. Assim sendo, o algoritmo Dilithium torna-se seguro devido à complexidade do problema onde ele é sustentado (Micciancio e Regev, 2008; BAI, Shi, et. al. 2023).

Dessa forma, com o fim de realizar uma migração gradual entre as criptografias modernas para pós-quântica, o uso de ferramentas como o OpenSSL facilitará o processo. Ele é um software de criptografia amplamente utilizado pela internet para criar e lidar com certificados e arquivos relacionados, possui uma biblioteca de criptografia extensa e abrangente (*libcrypto*) e realiza conexões entre servidores usando os protocolos *Security Sockets Layer/Transport Layer Security* (SSL/TLS).

Nesse contexto, surge um projeto de código aberto recente chamado *Open Quantum Safe* (OQS), cujo objetivo é implementar algoritmos pós-quânticos seguros. Combinado com a aplicação OpenSSL, é facilitada a transição entre os dois tipos de cifragem. De forma semelhante ao OpenSSL, o OQS é criado por uma biblioteca de criptografia pós-quântica, bem como um conjunto de ferramentas e protocolos que permitem a integração dessas bibliotecas



ao OpenSSL. Seu desenvolvimento fica nos repositórios do GitHub e há diversas aplicações de criptografia pós-quântica em ferramentas como OpenSSH, Python, Rust, entre outros.

A partir do projeto OQS, surgiram diversos projetos de terceiros, destinados a implementar a criptografia pós-quântica em programas de segurança integrados com o OQS. Um exemplo é a implementação na aplicação de código aberto *Strongswan*, utilizando a biblioteca *liboqs* do OQS. Essa ferramenta permite a criação de uma comunicação via VPN usando o protocolo IKE no IPsec, proporcionando a proteção do tráfego baseado em políticas e roteamento de rede. Através dessa colaboração entre o *Strongswan* e a biblioteca *liboqs*, torna-se viável a criação dos túneis seguros utilizando criptografia pós-quântica.

Finalmente, será feita uma análise comparativa entre as criptografias atuais como RSA e EC com a criptografia pós-quântica, Dilithium, a fim de demonstrar seus desempenhos de uso de CPU, memória e o tempo levado para executar suas funções. Esta análise será conduzida pela ferramenta *Perf*, que é uma aplicação própria do Linux que desempenha um papel crucial nessa avaliação, fornecendo métricas detalhadas sobre o desempenho do sistema para os três tipos de criptografia. Na sequência, será realizada uma simulação entre dois *hosts* utilizando certificados digitais do Dilithium para garantir sua autenticidade. Isso será feito por meio do OpenSSL, integrado com o OQS, utilizado, tanto em um servidor *web*, quanto para criação de uma VPN utilizando o *Strongswan*. Isso permitirá examinar de forma mais abrangente a aplicabilidade prática e a segurança dessas aplicações em cenários reais. Essa abordagem permitirá compreender não somente o desempenho técnico das criptografias, mas também avaliar sua viabilidade e efetividade em operações do dia a dia.

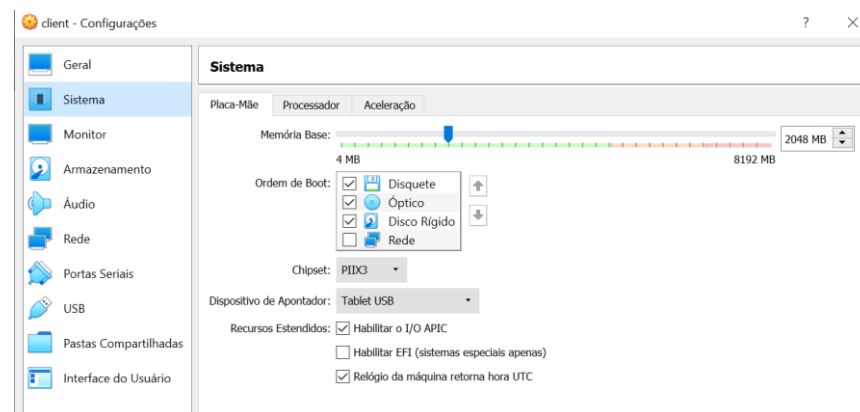
### 3 DESENVOLVIMENTO

#### 3.1 Preparando o ambiente

Duas máquinas virtuais, com sistema operacional Linux Ubuntu 5.15.0-91-generic x86\_64, foram utilizadas para simulação de comunicação ponto a ponto de forma segura, sendo uma chamada de *client* e a outra *server* (clone do cliente), como mostrado nas figuras 1, 2 e 3.

Máquina configurada com memória de 2048 *Megabytes*, *chipset* PIX3 e os outros itens padrões do Oracle, conforme figura 1:

Figura 1: VM – Configuração do Sistema

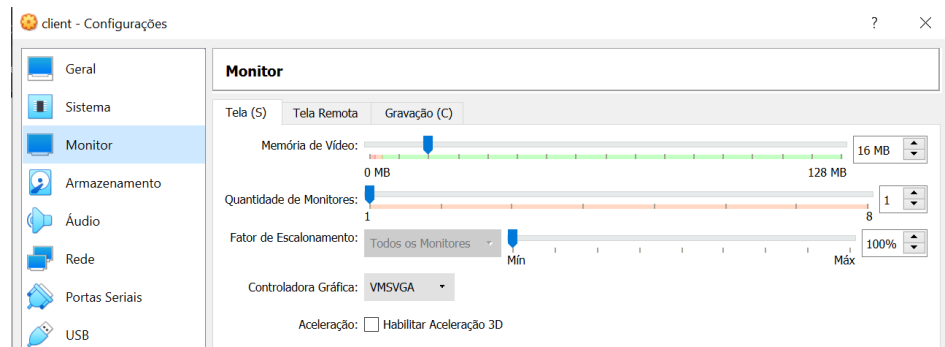


Fonte: Autor, 2024

A figura 2 mostra como foi configurada a memória de vídeo padrão do VirtualBox de 16 *Megabytes*, com 1 monitor e controle gráfico VMSVGA:



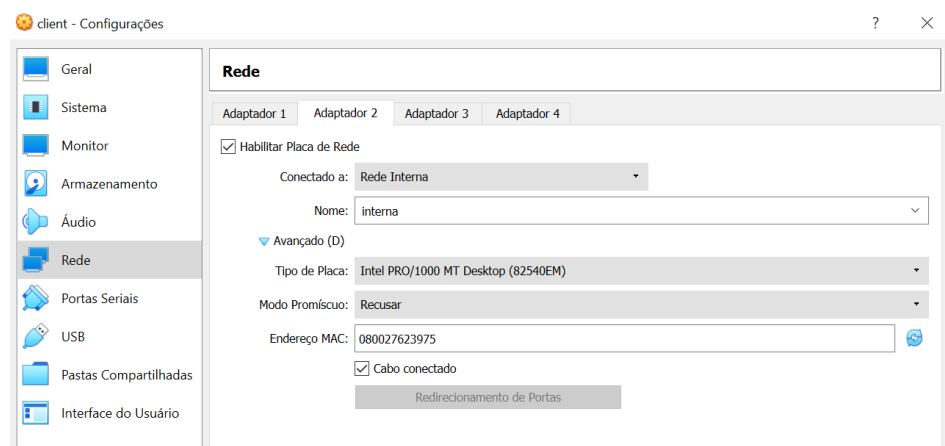
Figura 2: VM – Configuração do Monitor



Fonte: Autor, 2024

A figura 3 mostra a configuração da placa de rede interna para comunicação entre os *hosts*:

Figura 3: Configuração da Rede



Fonte: Autor, 2024

Na sequência, serão apresentados as instalações e configurações dos pacotes e bibliotecas necessárias.



### 3.1.1 Instalações

As instalações das ferramentas necessárias no ambiente serão o OpenSSL, bibliotecas do repositório OQS e instalação da aplicação *perf* do Linux para análise do desempenho dos algoritmos.

#### 3.1.1.1 OpenSSL e repositórios Open Quantum Safe:

Primeiro será necessário atualizar todo sistema operacional e instalar pacotes necessários para configuração dos repositórios do GitHub:

```
apt update -y && apt install git build-essential perl cmake autoconf libtool zlib1g-dev -y
```

#### 3.1.1.2 Instalação de ferramentas e dependências do perf:

Na ferramenta de desempenho, o comando abaixo foi utilizado para instalação dos pacotes necessários:

```
apt install linux-tools-5.15.0-91-generic linux-cloud-tools-5.15.0-91-generic -y
```

Configurou-se o ambiente de trabalho na pasta *quantum* adicionando variáveis *WORKSPACE*, *BUILD\_DIR* para realizar as configurações e *linkando* o diretório *lib64* com *lib*:

```
export WORKSPACE=~/.quantum
export BUILD_DIR=$WORKSPACE/build
ln -s $BUILD_DIR/lib64 $BUILD_DIR/lib
```



### 3.1.2 *Configuração dos repositórios*

Com a instalação das dependências realizadas e preparado o ambiente, será realizado as configurações dos repositórios das ferramentas OpenSSL integrando com a biblioteca liboqs do OQS.

#### 3.1.2.1 *OpenSSL 3.x*

A instalação deve ser a partir da versão 3 para que haja compatibilidade em usar os repositórios do projeto Open Quantum Safe. Foram desinstaladas versões até 1.2 do TLS (Transport Layer Security) que não suportam os algoritmos quânticos:

```
cd $WORKSPACE
git clone https://github.com/OpenSSL/OpenSSL.git
cd OpenSSL
./Configure \
    --prefix=$BUILD_DIR
    no-ssl no-tls1 no-tls1_1 no-afalgeng \
    no-shared threads -lm
make -j $(nproc)
make -j $(nproc) install_sw install_ssldirs
```

Resultado é demonstrado conforme figura 4:

Figura 4: Instalação OpenSSL

```
Cloning into 'openssl'...
remote: Enumerating objects: 473908, done.
remote: Counting objects: 100% (601/601), done.
remote: Compressing objects: 100% (370/370), done.
remote: Total 473908 (delta 410), reused 333 (delta 230), pack-reused 473307
Receiving objects: 100% (473908/473908), 215.25 MiB | 1.49 MiB/s, done.
Resolving deltas: 100% (343661/343661), done.
mint@client-VM:~/quantum$ cd openssl/
mint@client-VM:~/quantum/openssl$
mint@client-VM:~/quantum/openssl$
mint@client-VM:~/quantum/openssl$ ./Configure \
--prefix=$BUILD_DIR \
no-ssl no-tls1 no-tls1_1 no-afalgeng \
no-shared threads -lm
make -j $(nproc)
make -j $(nproc) install sw install ssldirs
```

Fonte: Autor, 2024

### 3.1.2.2 Biblioteca liboqs

Essa é uma biblioteca em linguagem C, disponibilizada pelo OQS para implementação dos diversos algoritmos de criptografia pós-quântica.

Foram modificadas as configurações padrão `BUILD_SHARED_LIBS` e `OQS_USE_OPENSSL` para serem usados os arquivos locais do OpenSSL, instalados anteriormente.

Além disso, foi configurado somente o provedor `oqsprovider`, através da opção `OQS_BUILD_ONLY_LIB=ON`.

```
cd $WORKSPACE
git clone https://github.com/open-quantum-safe/liboqs.git
cd liboqs
mkdir build && cd build
cmake \
-DCMAKE_INSTALL_PREFIX=$BUILD_DIR \
-DBUILD_SHARED_LIBS=ON \
-DOQS_USE_OPENSSL=OFF \
```

```
-DCMAKE_BUILD_TYPE=Release \  
-DOQS_BUILD_ONLY_LIB=ON \  
-DOQS_DIST_BUILD=ON \  
..  
make -j $(nproc)  
make -j $(nproc) install
```

A figura 5 demonstra o resultado dos comandos:

Figura 5: Configuração da biblioteca liboqs

```
mint@client-VM:~/quantum/liboqs/build$ cmake \  
-DCMAKE_INSTALL_PREFIX=$BUILD_DIR \  
-DBUILD_SHARED_LIBS=ON \  
-DOQS_USE_OPENSSL=OFF \  
-DCMAKE_BUILD_TYPE=Release \  
-DOQS_BUILD_ONLY_LIB=ON \  
-DOQS_DIST_BUILD=ON \  
..  
-- The C compiler identification is GNU 11.4.0  
-- The ASM compiler identification is GNU  
-- Found assembler: /usr/bin/cc  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: /usr/bin/cc - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Performing Test CC_SUPPORTS_WA_NOEXECSTACK  
-- Performing Test CC_SUPPORTS_WA_NOEXECSTACK - Success  
-- Performing Test LD_SUPPORTS_WL_Z_NOEXECSTACK  
-- Performing Test LD_SUPPORTS_WL_Z_NOEXECSTACK - Success  
-- Looking for pthread.h  
-- Looking for pthread.h - found  
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD  
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success  
-- Found Threads: TRUE  
-- Alg enablement unchanged  
-- Looking for getentropy  
-- Looking for getentropy - found  
-- Looking for aligned_alloc  
-- Looking for aligned_alloc - found  
-- Looking for posix_memalign  
-- Looking for posix_memalign - found  
-- Looking for memalign  
-- Looking for memalign - found  
-- Looking for explicit_bzero  
-- Looking for explicit_bzero - found  
-- Looking for memset_s  
-- Looking for memset_s - not found  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/mint/quantum/liboqs/build  
mint@client-VM:~/quantum/liboqs/build$
```

Fonte: Autor, 2024

Finalizada a instalação com sucesso da biblioteca, conforme figura 6:

Figura 6: Instalação da biblioteca liboqs

```
[100%] Built target oqs
Consolidate compiler generated dependencies of target internal
[100%] Built target internal
[100%] Built target oqs-internal
Install the project...
-- Install configuration: "Release"
-- Installing: /home/mint/quantum/build/lib/cmake/liboqs/liboqsConfig.cmake
-- Installing: /home/mint/quantum/build/lib/cmake/liboqs/liboqsConfigVersion.cmake
-- Installing: /home/mint/quantum/build/lib/pkgconfig/liboqs.pc
-- Installing: /home/mint/quantum/build/lib/liboqs.so.0.10.0-dev
-- Installing: /home/mint/quantum/build/lib/liboqs.so.5
-- Installing: /home/mint/quantum/build/lib/liboqs.so
-- Installing: /home/mint/quantum/build/lib/cmake/liboqs/liboqsTargets.cmake
-- Installing: /home/mint/quantum/build/lib/cmake/liboqs/liboqsTargets-release.cmake
-- Installing: /home/mint/quantum/build/include/oqs/oqs.h
-- Installing: /home/mint/quantum/build/include/oqs/common.h
-- Installing: /home/mint/quantum/build/include/oqs/rand.h
-- Installing: /home/mint/quantum/build/include/oqs/kem.h
-- Installing: /home/mint/quantum/build/include/oqs/sig.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_bike.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_frodokey.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_ntruprime.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_classic_mceliece.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_hqc.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_kyber.h
-- Installing: /home/mint/quantum/build/include/oqs/kem_ml_kem.h
-- Installing: /home/mint/quantum/build/include/oqs/sig_dilithium.h
-- Installing: /home/mint/quantum/build/include/oqs/sig_ml_dsa.h
-- Installing: /home/mint/quantum/build/include/oqs/sig_falcon.h
-- Installing: /home/mint/quantum/build/include/oqs/sig_sphincs.h
-- Installing: /home/mint/quantum/build/include/oqs/oqsconfig.h
mint@client-VM:~/quantum/liboqs/build$
```

Fonte: Autor, 2024

Terminadas as instalações das bibliotecas necessárias para utilizar o conjunto de criptografias pós-quântica dentro do OpenSSL, resta a instalação de um provedor com os algoritmos e ferramentas para sua utilização.

### 3.1.2.3 Provedor *oqsprovider* para *OpenSSL v. 3.x*

Esse provedor é um conjunto de algoritmos de criptografia para uso pelo OpenSSL. Ele normalmente é chamado com o parâmetro *-provider* ou *-path-provider* indicando os algoritmos que serão implementados no protocolo.

Nas configurações, foram criadas variáveis até os caminhos raiz do OpenSSL e liboqs:

```
cd $WORKSPACE

git clone https://github.com/open-quantum-safe/oqs-provider.git

cd oqs-provider

liboqs_DIR=$BUILD_DIR cmake \

-DCMAKE_INSTALL_PREFIX=$WORKSPACE/oqs-provider \

-DOPENSSL_ROOT_DIR=$BUILD_DIR \

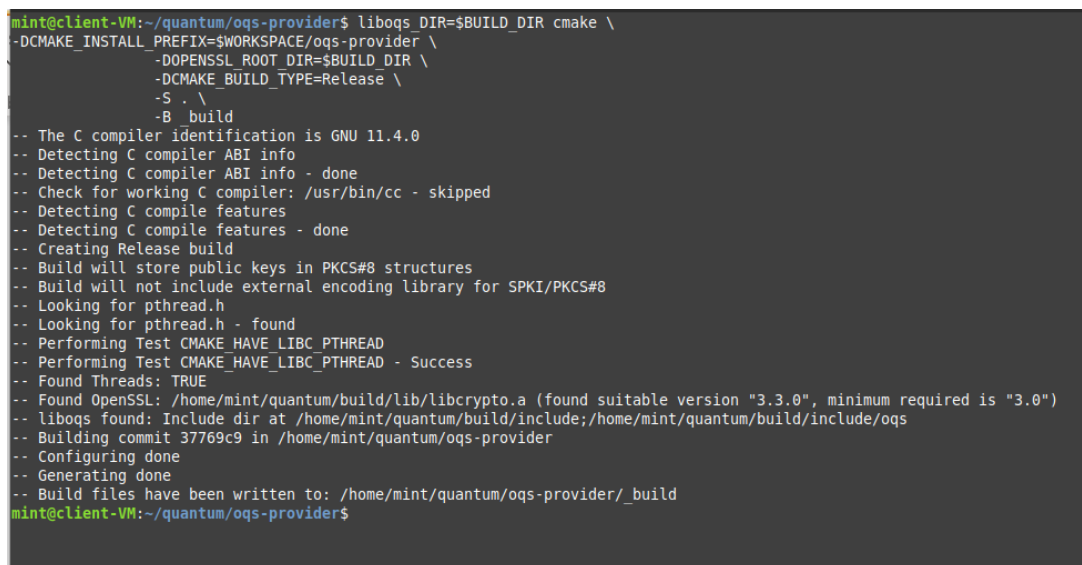
-DCMAKE_BUILD_TYPE=Release \

-S. \

-B _build
```

Como pode ser visto na figura 7:

Figura 7: Configuração oqsprovider



```
mint@client-VM:~/quantum/oqs-provider$ liboqs_DIR=$BUILD_DIR cmake \
-DCMAKE_INSTALL_PREFIX=$WORKSPACE/oqs-provider \
-DOPENSSL_ROOT_DIR=$BUILD_DIR \
-DCMAKE_BUILD_TYPE=Release \
-S. \
-B _build
-- The C compiler identification is GNU 11.4.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Creating Release build
-- Build will store public keys in PKCS#8 structures
-- Build will not include external encoding library for SPKI/PKCS#8
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- Found OpenSSL: /home/mint/quantum/build/lib/libcrypto.a (found suitable version "3.3.0", minimum required is "3.0")
-- liboqs found: Include dir at /home/mint/quantum/build/include;/home/mint/quantum/build/include/oqs
-- Building commit 37769c9 in /home/mint/quantum/oqs-provider
-- Configuring done
-- Generating done
-- Build files have been written to: /home/mint/quantum/oqs-provider/_build
mint@client-VM:~/quantum/oqs-provider$
```

Fonte: Autor, 2024

Na figura 8, observa-se o `'cmake --build _build'` para construir os repositórios e executáveis configurados anteriormente na pasta “`_build`” como pode ser visto na última linha do resultado na Figura 7.

Figura 8: Instalação do oqsprovider

```

mint@client-VM:~/quantum/oqs-provider$ cmake --build _build
[ 3%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqsprov.c.o
[ 6%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqsprov_capabilities.c.o
[ 9%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqsprov_keys.c.o
[ 12%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_kmgmt.c.o
[ 16%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_sig.c.o
[ 19%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_kem.c.o
[ 22%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_encode_key2any.c.o
[ 25%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_encoder_common.c.o
[ 29%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqs_decode_der2key.c.o
[ 32%] Building C object oqsprov/CMakeFiles/oqsprovider.dir/oqsprov_bio.c.o
[ 35%] Linking C shared module ../lib/oqsprovider.so
[ 35%] Built target oqsprovider
[ 38%] Building C object test/CMakeFiles/oqs_test_signatures.dir/oqs_test_signatures.c.o
[ 41%] Building C object test/CMakeFiles/oqs_test_signatures.dir/test_common.c.o
[ 45%] Linking C executable oqs_test_signatures
[ 45%] Built target oqs_test_signatures
[ 48%] Building C object test/CMakeFiles/oqs_test_kems.dir/oqs_test_kems.c.o
[ 51%] Building C object test/CMakeFiles/oqs_test_kems.dir/test_common.c.o
[ 54%] Linking C executable oqs_test_kems
[ 54%] Built target oqs_test_kems
[ 58%] Building C object test/CMakeFiles/oqs_test_groups.dir/oqs_test_groups.c.o
[ 61%] Building C object test/CMakeFiles/oqs_test_groups.dir/test_common.c.o
[ 64%] Building C object test/CMakeFiles/oqs_test_groups.dir/tltest_helpers.c.o
[ 67%] Linking C executable oqs_test_groups
[ 67%] Built target oqs_test_groups
[ 70%] Building C object test/CMakeFiles/oqs_test_tlssig.dir/oqs_test_tlssig.c.o
[ 74%] Building C object test/CMakeFiles/oqs_test_tlssig.dir/test_common.c.o
[ 77%] Building C object test/CMakeFiles/oqs_test_tlssig.dir/tltest_helpers.c.o
[ 80%] Linking C executable oqs_test_tlssig
[ 80%] Built target oqs_test_tlssig
[ 83%] Building C object test/CMakeFiles/oqs_test_encode.dir/oqs_test_encode.c.o
[ 87%] Building C object test/CMakeFiles/oqs_test_encode.dir/test_common.c.o
[ 90%] Linking C executable oqs_test_encode
[ 90%] Built target oqs_test_encode
[ 93%] Building C object test/CMakeFiles/oqs_test_evp_pkey_params.dir/oqs_test_evp_pkey_params.c.o
[ 96%] Building C object test/CMakeFiles/oqs_test_evp_pkey_params.dir/test_common.c.o
[100%] Linking C executable oqs_test_evp_pkey_params
[100%] Built target oqs_test_evp_pkey_params
mint@client-VM:~/quantum/oqs-provider$

```

Fonte: Autor, 2024

Foi realizada cópia manual das bibliotecas *\_build* do oqs-provider para *build* raiz *~quantum/build*:

```
cp _build/lib/* $BUILD_DIR/lib/
```

Isso permite que as criptografias pós quânticas do provedor oqsprovider sejam utilizadas pelo OpenSSL nos módulos que foram construídos em *\$BUILD\_DIR/lib*.

Por fim foi realizado a alteração das configurações do OpenSSL para ativar o provedor oqsprovider:

```
sed -i "s/default = default_sect/default = default_sect\noqsprovider = oqsprovider_sect/g"
$BUILD_DIR/ssl/openssl.cnf
```

```
sed -i "s/[default_sect]/[default_sect]\nactivate = 1\n[oqsprovider_sect]\nactivate = 1\n/g"
$BUILD_DIR/ssl/openssl.cnf
```

Na figura 9 é demonstrado a execução dos comandos:

Figura 9: Configurado no OpenSSL o provedor oqsprovider

```
mint@client-VM:~$ sed -i "s/default = default_sect/default = default_sect\noqsprovider = oqsprovider_sect/g" $BU
ILD_DIR/ssl/openssl.cnf
mint@client-VM:~$
mint@client-VM:~$ sed -i "s/[default_sect]/[default_sect]\nactivate = 1\n[oqsprovider_sect]\nactivate = 1\
n/g" $BUILD_DIR/ssl/openssl.cnf
mint@client-VM:~$
```

Fonte: Autor, 2024

Para o OpenSSL usar o provedor que foi instalado, foi necessário modificar as variáveis padrão *OPENSSL\_CONF* e *OPENSSL\_MODULES* para o ambiente construído:

```
export OPENSSL_CONF=$BUILD_DIR/ssl/openssl.cnf
```

```
export OPENSSL_MODULES=$BUILD_DIR/lib
```

```
openssl list -providers -verbose
```

Pode ser visto na figura 10 o resultado dos comandos:

Figura 10: Export das variáveis padrão para novo diretório

```

mint@client-VM:~/quantum/oqs-provider$ export OPENSSL_CONF=$BUILD_DIR/ssl/openssl.cnf
export OPENSSL_MODULES=$BUILD_DIR/lib
openssl list -providers -verbose
Providers:
  default
    name: OpenSSL Default Provider
    version: 3.0.2
    status: active
    build info: 3.0.2
    gettable provider parameters:
      name: pointer to a UTF8 encoded string (arbitrary size)
      version: pointer to a UTF8 encoded string (arbitrary size)
      buildinfo: pointer to a UTF8 encoded string (arbitrary size)
      status: integer (arbitrary size)
  oqsprovider
    name: OpenSSL OQS Provider
    version: 0.5.4-dev
    status: active
    build info: OQS Provider v.0.5.4-dev (37769c9) based on liboqs v.0.10.0-dev
    gettable provider parameters:
      name: pointer to a UTF8 encoded string (arbitrary size)
      version: pointer to a UTF8 encoded string (arbitrary size)
      buildinfo: pointer to a UTF8 encoded string (arbitrary size)
      status: integer (arbitrary size)
mint@client-VM:~/quantum/oqs-providers

```

Fonte: Autor, 2024

A figura 10 demonstra que o provedor *oqsprovider* já está listado e ativado, junto com o *default*, para utilização pelo OpenSSL.

### 3.2 Performance

Com o cenário pronto, foram realizados testes comparativos em *performance* e utilização de CPU pelos algoritmos de criptografia. Inicialmente, foram geradas chaves e certificados baseados nos algoritmos RSA, ECDSA com a curva Prime-256 e o pós-quântico Dilithium3, tendo em vista que essas são as versões e tamanho de chaves consideradas seguras pelo NIST e desenvolvedores dos algoritmos. Cada tabela e gráfico detalham os tempos de execução, trocas de contexto, *page-faults* e outras métricas relevantes para cada algoritmo de criptografia avaliado.

#### 3.2.1 Geração de Certificados Digitais

Para avaliar a *performance* na geração de certificados digitais, o estudo utilizou entre outros, o programa *perf-stat*, que fornece informações detalhadas sobre o desempenho do sistema. As métricas analisadas foram:



**a. Tempo de CPU:**

Tempo total gasto pelo processo na CPU, dividido em tempo de usuário (*user*) e tempo de sistema (*sys*).

**b. Trocas de Contexto (*context-switches*):**

Número de vezes que a CPU trocou de contexto para executar outro processo.

**c. Falhas de Página (*page-faults*):**

Número de vezes que o sistema operacional precisou acessar a memória virtual (disco) para atender a uma solicitação de memória do processo.

**d. Tempo Decorrido (*time elapsed*):**

Tempo total decorrido desde o início da execução do processo.

Para visualizar os resultados dessas métricas e entender melhor o desempenho de cada algoritmo de criptografia, serão apresentados os comandos utilizados para verificar a *performance* de cada criptografia.

A análise dessas métricas permite uma compreensão mais aprofundada do desempenho de cada algoritmo de geração de certificados digitais.

Para criação do certificado digital usando RSA é utilizado o comando:

```
openssl req -x509 -new -newkey rsa -keyout rsa_CA.key -out resa_CA.crt -nodes -subj "/CN=rsa CA" -days 365
```

Na Figura 11 está o resultado da *performance*:



Figura 11: Certificado RSA

```

160,85 msec task-clock                #   0,434 CPUs utilized
  1.100   context-switches            #   6,839 K/sec
    0     cpu-migrations              #   0,000 /sec
    826   page-faults                 #   5,135 K/sec
<not supported> cycles
<not supported> instructions
<not supported> branches
<not supported> branch-misses

  0,370874355 seconds time elapsed

  0,139431000 seconds user
  0,021451000 seconds sys

```

Fonte: Autor, 2024

Para gerar o certificado usando RSA, foi utilizado bastante recurso de CPU, sendo 160,85ms para execução da tarefa, alta troca de contextos sugerindo bastante tempo ocioso para execução. As outras métricas não são suportadas devido a utilização de VM limitando os recursos mais detalhados.

Foi gerado o certificado usando algoritmo de curvas elípticas e calculado sua *performance* pelo comando:

```

openssl req -x509 -new -newkey ec -pkeyopt ec_paramgen_curve:P-256 -keyout ECDSA_CA.key
-out ECDSA_CA.crt -nodes -subj "/CN=ECDSA CA" -days 365

```

Na figura 12 vemos o resultado para gerar esse certificado:

Figura 12: Certificado Elliptic Curve P-256

```

21,86 msec task-clock          #    0,387 CPUs utilized
    18      context-switches   #   823,241 /sec
     0      cpu-migrations     #    0,000 /sec
    828     page-faults        #   37,869 K/sec
<not supported>              cycles
<not supported>              instructions
<not supported>              branches
<not supported>              branch-misses

0,056523880 seconds time elapsed

0,022435000 seconds user
0,000000000 seconds sys

```

Fonte: Autor, 2024

Para gerar certificados com curva elíptica, a utilização de tempo para execução da tarefa foi aproximadamente oito vezes menor (21,86ms), demonstrando maior eficiência em relação ao RSA, além de poucas trocas de contextos.

Por fim, para criar o certificado digital pós-quântico dilithium3 utilizou-se o comando:

```
openssl req -x509 -new -newkey dilithium3 -keyout dilithium3_CA.key -out dilithium3_CA.crt -nodes -sub /CN=dilithium CA -days 365
```

A Figura 13 mostra as estatísticas da geração do certificado:

Figura 13: Certificado Dilithium3

```

18,64 msec task-clock          #    0,948 CPUs utilized
    20      context-switches   #    1,073 K/sec
     0      cpu-migrations     #    0,000 /sec
    856     page-faults        #   45,913 K/sec
<not supported>              cycles
<not supported>              instructions
<not supported>              branches
<not supported>              branch-misses

0,019673710 seconds time elapsed

0,018967000 seconds user
0,000000000 seconds sys

root@client-VM: /home/mint#

```

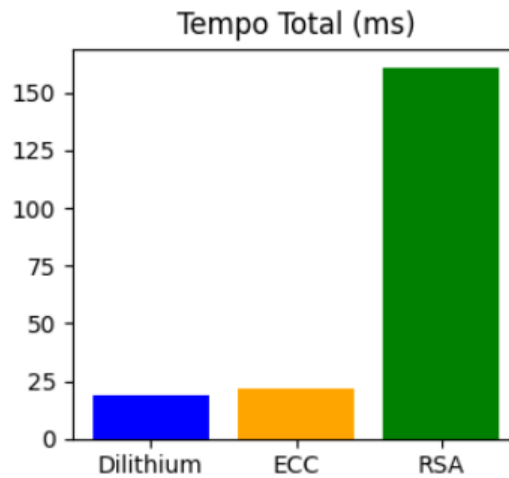
Fonte: Autor, 2024

O algoritmo Dilithium teve utilização de CPU e trocas de contextos semelhantes ao ECC, mas ainda com tempo menor (18,64ms), mostrando ainda melhor eficiência para gerar certificados pelo OpenSSL.

Os gráficos nas figuras 14, 15, 16 e 17 comparam os algoritmos em relação às principais métricas.

Na figura 14 é mostrado que o algoritmo RSA necessita de muito mais tempo para execução em comparação ao Dilithium e ECC:

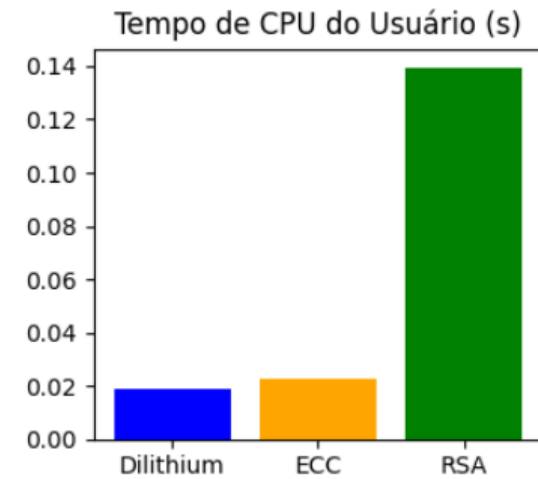
Figura 14: Gráfico do tempo total



Fonte: Autor, 2024

Para utilização de CPU do usuário que executa o código, os algoritmos Dilithium e ECC necessitam de menos quantidade de recurso em comparação ao RSA, como mostra a figura 15:

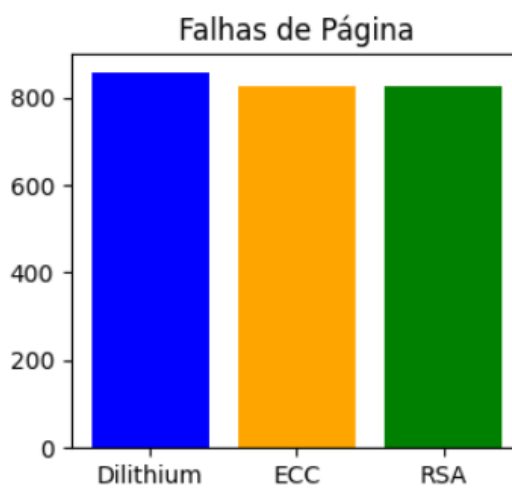
Figura 15: Tempo de CPU do usuário



Fonte: Autor, 2024

A figura 16 demonstra que as falhas de página são equilibradas entre os certificados:

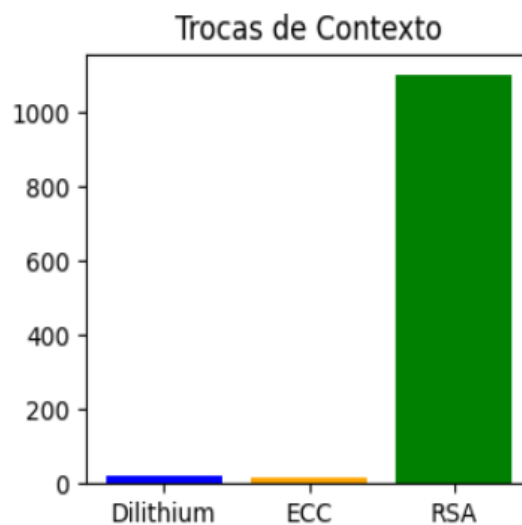
Figura 16: Gráfico de falhas de página



Fonte: Autor 2024

Em relação à quantidade de vezes que a CPU realizou trocas de processos durante execução do código, a figura 17 mostra que o algoritmo RSA consome muito mais processamento para isso:

Figura 17: Gráfico das trocas de contexto



Fonte: Autor, 2024

A análise dessas métricas demonstra que os algoritmos Dilithium e ECC são mais eficientes em termos de tempo de execução e utilização de CPU em comparação ao RSA. Esse resultado será explorado com mais detalhes através dos registros das funções para criar os certificados digitais, proporcionando uma visão mais aprofundada sobre o desempenho de cada algoritmo.

### 3.2.2 Registros

O comando *perf record* foi empregado para capturar a frequência de chamadas de funções durante o processo de geração de certificados. Essas métricas oferecem resultados valiosos sobre a alocação de recursos e a execução de partes específicas do código por cada algoritmo, permitindo uma avaliação comparativa de sua eficiência em relação aos demais.



Os principais resultados analisados serão as medidas de *Samples*, *Event Count*, Objeto Compartilhado pelas funções e *Overhead* utilizados em cada uma delas. Na sequência são apresentados os principais detalhes desses aspectos.

### 3.2.2.1 *Overhead*

No contexto de análise de desempenho com o comando “*perf record*”, o *overhead* indica quanto recurso adicional é consumido durante a execução das funções monitoradas.

### 3.2.2.2 *Samples*

Um *sample* é uma amostra do estado do sistema em um determinado momento durante a execução do programa. Cada amostra geralmente contém informações sobre a função atualmente em execução, o endereço de instrução, os contadores de desempenho e outros detalhes relevantes. O “*perf record*” captura uma série de amostras ao longo do tempo para análise posterior.

### 3.2.2.3 *Event Counts*

Os *event counts* são contadores que registram o número de vezes que eventos específicos ocorreram durante a execução do programa. Esses eventos podem incluir chamadas de função, instruções executadas, *cache misses*, acessos à memória, entre outros.

### 3.2.2.4 *Shared Object*

Um *shared object* (objeto compartilhado) refere-se a um arquivo de biblioteca que contém código executável compartilhado por várias funções. Neste caso os objetos utilizados para geração de certificados digitais através do OpenSSL são:

a. `libc.so.6`

Essa é a biblioteca do GNU em linguagem C, sendo utilizada pelo comando para operações básicas de entrada/saída, alocação de memória, manipulação de *strings* etc;

b. `kernel.kallsyms`

Este arquivo contém o mapeamento dos símbolos do *kernel* do Linux e é usado pela geração de certificados para acessar funções, *strings* e estruturas definidas no *kernel*;



c. `ld-linux-x86-64.so.2`

Este é o *linker* dinâmico do Linux para arquiteturas x86-64. Ele é responsável por carregar bibliotecas compartilhadas em tempo de execução do OpenSSL;

d. `libcrypto.so.3`

Esta é a biblioteca criptográfica do OpenSSL. Ela fornece funções para criptografia simétrica e assimétrica, geração de números pseudoaleatórios, *hashes*, sendo essencial para a geração de certificados.

### 3.2.2.5 Funções

As funções representam blocos de código específicos que executam tarefas ou operações dentro do programa.

Nas figuras 18, 19 e 20, foram analisados os registros de geração de certificados digitais de cada algoritmo e posteriormente criação de gráficos, como nas figuras 18 e 19, mostrando a comparação dos resultados baseado na frequência do uso de objetos para execução do código e quantidade de eventos necessários para isso.

Observa-se na figura 18, representando o algoritmo RSA, que há bastante utilização do objeto compartilhado `libcrypto.so.3` em diferentes funções:



Figura 18: Perf do certificado RSA

```

root@client-VM:/home/mint# perf report -i rsa_record | cat
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 124 of event 'cpu-clock:pppH'
# Event count (approx.): 3100000
#
# Overhead Command Shared Object Symbol
# .....
#
15.32% openssl libcrypto.so.3 [.] BN_consttime_swap
12.90% openssl [kernel.kallsyms] [k] finish_task_switch.isra.0
3.23% openssl libc.so.6 [.] pthread_rwlock_unlock@@GLIBC_2.34
2.42% openssl libcrypto.so.3 [.] BN_usub
1.61% openssl libcrypto.so.3 [.] BN_mod_exp_mont_consttime
1.61% openssl libcrypto.so.3 [.] OPENSSL_LH_strhash
1.61% openssl libcrypto.so.3 [.] 0x00000000002ae8fb
1.61% openssl libcrypto.so.3 [.] 0x00000000002ae92b
1.61% openssl libcrypto.so.3 [.] 0x00000000002ae965
1.61% openssl libcrypto.so.3 [.] 0x00000000002aea3b
1.61% openssl libcrypto.so.3 [.] 0x00000000002b4e94
0.81% openssl [kernel.kallsyms] [k] __alloc_pages
0.81% openssl [kernel.kallsyms] [k] __fget_light
0.81% openssl [kernel.kallsyms] [k] copy_page
0.81% openssl [kernel.kallsyms] [k] n_tty_write
0.81% openssl [kernel.kallsyms] [k] syscall_enter_from_user_mode
0.81% openssl ld-linux-x86-64.so.2 [.] dl_relocate_object
0.81% openssl ld-linux-x86-64.so.2 [.] do_lookup_x
0.81% openssl libc.so.6 [.] __strcmp_avx2
0.81% openssl libc.so.6 [.] pthread_rwlock_wrlock@@GLIBC_2.34
0.81% openssl libcrypto.so.3 [.] BN_add
0.81% openssl libcrypto.so.3 [.] BN_gcd
0.81% openssl libcrypto.so.3 [.] BN_get_flags
0.81% openssl libcrypto.so.3 [.] BN_is_zero
0.81% openssl libcrypto.so.3 [.] BN_mod_word
0.81% openssl libcrypto.so.3 [.] BN_rshift1
0.81% openssl libcrypto.so.3 [.] BN_uadd
0.81% openssl libcrypto.so.3 [.] CRYPTO_THREAD_lock_free
0.81% openssl libcrypto.so.3 [.] CRYPTO_malloc
0.81% openssl libcrypto.so.3 [.] DSO_free
0.81% openssl libcrypto.so.3 [.] EVP_DigestSignFinal
0.81% openssl libcrypto.so.3 [.] OPENSSL_strcasecmp

```

Fonte: Autor, 2024

Para geração de curvas elípticas, além da quantidade de eventos, a utilização de bibliotecas, como se vê na figura 19, é muito menor em comparação ao RSA na figura 18:

Figura 19: Perf do Elliptic Curve P-256

```

root@client-VM:/home/mint# perf report -i ECDSA_record | cat
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 20 of event 'cpu-clock:pppH'
# Event count (approx.): 5000000
#
# Overhead Command Shared Object Symbol
# .....
#
15.00% openssl libc.so.6 [.] pthread_rwlock_unlock@GLIBC_2.34
5.00% openssl [kernel.kallsyms] [k] copy_page
5.00% openssl [kernel.kallsyms] [k] rcu_all_qs
5.00% openssl [kernel.kallsyms] [k] vbg_req_perform
5.00% openssl [kernel.kallsyms] [k] zap_pte_range
5.00% openssl ld-linux-x86-64.so.2 [.] check_match
5.00% openssl libc.so.6 [.] __GI___qsort_r
5.00% openssl libc.so.6 [.] cfree@GLIBC_2.2.5
5.00% openssl libc.so.6 [.] pthread_rwlock_rdlock@GLIBC_2.2.5
5.00% openssl libcrypto.so.3 [.] OPENSSSL_LH_retrieve
5.00% openssl libcrypto.so.3 [.] OPENSSSL_LH_strhash
5.00% openssl libcrypto.so.3 [.] 0x000000000000b28b4
5.00% openssl libcrypto.so.3 [.] 0x0000000000016f235
5.00% openssl libcrypto.so.3 [.] 0x000000000001ac5b9
5.00% openssl libcrypto.so.3 [.] 0x000000000001ac5c6
5.00% openssl libcrypto.so.3 [.] 0x000000000001ac61b
5.00% openssl libcrypto.so.3 [.] 0x000000000001c238e
5.00% openssl libcrypto.so.3 [.] 0x000000000001c9f23
#
# (Cannot load tips.txt file, please install perf!)
#
root@client-VM:/home/mint#

```

Fonte: Autor, 2024

Para a geração de certificados digitais usando o algoritmo pós-quântico Dilithium, a figura 20 mostra a utilização de diversas funções durante a execução.

Figura 20: Perf do Dilithium3

```

root@client-VM:/home/mint# perf report -i dilithium_record | cat
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 27 of event 'cpu-clock:pppH'
# Event count (approx.): 6750000
#
# Overhead Command Shared Object Symbol
# .....
#
# 7.41% openssl libc.so.6 [.] __strcmp_avx2
# 7.41% openssl libc.so.6 [.] __int_free
# 7.41% openssl libc.so.6 [.] pthread_rwlock_rdlock@GLIBC_2.2.5
# 7.41% openssl libc.so.6 [.] pthread_rwlock_unlock@GLIBC_2.34
# 3.70% openssl [kernel.kallsyms] [k] charge_memcg
# 3.70% openssl [kernel.kallsyms] [k] finish_task_switch.isra.0
# 3.70% openssl [kernel.kallsyms] [k] kmem_cache_alloc
# 3.70% openssl [kernel.kallsyms] [k] mnt_want_write
# 3.70% openssl [kernel.kallsyms] [k] next_uptodate_page
# 3.70% openssl [kernel.kallsyms] [k] workingset_age_nonresident
# 3.70% openssl [kernel.kallsyms] [k] xas_find
# 3.70% openssl ld-linux-x86-64.so.2 [.] __dl_lookup_symbol_x
# 3.70% openssl ld-linux-x86-64.so.2 [.] __dl_relocate_object
# 3.70% openssl ld-linux-x86-64.so.2 [.] do_lookup_x
# 3.70% openssl libc.so.6 [.] memmove_avx_unaligned
# 3.70% openssl libcrypto.so.3 [.] BUF_MEM_free
# 3.70% openssl libcrypto.so.3 [.] EVP_CIPHER_get_nid
# 3.70% openssl libcrypto.so.3 [.] OPENSSL_LH_doall_arg
# 3.70% openssl libcrypto.so.3 [.] 0x00000000001ac5eb
# 3.70% openssl libcrypto.so.3 [.] 0x00000000001c23a1
# 3.70% openssl libcrypto.so.3 [.] 0x00000000001c23b3
# 3.70% openssl libcrypto.so.3 [.] 0x00000000001e7618
# 3.70% openssl libcrypto.so.3 [.] 0x00000000001e765f
#
# (Cannot load tips.txt file, please install perf!)
#
root@client-VM:/home/mint#

```

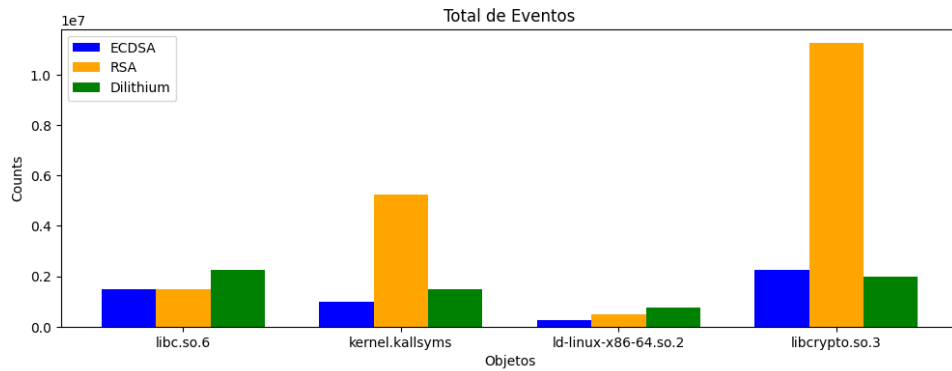
Fonte: Autor, 2024

A análise dos registros indica que, embora o Dilithium utilize um pouco mais de funções do que o ECDSA, ele ainda demonstra uma otimização superior em relação ao RSA.

As figuras 21 e 22 apresentam os resultados comparando-se os algoritmos e frequência dos objetos utilizados das funções. Verifica-se que, para utilização dos objetos *kernel.kallsyms* e *libcrypto.so.3*, o algoritmo RSA necessita chamar mais funções, enquanto os outros dois têm valores semelhantes.



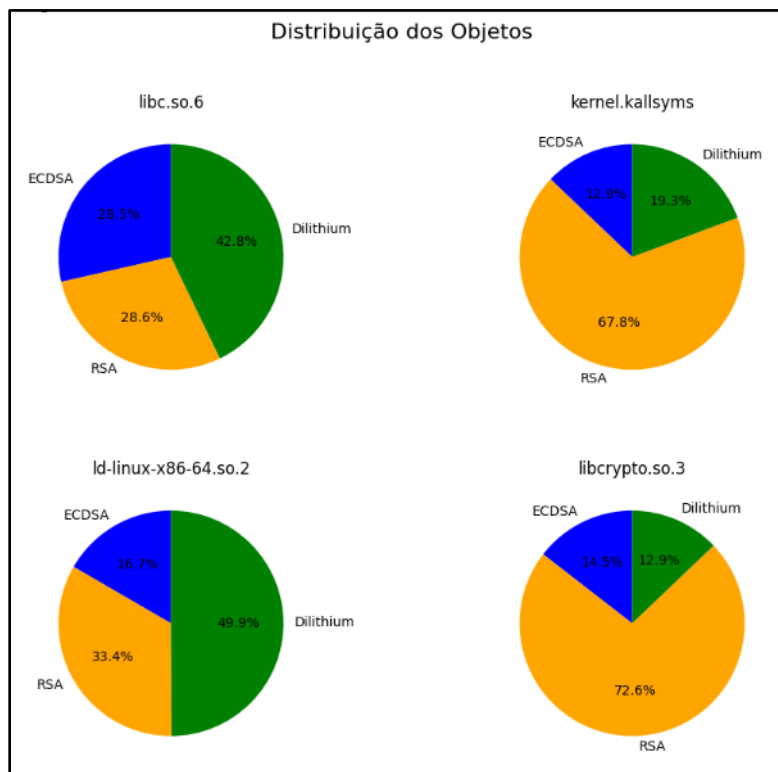
Figura 21: Gráfico do total de eventos por objeto



Fonte: Autor, 2024

Na figura 22, foi utilizado um cálculo de distribuição ponderado, onde a porcentagem de CPU de cada *share object* utilizada está relacionada com a quantidade de eventos registrados para cada algoritmo.

Figura 22: Gráfico distribuição dos objetos



Fonte: Autor, 2024

O comando *perf record* revelou informações importantes sobre a eficiência dos algoritmos de criptografia RSA, ECDSA e Dilithium durante a geração de certificados digitais. As métricas analisadas como *overhead*, *simples*, *event counts* e *shared objects*, mostraram como as alocações de recursos e execução das funções são distribuídas para cada algoritmo.

Os resultados indicam que, enquanto o RSA demanda um maior número de chamadas de funções e utiliza extensivamente a biblioteca *libcrypto.so.3*, os algoritmos ECDSA e Dilithium demonstram maior eficiência. Isso servirá de base para serem verificados os testes de velocidade dos algoritmos com o *openssl speed*, onde será mostrado o tempo para verificação e assinaturas de chaves.

### 3.2.3 Velocidade no OpenSSL

Nas figuras 20, 21 e 22 foram analisados os testes de velocidade de cada algoritmo pelo comando *OpenSSL-speed*. Para o algoritmo RSA, o comando mostra, além de assinaturas e verificação, o tempo de encriptação e decríptação. O algoritmo Dilithium apresenta, adicionalmente, a quantidade de chaves geradas por segundo. Por último, o algoritmo de curvas elípticas traz resultados somente para verificação e assinatura de certificados. Assim, o foco está nos resultados em comum aos três algoritmos: tempo de verificação e assinatura de chaves.

O resultado do teste do algoritmo RSA, na figura 23, mostra, de forma completa, a quantidade de assinaturas, verificações, decríptações e encriptações de dados por segundo.

Figura 23: Velocidade RSA

```

CPUINFO: OPENSSL_ia32cap=0xdefa220b478bffff:0x842421
      sign  verify  encrypt  decrypt  sign/s  verify/s  encr./s  decr./s
rsa  512 bits 0.000045s 0.000004s 0.000004s 0.000068s 22440.0 267129.7 255711.5 14784.7
rsa 1024 bits 0.000158s 0.000010s 0.000011s 0.000243s 6317.5 99460.1 87291.5 4123.5
rsa 2048 bits 0.000928s 0.000034s 0.000054s 0.000984s 1077.3 29189.3 18503.7 1016.3
rsa 3072 bits 0.003585s 0.000087s 0.000099s 0.005296s 278.9 11529.0 10121.3 188.8
rsa 4096 bits 0.008477s 0.000127s 0.000128s 0.008070s 118.0 7860.2 7787.9 123.9
rsa 7680 bits 0.066327s 0.000675s 0.000607s 0.073111s 15.1 1480.4 1647.1 13.7
rsa 15360 bits 0.430000s 0.002234s 0.001766s 0.557778s 2.3 447.7 566.3 1.8
mint@client-VM:~$

```

Fonte: Autor, 2024

Na figura 24, pode ser visto o resultado somente para assinaturas e verificações de certificados, uma vez que o algoritmo de curva elíptica é utilizado com essa finalidade.

Figura 24: Velocidade Elliptic Curve P-256

```
CPUINFO: OPENSLL_ia32cap=0xdefa220b478bffff:0x842421
                sign    verify    sign/s verify/s
256 bits ecdsa (nistp256) 0.0000s 0.0001s 27533.9 7668.7
mint@client-VM:~$
```

Fonte: Autor, 2024

Para o algoritmo Dilithium de criptografia pós-quântica, além de validações e assinaturas, o teste mostra quantas chaves foram geradas por segundo.

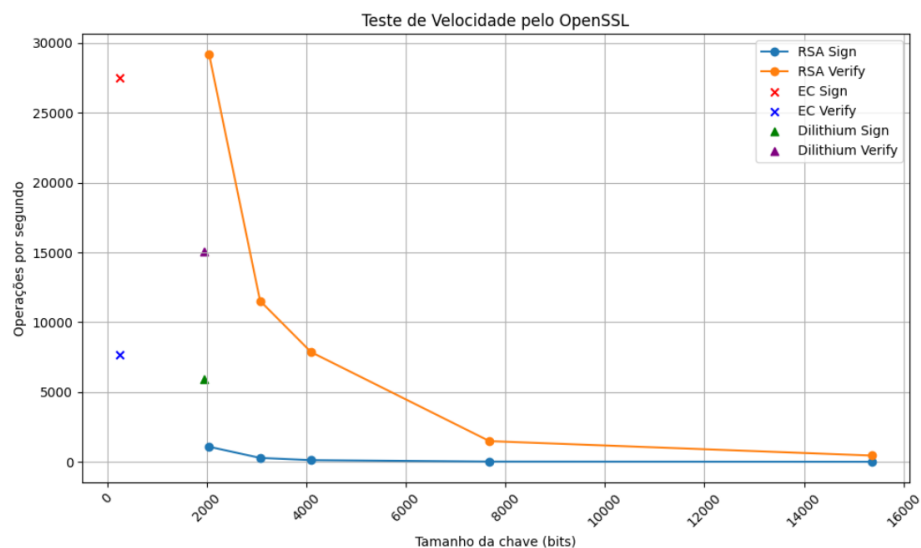
Figura 25: Velocidade Dilithium3

```
CPUINFO: OPENSLL_ia32cap=0xdefa220b478bffff:0x842421
                keygen    signs    verify keygens/s    sign/s    verify/s
dilithium3 0.000083s 0.000168s 0.000066s 11989.7    5947.1    15093.2
mint@client-VM:~$
```

Fonte: Autor, 2024

Por fim, é apresentado o gráfico dos testes de velocidade na figura 26:

Figura 26: Gráficos do teste de velocidade



Fonte: Autor, 2024



O gráfico da figura 26 mostra as operações de assinatura e verificação de certificados, comuns aos três algoritmos, em relação ao tamanho da chave utilizada.

Para o algoritmo EC, foi utilizado tamanho de chave de 256 *bits*, pois esse tamanho é suficientemente seguro para comunicação. O algoritmo Dilithium utiliza por padrão a chave de 1952 *bytes* que é considerado pelos criadores do algoritmo e validado pelo NIST como seguro nível AES128 (*Advanced Encryption Standard 128 bits*). O algoritmo RSA (da função do OpenSSL) faz o cálculo com chaves de tamanhos variados acima de 512, porém, a segurança só é igualmente validada a partir de 2048 *bits*.

#### 4 RESULTADOS

Os resultados mostram que o algoritmo RSA utiliza muito mais recursos para geração de certificados digitais em relação ao ECC e Dilithium. Na figura 14, o uso da CPU tanto no ambiente do usuário como no sistema de forma geral, têm valores discrepantes em alto nível em relação aos algoritmos Dilithium e ECDSA. Isso mostra que, para o uso de tarefas pela CPU rodando o algoritmo RSA, leva mais tempo no ambiente do usuário ou sistema operacional de forma geral. Além disso, o RSA realizou 1100 trocas de contexto para utilizar outras chamadas de funções durante sua execução, trazendo *delay* na finalização de cada uma delas. Enquanto isso, nos algoritmos ECC e Dilithium houve 18 e 20 trocas, respectivamente. Isso mostra que o processamento de informações leva muito mais tempo e recurso para ser realizado no RSA.

Já entre o Dilithium3 e a curva elíptica P-256, o consumo de CPU e trocas de contexto são praticamente iguais, apesar do primeiro utilizar menos recurso da CPU. Em relação às falhas de página, o Dilithium foi o algoritmo que mais obteve, em torno de 856. Logo, foi necessário acessar a memória virtual mais vezes para prosseguir com execução das tarefas. De qualquer forma, não foi discrepante em relação aos outros dois, RSA e ECC, que tiveram 826 e 828, nesta ordem.

Foi analisado, com mais detalhes de forma majoritária, o fato de o RSA utilizar mais recursos para geração de certificados em relação do P-256 e Dilithium3, como se vê na figura 22, mostrando a distribuição dos objetos utilizados durante geração de certificados pelos



algoritmos, onde a maior parte das áreas ocupadas nos gráficos é em relação as funções do RSA. Novamente, o RSA é o que possui maior *overhead*, ou seja, consumo excessivo da CPU para execução das funções nos objetos *kernel.kallsyms* e *libcrypto.so.3*. Em relação à melhor distribuição de recursos para execução das suas tarefas em todos os objetos, o algoritmo de curva elíptica apresentou o melhor resultado.

## 5 CONSIDERAÇÕES FINAIS DO DESEMPENHO

A análise detalhada da distribuição de recursos durante a geração de certificado, reforça a superioridade da curva elíptica em termos de eficiência de execução. Por isso, considerando tanto o consumo de recursos quanto o desempenho, a curva elíptica P-256 emerge como a escolha mais eficaz para aplicações que exigem geração de certificados digitais.

Além disso, é importante reconhecer o algoritmo Dilithium3 por sua eficiência em proteção contra os ataques de computadores quânticos. Enquanto a curva elíptica P-256 oferece eficiência em termos de consumo de recursos e desempenho, o Dilithium3 se destaca por sua resistência aos avanços potenciais da computação quântica, além da sua eficiência ser relativamente próxima ao método de curva elíptica. Com sua forte composição de assinatura pós-quântica, ele oferece uma camada adicional de segurança para aplicações que exigem proteção contra ameaças futuras de computação quântica.

Portanto, ao se considerar a escolha de algoritmos para geração de certificados digitais, é essencial ponderar não apenas o desempenho e o consumo de recursos, mas também a segurança contra possíveis avanços tecnológicos.

## 6 IMPLEMENTAÇÃO DO DILITHIUM 3

Será descrita a implementação prática de uma conexão segura entre um cliente e um servidor, criado com máquinas virtuais utilizando o algoritmo de criptografia Dilithium3. O objetivo é demonstrar a eficiência e viabilidade do Dilithium3 na geração e uso de certificados digitais para estabelecer uma comunicação segura.



## 6.1 Servidor *Web* com Dilithium3

As configurações de rede das máquinas foram realizadas na rede 192.168.0.0/30, como pode ser visto nas figuras 27 e 28.

Figura 27: Configuração de rede server

```

mint@server-VM:~$ ifconfig enp0s8:1
enp0s8:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.2 netmask 255.255.255.252 broadcast 192.168.0.3
    ether 08:00:27:56:49:88 txqueuelen 1000 (Ethernet)

mint@server-VM:~$ ping 192.168.0.1 -c 5
PING 192.168.0.1 (192.168.0.1): 56 data bytes
64 bytes from 192.168.0.1: icmp_seq=0 ttl=64 time=0,735 ms
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=1,164 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=64 time=0,616 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=64 time=0,848 ms
64 bytes from 192.168.0.1: icmp_seq=4 ttl=64 time=0,523 ms
--- 192.168.0.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0,523/0,777/1,164/0,222 ms
mint@server-VM:~$ █

```

Fonte: Autor

O servidor obtém o IP 192.168.0.1/30 e tem comunicação com o cliente via ICMP.

Figura 28: Configuração de rede client

```

mint@client-VM:~$ ifconfig enp0s8:1
enp0s8:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.0.1 netmask 255.255.255.252 broadcast 192.168.0.3
    ether 08:00:27:a4:72:68 txqueuelen 1000 (Ethernet)

mint@client-VM:~$ ping 192.168.0.2 -c 5
PING 192.168.0.2 (192.168.0.2): 56 data bytes
64 bytes from 192.168.0.2: icmp_seq=0 ttl=64 time=0,676 ms
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=1,071 ms
64 bytes from 192.168.0.2: icmp_seq=2 ttl=64 time=1,347 ms
64 bytes from 192.168.0.2: icmp_seq=3 ttl=64 time=0,754 ms
64 bytes from 192.168.0.2: icmp_seq=4 ttl=64 time=0,867 ms
--- 192.168.0.2 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0,676/0,943/1,347/0,242 ms
mint@client-VM:~$ █

```

Fonte: Autor

O cliente está com o IP 192.168.0.2/30 e tem comunicação até o servidor. Foi utilizado o certificado digital Dilithium3, gerado conforme vê-se na figura 13, para estabelecer a comunicação.

Figura 29: Abrindo conexão com servidor *web*

```
mint@server-VM:~$ quantum/build/bin/openssl s_server -cert dilithium3_CA.crt -
key dilithium3_CA.key -www -tls1_3 -accept 4443
Using default temp DH parameters
ACCEPT
```

Fonte: Autor

Foi utilizado o OpenSSL para abrir comunicação do servidor *web*, na porta 4443, para diferenciar a porta padrão 443 do HTTPS, usando a versão 1.3 do TLS. Com o servidor à espera (ou em estado de *listen*) na porta 4443, qualquer tentativa de acesso nesse *host*/porta retornará um *ACCEPT* para fechar a comunicação conforme a figura 30.

Figura 30: Conexão com servidor *web*

```
mint@client-VM:~$ openssl s_client -connect 192.168.0.2:4443
Connecting to 192.168.0.2
CONNECTED(00000003)
Can't use SSL_get_servername
depth=0 CN=dilithium CA
verify error:num=18:self-signed certificate
verify return:1
depth=0 CN=dilithium CA
verify return:1
---
Certificate chain
 0 s:CN=dilithium CA
  i:CN=dilithium CA
  a:PKEY: UNDEF, 192 (bit); sigalg: dilithium3
  v:NotBefore: Mar 20 00:32:04 2024 GMT; NotAfter: Mar 20 00:32:04 2025 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIVfjCCCimgAwIBAgIUQit2pxtm0PjIdU3XjDX3yRQRK/YwDQYLKwYBBAECggsH
BgUwFzEVBMBGA1UEAwMZGlsaxRoaXVtIENBMB4XDTI0MDMyMDAwMzIwNFoXDTI1
MDMyMDAwMzIwNFowFzEVBMBGA1UEAwMZGlsaxRoaXVtIENBMBIiHtDANBgsrBgEE
AQKCCwCBQ0CB6EAI049smWsy70wVLD0mxb20TynuKJH9dTd6+ST00jm3KvBRzYf
GvGo7rNJS9R9kbFk8rBaiu30JFdMLDGLt3jy72qt7HgH0o/v8vN9A6vW8BCKvRE3
BLAKyFY6G2ncb8hHkyUrmh19BEzFZE1gPeRuiUk2PlNCC30z6aYdGrBs0659BErR
Rt5ZMyMx7JQKf2Va6Q0nWs49EE2n+SGjom7kCW3pbziTTPhgYP+4mRnvQRyQLHkq
2Qb6UUzqXL3Lc6117K2X9/RtwgoighCBs2LGPbapZnmA+keynjs0qwtMIBvxd+tm
KyiyfmXVCSLguWgGfjXW0g8bWM2nSZQPGZ6e9rLujdNtN1smhn2HCFiS4liEQL5w
8YubJvWP9XidUCtngSWegwYxxIIUsvrdoL07PSZmuBLsa/EBdqE51tJdXHH1IngD
qt6IcyzXJA3Fr3cgdlma8B2gILtkmJomNEgghbRONGPRXwWn7RYTBxrRskLUuZvc
dhfhwQ39dyn0sN7atuFvoqZha73vgNohA/b+LDFof+8MwRcDFWbfBxcccE4Xer9y
LeAJCf+6AJ0jTa9KMftLP2AnLZT0AC1eMCQwZo8RndMKLuCL8g5bBtRbGMBDGs0Y
xz5yJjG7A5IY2ISuUKRZBR7dcE+8KbZPbse7/Nq3qFWQSBzxdXfpvlt5aVH4sCgf
nLhXotgwLITL+Zeis+DcNtUKJHrhffWpfqNKZ2Zcef+U5fv+z5Mu+Qf+G3FR82sP
4IZeXv2Rsf9I0uKsRvi6RA+upgfbwLTXP536rWW2MPEfvWAv0BAnXsW66ECgpwzX
TVnRwLdD2xS83S/0+vC5tRf4s7f5M0XX08Khhk17a9a+BsZvTEBrTPRDEhgY/k0N
```

Fonte: Autor

A conexão foi feita com sucesso no IP do servidor 192.168.0.2: *CONNECTED (00000003)*. Não é possível identificar o nome do servidor, pois não foi atribuído. Verifica-se que é utilizado o certificado do Dilithium e é identificado seu *Common Name CN=dilithium CA*, definido quando foi criado o certificado, conforme a figura 13. Houve um erro, informando que o certificado é autoassinado, pois não foi utilizada uma autoridade de certificação confiável (CA).

Na figura 31, após o final do certificado, é detalhado sobre tipo de assinatura, o algoritmo utilizado para realizar troca de chaves, erro de verificação por ser um certificado autoassinado e informações do TLS.

Figura 31: Negociando criptografia AES256SHA384 da conexão com servidor *web*

```
8nIRJS1s9g3E57ldmHZysF5Yu/5gxjbeGHMbtpp16UWauTQr6J17C7yslf4L+j
9Wuo/yTqluMQNiRelPz3n8ApUack9mfSd9fC0GuvxVwW2niIwuv/l7jmep9zfSWX
4TxULhKqzxX9hnyJaascMuBjhF0XThQM7JKsRr3P/GeP0Um8Aymhtbpp00yrXs0t
9c6ip2tv4kHEqt72sC5XGvy5A6hgZWyKkKGws cnT4nJ+naHZ7QAL0qCt3vX5dXiA
jKy+yvYDSGxvgQLZ2gAAAAAAAAAAAAAAAAAAAAoQGCA\KA==
-----END CERTIFICATE-----
subject=CN=dilithium CA
issuer=CN=dilithium CA
---
No client certificate CA names sent
Peer signature type: dilithium3
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 9099 bytes and written 449 bytes
Verification error: self-signed certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 192 bit
This TLS version forbids renegotiation.
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 18 (self-signed certificate)
---
```

Fonte: Autor

Após validar a certificação, são iniciados os *handshakes* entre os pares, onde é feita validação da criptografia, *AES256SHA384*, utilizando troca de chaves pela curva elíptica, *X25519*, que é usada por padrão no OpenSSL.

A figura 32 ilustra os detalhes da sessão estabelecida, destacando as especificações do protocolo utilizado, as cifras, o identificador da sessão (ID), o tempo de vida e o momento de início da sessão, assim como as linhas das mensagens recebidas. Após a configuração inicial e a troca de informações necessárias para estabelecer a conexão segura, a sessão está pronta para iniciar a troca de conteúdo via protocolo HTTP.

Figura 32: Sessão da conexão com servidor *web*

```

read R BLOCK
---
Post-Handshake New Session Ticket arrived:
SSL-Session:
  Protocol      : TLSv1.3
  Cipher       : TLS_AES_256_GCM_SHA384
  Session-ID:  EA81A194C5916020A1CE80A38F293ACE32998645DB57906B84F17A5A0E618866
  Session-ID-ctx:
  Resumption PSK: 51CF009EBFC0FA3154F28726B802BCEF00ED0F90833761C9617E0EBE2A17A931B4AC038384
A6B87D9D16266B94CE7535
  PSK identity: None
  PSK identity hint: None
  SRP username: None
  TLS session ticket lifetime hint: 7200 (seconds)
  TLS session ticket:
0000 - 16 b4 0c c7 b1 c7 d2 1d-6f 81 71 87 24 f0 e3 3f  .....o.q.$..?
0010 - ec 60 cf f7 4f dd db c5-a2 a1 84 1e c6 cb eb b5  ..0.....
0020 - 7b ed fd 4f 25 63 5a a4-01 e0 63 08 58 bb 71 90  {...0%cZ...c.X.q.
0030 - 6d 8f c4 76 f7 5a 14 d3-07 1b 2a a8 1e d1 4b 78  m..v.Z.....*..Kx
0040 - 2a 4e 2a 3f d3 93 92 f4-cb 1c fa 94 36 b0 7d 84  *N*?.....6.}.
0050 - e2 f1 25 21 1f 9c 9e 01-2e b6 5a b0 74 ec 3a 07  ..%!.....Z.t.:.
0060 - 11 b1 8d 88 40 e5 4f b7-f6 ae 2c ae c1 79 ca a9  ...@.0.....y..
0070 - ab e7 69 a3 b5 fb f9 a1-49 96 ca 84 7f 2a 22 e4  ..i.....I.....*".
0080 - 4e da 3a 12 58 7e 47 cd-f2 91 c1 a9 3c d3 9e b7  N...X-G.....<...
0090 - 0c 1d db fc d0 7c 5e a9-b5 56 03 15 89 91 13 59  ....|^..V.....Y
00a0 - 32 2c 61 5f dd 74 54 c1-a3 cf 35 fe 78 df 39 55  2,a..t...5..x.9U
00b0 - 1a 9d b8 63 dd 58 c6 14-f6 00 a9 61 e5 1f 9c 88  ...c.X.....a....
00c0 - 81 e6 52 c5 d8 28 73 b2-e3 6d e5 bb 79 23 55 3e  ..R..(s..m..y#U>

  Start Time: 1715482863
  Timeout    : 7200 (sec)
  Verify return code: 18 (self-signed certificate)
  Extended master secret: no
  Max Early Data: 0
---
read R BLOCK
GET /
HTTP/1.0 200 ok
Content-type: text/html

```

Fonte: Autor

Estabelecida a nova sessão entre o servidor e cliente confirmando os parâmetros de segurança que irão utilizar na comunicação. Após isso, foi realizado *GET /* onde o servidor respondeu que foi bem-sucedida.

Os testes mostraram que o servidor *web*, configurado na porta 4443, usando o OpenSSL, pôde estabelecer uma conexão segura com o cliente, usando o certificado pós-quântico Dilithium3.

A sessão estabelecida foi capaz de negociar a criptografia *AES256SHA384* e utilizar a curva elíptica *X25519* para a troca de chaves, confirmando a capacidade do Dilithium3 de integrar-se com padrões de criptografia modernos.

## 6.2 IPsec com certificado Dilithium3

Neste tópico, será realizada a comunicação entre essas duas máquinas através do IPsec utilizando a aplicação *Strongswan*. Será criado o ambiente do *Strongswan* dentro do diretório raiz construído no início da montagem do ambiente conforme segue:

```
wget https://download.Strongswan.org/Strongswan-6.0.0beta6.tar.bz2
tar xf Strongswan-6.0.0beta6.tar.bz2 && cd Strongswan-6.0.0beta6

LIBS=-loqs \
    CFLAGS=-I$BUILD_DIR/include/ \
    LDFLAGS=-L$BUILD_DIR/lib/ \
    ./configure --prefix=$BUILD_DIR --enable-oqs --disable-ikev1
make -j$(nproc) && sudo make install
```

Onde *\$BUILD\_DIR* = */home/mint/quantum/build*.

O parâmetro *LIBS* indica as bibliotecas do Open Quantum Safe (*oqs*) que serão usadas (*-l<library>*), *LDFLAGS* direciona o caminho de bibliotecas que não são padrões do sistema (*-L<lib dir>*), *CFLAGS* indica as flags do compilador C para o *oqs* e o *--prefix* instala os arquivos no *build* raiz do nosso ambiente.

No *Strongswan*, os principais programas para seu funcionamento são o *charon* e *swanctl*. O *charon* é uma *daemon* que realiza todas as funcionalidades do IKE em segundo plano, enquanto o *swanctl* controla suas configurações e funcionalidade como uso de certificados, chaves, parâmetros de conexão etc.

Foi realizado o *link* do comando *charon* para o diretório */home/mint*:

```
ln -s quantum/build/libexec/ipsec/charon ~/charon
```

A figura 33 mostra a iniciação do *charon* com as configurações utilizadas e os tokens em execução que foi instalado anteriormente.

Figura 33: Execução do Charon

```
mint@client:~$ sudo ./charon
00[DMN] Starting IKE charon daemon (strongSwan 6.0.0beta6, Linux 5.15.0-76-generic, x86_64)
00[LIB] providers loaded by OpenSSL: legacy default
00[CFG] install DNS servers in '/home/mint/quantum/build/etc/resolv.conf'
00[LIB] loaded plugins: charon random nonce x509 revocation constraints pubkey pkcs1 pkcs7 pgp
dnskey sshkey pem openssl pkcs8 xcbc cmac kdf oqs drbg attr kernel-netlink resolve socket-defau
lt vici updown
00[JOB] spawning 16 worker threads
```

Fonte: Autor

Após a execução, os *plugins* carregados pelo programa são listados na Figura 33 e é visto o *oqs* com êxito. Assim, será realizada a criação do certificado autoassinado do servidor conforme comando abaixo pelo OpenSSL:

```
openssl req -x509 -new -newkey dilithium3 -keyout server.key -out server.crt -nodes -subj
"/CN=oqs_server" -days 10
```

Para o cliente, foi criado sua chave privada *client\_srv.key* usando Dilithium e gerando o certificado *client\_srv.csr*, onde será necessário assinar pelo certificado do servidor *server.crt*:

```
openssl genpkey -algorithm dilithium3 -out client.key
openssl req -new -newkey dilithium3 -keyout client_srv.key -out client_srv.csr -nodes -subj
"/CN=oqs_client"
```



Assinando o certificado do cliente com o certificado CA do servidor (autoassinado):

```
openssl x509 -req -in client_srv.csr -out client_srv.crt -CA server.crt -CAkey server.key -CA0001 -  
days 10
```

Após mover os certificados *server.crt* e *client.crt* para o diretório *\$BUILD\_DIR/etc/swanctl/x509* em ambas as máquinas, são movidas as chaves privadas *server.key* e *cliente.key* para os diretórios *\$BUILD\_DIR/etc/swanctl/private* nas máquinas *server* e *cliente*, respectivamente, assim como o *server.crt* no diretório da Autoridade Certificadora *\$BUILD\_DIR/etc/swanctl/x509ca*, podendo-se configurar os *peers* para estabelecer a comunicação.

Foi realizada a edição do arquivo de configurações *swanctl.conf* localizado no *\$BUILD\_DIR/etc/swanctl/*.

As configurações demonstradas na Figura 34 indicam a conexão com o cliente no IP 192.168.0.1, indicando que a autenticação será por chave pública (*pubkey*) e mostrando qual certificado o *host* local irá utilizar (*server.crt*). Na configuração do cliente (*remote*), indica-se que será por chave pública também a autenticação, e o certificado do CA que seria do próprio servidor. Na seção *children* adiciona-se somente o parâmetro *start\_action*, que especifica quando será fechado o túnel. Nesse caso, a opção *trap* informa o *charon* para iniciar somente quando houver tráfego na rede entre os dois *hosts*.

Figura 34: Configuração swanctl

```
connections {
  host-host {
    remote_addrs = 192.168.0.1

    local {
      auth=pubkey
      certs = server.crt
    }
    remote {
      auth = pubkey
      cacerts = server.crt
    }
    children {
      net-net {
        start_action = trap
      }
    }
  }
}

# Include config snippets
#include conf.d/*.conf
```

Fonte: Autor

No lado do cliente, essas configurações são parecidas, o que muda é que o endereço do *host* remoto será 192.168.0.2 (do servidor), o certificado local *cliente.crt*, que foi assinado pelo CA do servidor e o *id* da CA, que será *oqs\_server* do próprio certificado do servidor, como pode ser visto na figura 35.

Figura 35: Configuração do swanctl no cliente

```
connections {
  host-host {
    remote_addrs = 192.168.0.2

    local {
      auth=pubkey
      certs = client.crt
    }
    remote {
      auth = pubkey
      id = "CN=oqs_server"
    }
    children {
      net-net {
        start_action = trap
      }
    }
  }
}

# Include config snippets
#include conf.d/*.conf
```

Fonte: Autor





Com o charon ainda ativo em segundo plano, deve-se abrir uma nova janela e executar o comando `swanctl -q` para carregar todas as configurações e os certificados digitais do servidor e cliente, conforme pode ser visto na figura 36.

Figura 36: Erro ao carregar chave Dilithium

```

mint@client:~/quantum/build/sbin$ sudo ./swanctl -q
loaded certificate from '/home/mint/quantum/build/etc/swanctl/x509/client.crt'
loaded certificate from '/home/mint/quantum/build/etc/swanctl/x509ca/server.pem'
OQS RNG could not be switched to openssl
building CRED_PRIVATE_KEY - ANY failed, tried 6 builders
loading '/home/mint/quantum/build/etc/swanctl/private/client_srv.key' failed: parsing ANY private key failed
no authorities found, 0 unloaded
no pools found, 0 unloaded
loaded connection 'host-host'
successfully loaded 1 connections, 0 unloaded
mint@client:~/quantum/build/sbin$

```

Fonte: Autor

Ocorreu um erro ao carregar a chave privada do Dilithium `client.key`, indicando que o *Strongswan* não conseguiu alternar para o gerador de números aleatórios (*RNG*) do OpenSSL ao usar o *plugin OQS*. Para essa correção ser realizada, é preciso reconstruir o `liboqs` que foi feito anteriormente com a opção de integrar junto ao OpenSSL, já que foi instalado o nosso OpenSSL nas máquinas. Com isso, o `liboqs` poderá usar as bibliotecas em conjunto com OpenSSL já construído com o `oqsprovider`.

```

cd $WORKSPACE/liboqs/build \
    cmake -DOQS_USE_OPENSSL=ON .. \
    make -j$(nproc) && make install

```

Os outros parâmetros não foram declarados, pois ele já os obtém do seu próprio *cache*. Assim, após a reinstalação do `liboqs` com esse parâmetro, foi carregada a chave Dilithium, utilizando o *plugin* do OpenSSL no *Strongswan* com sucesso.

A figura 37 mostra que foram carregados com sucesso os certificados e a chave pública criados pelo algoritmo Dilithium3.



Figura 37: Chave Dilithium carregada com sucesso

```

mint@client:~/quantum/build/sbin$ sudo ./swanctl -q
loaded certificate from '/home/mint/quantum/build/etc/swanctl/x509/client.crt'
loaded certificate from '/home/mint/quantum/build/etc/swanctl/x509ca/server.pem'
loaded Dilithium3 key from '/home/mint/quantum/build/etc/swanctl/private/client_srv.key'
no authorities found, 0 unloaded
no pools found, 0 unloaded
loaded connection 'host-host'
successfully loaded 1 connections, 0 unloaded
mint@client:~/quantum/build/sbin$

```

Fonte: Autor

Finalmente, será iniciada a comunicação ponto-a-ponto IPsec solicitado pelo cliente ao servidor:

```

mint@client:~/quantum/build/sbin$ sudo ./swanctl --initiate --child net-net
[...]
[IKE] received end entity cert "CN=oqs_server"
[CFG] using trusted certificate "CN=oqs_server"
[IKE] authentication of 'CN=oqs_server' with DILITHIUM_3 successful
[IKE] peer supports MOBIKE
[IKE] IKE_SA host-host[1] established between
192.168.0.1[CN=oqs_client]...192.168.0.2[CN=oqs_server]
[IKE] scheduling rekeying in 13035s
[IKE] maximum IKE_SA lifetime 14475s
[CFG] selected proposal: ESP:AES_GCM_16_128/NO_EXT_SEQ
[IKE] CHILD_SA net-net{2} established with SPIs c5b19ddc_i ce6d5687_o and TS 192.168.0.1/32
=== 192.168.0.2/32
initiate completed successfully

```

O *output* final da comunicação VPN *IKE\_SA* mostra o cliente recebendo o certificado do servidor e que foi autenticado pelo *DILITHIUM\_3*. Além disso, os pares têm suporte ao *MOBIKE* que permite a VPN continuar estável mesmo se um dos *hosts* trocar de endereço IP.

Após isso, é estabelecido *IKE\_SA host-host*, que foi configurado no *swanctl.conf* entre 192.168.0.1 (*oqs\_client*) e 192.168.0.2 (*oqs\_server*).

Foi definido um tempo para realizar nova troca de chaves em 13035 segundos (mais de 3h) e foi escolhido o algoritmo de criptografia simétrica no protocolo ESP *AES\_GCM\_16\_128/NO\_EXT\_SEQ*, ou seja, é o algoritmo AES com *Galois Counter Mode (GCM)* que permite criptografia e autenticação simultânea. Não há extensão de sequência para realizar o encapsulamento dos pacotes.

Por fim, com o comando *swanctl --list-sas* é possível listar as SAs criadas, ou seja, a VPN estabelecida entre os dois *peers*, conforme figura 38, mostrando o protocolo IKEv2, os endereços IP, criptografia utilizada, tempo de realizar nova criação de chaves e status da seção.

Figura 38: IPsec estabelecido

```

mint@client:~/quantum/build/sbin$
mint@client:~/quantum/build/sbin$ sudo ./swanctl --list-sas
host-host: #1, ESTABLISHED, IKEv2, 4dae9088fa99ae27_i* 2554024907a20f0b_r
local 'CN=oqs_client' @ 192.168.0.1[4500]
remote 'CN=oqs_server' @ 192.168.0.2[4500]
AES_CBC-128/HMAC_SHA2_256_128/PRF_HMAC_SHA2_256/ECP_256
established 224s ago, rekeying in 12811s
net-net: #2, reqid 1, INSTALLED, TUNNEL, ESP:AES_GCM_16-128
installed 224s ago, rekeying in 3019s, expires in 3736s
in c5b19ddc, 0 bytes, 0 packets
out ce6d5687, 0 bytes, 0 packets
local 192.168.0.1/32
remote 192.168.0.2/32
mint@client:~/quantum/build/sbin$ date
Sun May 19 12:00:46 PM -03 2024
mint@client:~/quantum/build/sbin$

```

Fonte: Autor

A implementação do IPsec, utilizando certificados gerados pelo algoritmo Dilithium3, demonstrou a capacidade de integrar criptografia pós-quântica com a aplicação *Strongswan* em estabelecer conexões seguras. A configuração do ambiente *Strongswan*, a geração e assinatura dos certificados, e a configuração detalhada dos parâmetros de conexão foram realizadas com sucesso, comprovando a possibilidade de utilizar algoritmos pós-quânticos em ambientes de rede seguros.



---

A utilização da criptografia pós-quântica, especificamente com o algoritmo Dilithium3, em um servidor *web* e em uma VPN com o IPsec, demonstrou ser viável e eficaz o uso dessas tecnologias em situações reais. Essas implementações são um passo significativo para garantir que a segurança das comunicações digitais permaneça diante dos avanços da computação quântica. Continuar investindo em testes e melhorias nessa área é fundamental para preparar a infraestrutura de segurança para os desafios futuros.



## REFERÊNCIAS

AMORIM, Pedro Rubbioli; HENRIQUES, Marco A. A. **Uma comparação de desempenho de algoritmos para criptografia pós-quântica**. In: WORKSHOP DE TRABALHOS DE INICIAÇÃO CIENTÍFICA E DE GRADUAÇÃO – SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS (SBSEG), 20., 2020, Evento Online. Porto Alegre: Sociedade Brasileira de Computação, 2020. P. 256-269

BAI, Shi. et. al. **CRYSTALS-Dilithium: algorithm specifications and supporting documentation**. 2023. Disponível em: <<https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf>>. Acesso em 24 abr. 2024.

BARRETO, P. et al. **Introdução à criptografia pós-quântica**. In: Minicursos do XIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, 2013, pp. 46-100. SBC. DOI: 10.5753/sbc.9275.1.2.

BOZARTH, Alex. **Developing with quantum-safe OpenSSL: discover why quantum-safe so important to OpenSSL**. Publicado em 1 nov. 2023. Atualizado em 15 dez. 2023. Disponível em: <<https://developer.ibm.com/tutorials/awb-quantum-safe-openssl>>. Acesso em: 21 abr. 2024.

BRASSARD, Gilles. **Modern Cryptology: A Tutorial**. New York: Springer-Verlag, 1988. ISBN: 3-540-96842-3.

CHENG, P. C. et al. **A security architecture for the internet protocol**. IBM Systems Journal, v. 37, n. 1, p. 42-60, 1998. DOI: 10.1147/sj.371.0042. Disponível em: <<https://ieeexplore.ieee.org/document/5387134>>. Acesso em 19 de Mai. 2024

CREMERS, C. **Key exchange in IPsec revisited: Formal analysis of IKEv1 and IKEv2**. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), v. 6879 LNCS, Springer Verlag, 2011, pp. 315-334. DOI: 10.1007/978-3-642-23822-2\_18

HELLMAN, M. E. **An overview of public key cryptography**. IEEE Communications Magazine, v. 40, n. 5, p. 42-49, 2005. DOI: 10.1109/mcom.2002.1006971. Disponível em: <<https://doi.org/10.1109/mcom.2002.1006971>>. Acesso em: 5 Mai 2024.

KENT, S.; SEO, K. RFC 4301: **Security Architecture for Internet Protocol (IPSec)**. Request for Comments, IETF, 2005. Disponível em: <<https://datatracker.ietf.org/doc/html/rfc4301>>. Acesso em: 19 Mai. 2024.



MICCIANCIO, D.; REGEV, O. **Lattice-based Cryptography**. Artigo, 2008.

NATIONAL SECURITY AGENCY - NSA. **The Case for Elliptic Curve Cryptography**. Archived from the original on January 17, 2009. Disponível em: <[https://web.archive.org/web/20090117023500/http://www.nsa.gov/business/programs/elliptic\\_curve.shtml](https://web.archive.org/web/20090117023500/http://www.nsa.gov/business/programs/elliptic_curve.shtml)>. Acesso em 5 Mai. 2024.

NIST - NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **NIST announces first four quantum-resistant cryptographic algorithms**. Disponível em: <<https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms>>. Acesso em 21 abr. 2024.

NIST - NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. (2021). **Post-Quantum Cryptography**. Disponível em: <<https://csrc.nist.gov/projects/post-quantum-cryptography>>. Acesso em 21 abr. 2024.

OPEN QUANTUM SAFE. Disponível em: <<https://openquantumsafe.org/>>. Acesso em 21 abr. 2024.

OPENSSL. **OpenSSL Overview**. Disponível em: <[https://wiki.openssl.org/index.php/OpenSSL\\_Overview](https://wiki.openssl.org/index.php/OpenSSL_Overview)>. Acesso em 05 Mai. 2024.

PQ CRYSTALS. Disponível em: <<https://pq-crystals.org/index.shtml>>. Acesso em 21 abr. 2024.

PRESKILL, J. **Quantum computing in the NISQ era and beyond**. Quantum, v. 2, 2018. DOI: 10.22331/q-2018-08-06-79.

SAFDAR U.. **Post Quantum Secure Strongswan with Liboqs**. Nov 24, 2023. Disponível em: <<https://medium.com/@umairsafdar768/post-quantum-secure-Strongswan-with-liboqs-9659141ffb9>>. Acesso 17 Mai. 2024

SHOR, P. W. **Algorithms for quantum computation: Discrete logarithms and factoring**. In: Proceedings – Annual IEEE Symposium on Foundations of Computer Science, FOCS, 1994, pp. 124-134. IEEE Computer Society. DOI: 10.1109/SFCS.1994.365700.



---

STINSON, Douglas R.; PATERSON, Maura B. **Cryptography: Theory and Practice**, Third Edition (Discrete Mathematics and Its Applications). 2005. Chapman and Hall/CRC.

VAN HEESCH, Maran; et. al. **Towards Quantum-Safe VPNs and Internet**. IACR Cryptology ePrint Archive, 2019. Disponível em: <<https://eprint.iacr.org/2019/1277.pdf>>. Acesso em: 21 abr. 2024.