UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Matheus de Oliveira Saldanha

PROTOCOLO ACME PÓS-QUÂNTICO

Florianópolis, Santa Catarina – Brasil
2022

Matheus de Oliveira Saldanha

PROTOCOLO ACME PÓS-QUÂNTICO

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador(a): Ricardo Felipe Custódio, Dr.
Coorientador(a): Alexandre Augusto Giron, Me.

Florianópolis, Santa Catarina – Brasil
2022

Notas legais:

Não há garantia para qualquer parte do software documentado. Os autores tomaram cuidado na preparação desta tese, mas não fazem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem qualquer responsabilidade por erros ou omissões. Não se assume qualquer responsabilidade por danos incidentais ou consequentes em conexão ou decorrentes do uso das informações ou programas aqui contidos.

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 02:47h do dia 24 de março de 2022.

Matheus de Oliveira Saldanha
Protocolo ACME Pós-quântico / Matheus de Oliveira Saldanha; Orientador(a), Ricardo Felipe Custódio, Dr.; Coorientador(a), Alexandre Augusto Giron, Me. – Florianópolis, Santa Catarina – Brasil, 30 de março de 2022.
192 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico, Programa de Graduação em Ciências da Computação.

Inclui referências

1. Pós-quântico, 2. ACME, 3. Protocolo, I. Ricardo Felipe Custódio, Dr. II. Alexandre Augusto Giron, Me. III. Programa de Graduação em Ciências da Computação IV. Protocolo ACME Pós-quântico

CDU 02:141:005.7

Matheus de Oliveira Saldanha

PROTOCOLO ACME PÓS-QUÂNTICO

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciências da Computação, e foi aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 30 de março de 2022.

Renato Cislaghi
Coordenador(a) do Programa de
Graduação em Ciências da Computação

Banca Examinadora:

Ricardo Felipe Custódio, Dr.
Orientador(a)
Universidade Federal de Santa Catarina

Alexandre Augusto Giron, Me.
Coorientador(a)
Universidade Federal de Santa Catarina

Thaís Bardini Idalino, Dr.
Universidade Federal de Santa Catarina

Gustavo Zambonin, Me.
Universidade de Ottawa

Este trabalho é dedicado aos meus pais.

AGRADECIMENTOS

Eu gostaria de agradecer ao meu orientador, Dr. Ricardo Felipe Custódio pelos incentivos e sugestões sobre o trabalho. Ao meu coorientador, Me. Alexandre Augusto Giron pelos conselhos e direcionamentos, sem os quais esse trabalho não seria possível. Também aos meus colegas e amigos, pelas discussões durante todo o curso, onde muito conhecimento foi construído. E, por fim, aos meus pais por sempre me incentivarem e investirem na minha educação.

“Não é o conhecimento, mas o ato de aprender, não é a posse, mas o ato de chegar lá, que nos concede a maior satisfação.”

Carl Friedrich Gauss

RESUMO

Cada vez mais a internet rege a troca de informações entre diversas regiões do mundo, sendo o principal meio de comunicação contemporâneo. Para isso, ela é regulada por diversos protocolos que permitem informações de serem enviadas, recebidas e interpretadas. Um dos principais protocolos de rede utilizados para a troca de informações é o HTTP, muito utilizado na web para acessar diversos domínios. Contudo, quanto mais informações são trocadas mais importante se torna a segurança das mesmas, tendo em vista que muitas delas são confidenciais. A extensão do protocolo HTTP para o protocolo HTTPS surge nesse ambiente, permitindo agora a confidencialidade das informações, assim como a confiança para quem está sendo enviada por meio de toda uma infraestrutura de chaves públicas com certificados digitais como a principal entidade. Entretanto, o trabalho manual e lento de requisição e instalação de certificados digitais em domínios abre espaço para erros humanos. Dessa forma, surge o protocolo ACME o qual trata de automatizar o ciclo de vida de certificados, desde a emissão até a instalação em um domínio. Outra ameaça à segurança da informação é o desenvolvimento de computadores quânticos. Computadores quânticos funcionam de maneira diferente dos clássicos e podem tirar proveito de certos algoritmos para tornar a criptografia utilizada atualmente muito vulnerável. Dessa forma, surge o desenvolvimento da criptografia pós-quântica com algoritmos os quais funcionam em computadores clássicos, porém os quais os computadores quânticos não conseguem tirar proveito, sendo, dessa maneira, seguros à ataques de computadores quânticos. Este trabalho inicia com uma introdução aos principais conceitos de criptografia, infraestrutura de chaves públicas e redes de computadores até uma explicação detalhada do funcionamento do protocolo ACME. E então, é realizada uma implementação ao protocolo ACME com o uso de algoritmos pós-quânticos, criando uma extensão do mesmo. Nota-se que o protocolo já é genérico o suficiente para a integração de novos algoritmos, sendo apenas necessário mudanças em implementações em código. O trabalho é finalizado com a realização de testes de desempenho para diversas fases do protocolo ACME, comparando os diversos algoritmos clássicos e pós-quânticos entre si. É observado melhor desemepenho do algoritmo Dilithium – um dos algoritmos pós-quânticos – na maioria dos testes e um desempenho inferior na maioria para o algoritmo Rainbow – outro candidato pós-quântico.

Palavras-chaves: Pós-quântico. ACME. Protocolo.

ABSTRACT

The internet increasingly governs the exchange of information between different regions of the world, being the main contemporary mean of communication. For that, it is regulated by several protocols that allow information to be sent, received and interpreted. One of the main network protocols used to exchange information is HTTP, which is widely used on the web to access different domains. However, the more information that is exchanged, the more important its security becomes, given that many of it is confidential. The extension of the HTTP protocol to the HTTPS protocol emerges in this environment, now allowing for confidentiality of information, as well as trust for whoever is being sent through an entire infrastructure of public keys with digital certificates as the main entity. However, the manual and slow process of requesting and installing digital certificates in domains makes room for human errors. Thus, the ACME protocol appears, which deals with automating the life cycle of certificates, from issuance to its installation in a domain. Another threat to information security is the development of quantum computers. Quantum computers work differently from classic ones and can take advantage of certain algorithms to make currently used cryptography very vulnerable. Thus, there is the development of post-quantum cryptography with algorithms which work on classical computers, but which quantum computers cannot take advantage of, thus being safe from quantum computer attacks. This work begins with an introduction to the main concepts of cryptography, public key infrastructure and computer networks, to a detailed explanation of the functioning of the ACME protocol. And then, an implementation of the ACME protocol is performed using post-quantum algorithms, creating an extension of it. Note that the protocol is already generic enough for the integration of new algorithms, requiring only changes in code implementations. The work is finalized with performance tests for different phases of the ACME protocol, comparing the different classical and post-quantum algorithms with each other. Better performance is observed for the Dilithium algorithm – one of the post-quantum algorithms – in most tests and poorer performance in most for the Rainbow algorithm – another post-quantum candidate.

Keywords: Post quantum. ACME. Protocol.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

HTTPS	Protocolo de Transferência de Hipertexto Seguro
HTTP	Protocolo de Transferência de Hipertexto
DNS	Sistema de Nomes de Domínio
TLS	Segurança da Camada de Transporte
ACME	Ambiente Automático de Gerenciamento de Certificados
RFC	Pedido para Comentários
IoT	*Internet* das Coisas
NIST	Instituto Nacional de Padrões e Tecnologia
DES	*Data Encryption Standard*
3DES	*Data Encryption Standard 3*
AES	Padrão de Criptografia Avançada
RSA	Rivest-Shamir-Adleman
ECDSA	Algoritmo de Assinatura Digital de Curvas Elípticas
SHA	Algoritmo Seguro de Hash
AC	Autoridade Certificadora
ICP	Infraestrutura de Chaves Públicas
ISRG	Grupo de Pesquisa de Segurança da Internet
IETF	Internet Engineering TaskForce
JSON	Notação de Objetos JavaScript
CSR	Solicitação de Assinatura de Certificado
ES256	ECDSA com SHA256
SHA256	Algoritmo Seguro de Hash 256
EdDSA	Algoritmo de Assinatura Digital de Curvas-Edwards
Ed25519	EdDSA com Curva 25519

SUMÁRIO

1 INTRODUÇÃO

Sabe-se que hoje em dia grande parte da informação é mantida na internet e distribuída na mesma, principalmente, via websites (COFFMAN; ODLYZKO, 2002). Dessa forma, questões de integridade, sigilo e autenticidade da informação tornam-se cada vez mais relevantes no mundo moderno. Nesse aspecto, é importante garantir que essas propriedades de segurança da informação sejam garantidas.

O protocolo HTTPS – HTTP (NAYLOR *et al.*, 2014) com TLS (CHAN *et al.*, 2018) – inicialmente foi feito na década de 1990 para transações bancárias e outros processos onde informações confidenciais estão em risco. Porém, hoje em dia, o protocolo é o mais utilizado para navegar na web, onde é feita grande parte da troca de informação da internet. Para isso, o HTTPS nos permite verificar a integridade dos dados, ou seja, permite que a informação não seja modificada no processo de transmissão. Também oferece confidencialidade, mantendo a informação em sigilo onde apenas as partes autorizadas têm acesso. E garante a autenticação do servidor, ou seja, o servidor é quem diz ser e, também, permite – de maneira opcional – que o cliente se autentique. Entretanto, essa última questão exige certa complexidade para ser implementada, requerendo toda uma infraestrutura.

Para garantir a integridade dos dados, o protocolo HTTPS utiliza mecanismos de resumos criptográficos. Os resumos criptográficos, adicionados com outros processos, asseguram que a informação foi mantida imutável durante a troca.

Assim como resumos criptográficos sustentam a integridade de dados no HTTPS, a cifragem garante a confidencialidade da informação. Isso é feito transformando a informação clara em um informação cifrada, sendo inviável a leitura para indivíduos sem autorização (que não possuem a chave criptográfica).

Já a autenticação da informação, como mencionado anteriormente, requer diferentes abstrações para que seja assegurada. Dentre elas, temos os certificados digitais – os quais terão grande foco nesse trabalho – que podem ser vistos como identidades digitais, de uma maneira simplificada. Os certificados digitais se baseiam em diversas primitivas criptográficas, como criptografia assimétrica e assinaturas digitais, por exemplo. Além disso, a partir deles criam-se outras entidades como as Autoridades Certificadoras e Infraestruturas de Chaves Públicas, essenciais para a autenticação na internet. Esses e outros tópicos serão vistos a fundo no decorrer do trabalho.

Nesse âmbito, surge o protocolo ACME (BARNES *et al.*, 2019). ACME ou Ambiente Automático de Gerenciamento de Certificados (*Automated Certificate Management Environment*, em inglês) automatiza toda essa complexidade associada a um certificado digital, englobando processos de emissão, renovação e revogação dos certificados. Além de facilitar essas tarefas, o protocolo é de uso gratuito, mantido por uma comunidade e não apenas centralizado em uma organização.

O protocolo, desde sua primeira versão publicada em 2016 e sua segunda sendo publicada em 2018 já rendeu alguns resultados. Isso pode ser notado pelo fato de que de 2016 para 2018 o percentual de sites que utilizam HTTPS passou de 40% para 80% (MOORE; HALDERMAN, 2021) (LET'S. . . , 2021). Isso se deve, principalmente, pelos esforços da Let's Encrypt, principal entidade responsável pelo desenvolvimento do protocolo e sua larga adesão. Sendo um dos principais objetivos da Let's Encrypt a difusão do uso do HTTPS na web.

Contudo, novos riscos à segurança digital estão surgindo. O desenvolvimento de computadores quânticos é uma ameaça aos esquemas tradicionais de criptografia. Apesar de, publicamente, estar numa fase muito primitiva, o surgimento de computadores quânticos de larga escala oferece riscos à toda uma infraestrutura de segurança já bem estabelecida.

Computadores quânticos trabalham de forma diferente do computadores clássicos, não utilizando bits e sim qubits. Eles exploram propriedades da mecânica quântica, como superposição e interferência e, assim, conseguem computar certos algoritmos que computadores clássicos são incapazes de resolver eficientemente. Alguns desses algoritmos, como o algoritmo de Shor (SHOR, 1997), são passíveis de ameaça a criptografia clássica, a qual supõe a dificuldade de computar certos problemas matemáticos.

Dessa forma, a proteção da infraestrutura atual requer a modificação de protocolos e de algoritmos para dar suporte à criptografia pós-quântica (BERNSTEIN; LANGE, 2017), a qual utiliza algoritmos em computadores clássicos de forma a serem seguros à ataques de computadores quânticos.

Para isso, é importante destacar que os sistemas que utilizam de certificação digital para autenticação precisarão ser atualizados, gerando novamente a necessidade de intervenção. Assim, este trabalho tem objetivo de avaliar e modificar o protocolo ACME para a atualização destes sistemas para uso de criptografia pós-quantica. Espera-se que o ACME possa automatizar a atualização de sistemas para um ambiente pós-quântico facilitando a atualização para diferentes cenários e com menor esforço de intervenção humana, tais como: IoT ou Internet das Coisas (*Internet of Things*), Servidores HTTPS distribuídos, Redes de sensores, Indústria 4.0 e outros.

1.1 OBJETIVOS

Nesse trabalho foca-se em avaliar e modificar o protocolo ACME para dar suporte a criptografia pós-quântica.

1.1.1 Objetivos Específicos

- Analisar a extensibilidade do protocolo ACME para primitivas criptográficas distintas

- Modificar o protocolo ACME para dar suporte aos algoritmos finalistas e para o candidato SPHINCS+ (BERNSTEIN; HÜLSING *et al.*, 2019) da terceira etapa de padronização do NIST ou Instituto Nacional de Padrões e Tecnologia (*National Institute of Standards and Technology*) (MOODY *et al.*, 2020)
- Realizar testes de performance em relação as modificações nas implementações: testes de tempo de emissão de certificado, testes de latência, testes de perda de pacotes, etc
- Comparar o protocolo modificado em relação a cada algoritmo integrado considerando os resultados dos testes
- Comparar os resultados dos testes em relação ao protocolo modificado com a versão original

1.2 METODOLOGIA

Começa-se o trabalho com o estudo do protocolo ACME, investigando possíveis modificações necessárias para torná-lo pós-quântico. Para isso, estuda-se a RFC8555 (BARNES *et al.*, 2019) do protocolo ACME verificando a possibilidade de integração de algoritmos pós-quânticos e possíveis mudanças que essa integração podem desencadear no protocolo. Então, é escolhido implementações do protocolo ACME as quais serão realizadas as modificações necessárias. Para a escolha, é levado em conta as implementações que oferecem facilidade de realizar testes, uso da implementação pelo público e maior compatibilidade entre a implementação cliente e a servidor. Por fim, produz-se testes de desempenho para avaliar o impacto das novas implementações.

1.3 ESTRUTURA

A Seção 1 introduz a motivação do trabalho, alguns do principais tópicos que serão discutidos em seções seguintes e como será realizado o estudo. Na Seção 2 introduz-se os principais conceitos necessários para que o desenvolvimento do trabalho seja compreendido. Já na Seção 3 é realizada uma introdução aos principais conceitos do protocolo ACME. Na Seção 4 são demonstradas todas as modificações e adições necessárias para realizar a implementação do protocolo ACME pós-quântico. Em seguida, na Seção 5 é realizada desde a configuração do ambiente de testes até seus resultados e análise dos mesmos. Por fim, tem-se a Seção 6 de conlusão de trabalhos futuros onde o trabalho é finalizado.

1.4 LIMITAÇÕES

Durante a elaboração deste trabalho nos limitamos aos algoritmos pós-quânticos finalistas da terceira rodada do NIST e o candidato SPHINCS+, os quais não são padronizados.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CRIPTOGRAFIA SIMÉTRICA

A criptografia simétrica tem como principal função a garantia do sigilo na troca de mensagens. Para manter a confidencialidade das mensagens ela possui dois principais componentes: um **algoritmo** e, também, uma **chave** (STALLINGS, 2013).

O sistema funciona, de maneira simples, como uma função de dois parâmetros:

$$f(k, p) = c$$

Onde k é a chave e p o texto a ser cifrado. Já c é o texto cifrado e f o algoritmo.

Conforme o desenvolvimento da criptografia simétrica, outras questões foram incorporadas para aumentar a segurança do sistema, como modos de operação dos algoritmos.

Como forma de ilustrar algoritmos de criptografia simétrica tem-se o DES (KAUR; SODHI, 2016), 3DES (KARTHIK, 2014) e AES (ABDULLAH, 2017).

A chave é a segurança do sistema criptográfico. É com a ocultação da chave do meio público que o sistema deve se manter seguro, deixando com que o algoritmo seja de domínio público. Isso pode parecer contraditório em algum ponto, pois um pensamento inicial seria de que esconder o algoritmo tornasse o sistema mais seguro. Porém não é o que se tem sido afirmado com o tempo. Esconder o algoritmo faz com que ele seja menos testado pelo público, podendo assim existir falhas e vulnerabilidades que podem ser exploradas. Isso pode ser visto também nos príncipios de Kerckhoffs (PETITCOLAS, 2011) e também nos textos de Claude Shannon (SHANNON, 1949). A chave então é acordada entre as partes que vão efetuar a troca de mensagens e deve ser mantida oculta pelas mesmas.

Uma das principais vantagens desses sistemas é sua performance, sendo aplamente utilizada devido sua velocidade e tamanho de chaves. Porém, o gerenciamento de chaves, por exemplo, torna-se mais complexo à medida que aumenta o número de pessoas que se comunicam. Para cada N usuários, são necessários N usuários tomados dois a dois, ou seja, $\binom{N}{2} = \frac{N^2-N}{2}$ chaves.

2.2 CRIPTOGRAFIA ASSIMÉTRICA

Criptografia assimétrica ou criptografia de chave pública recebe este nome, pois temos um par de chaves composto de uma **chave pública** e uma **chave privada** (STALLINGS, 2013). A chave pública recebe este nome justamente por ser de domínio público, qualquer indivíduo pode conhecê-la sem prejudicar a segurança do sistema. Já a chave privada, assim como na criptografia simétrica, deve ser mantida em segredo, longe do público.

Para cifrar uma mensagem e torná-la sigilosa entre duas partes, a entidade que deseja enviar a mensagem a cifra com a chave pública do destinatário. Dessa forma, apenas o destinatário (que possui a chave privada correspondente) pode decifrar a mensagem. Também existem outros usos da criptografia assimétrica, como a assinatura digital que será visto mais a frente.

Para exemplificar alguns algoritmos de criptografia assimétrica, tem-se o RSA (WARDLAW, 2000) e o ECDSA (JOHNSON; MENEZES; VANSTONE, 2001).

Além de possibilitar outros usos das chaves além de cifrar mensagens, a criptografia de chave pública facilita o gerenciamento das chaves. Pois cada entidade necessita apenas de um par de chaves, ou seja, para N entidades precisa-se de apenas N pares de chave, diferente da criptografia simétrica onde o gerenciamento de chaves torna-se um problema, como foi visto anteriormente. Entretanto, seu desempenho – da criptografia assimétrica – tende a ser inferior quando comparado à criptografia simétrica (MARQAS; ALMUFTI; REBAR, 2020). Porém, seus usos variam e, dessa forma, os dois modelos coexistem para diversas aplicações. Na assinatura digital, por exemplo, a criptografia assimétrica é empregada em conjunto com resumos criptográficos, conceito apresentado a seguir.

2.3 RESUMO CRIPTOGRÁFICO

Resumo criptográfico ou função *hash* criptográfica recebe este nome por representar, de certa forma, um resumo da mensagem a qual função foi aplicada. São funções, também, que são inviáveis de serem invertidas.

Pode-se definir uma função *hash* criptográfica como:

$$f(m) = h$$

onde **f** é a função, **m** a mensagem e **h** o resumo criptográfico (STALLINGS, 2013). E deve possuir as seguintes propriedades:

- **Resistência à pré-imagem**: Dado um valor de resumo criptográfico h, deve ser inviável encontrar qualquer mensagem m tal que $h = f(m)$. Essa propriedade está relacionada ao fato da função ser difícil de ser invertida.
- **Resistência à segunda pré-imagem**: Dada uma mensagem m_1, deve ser difícil encontrar uma mensagem $m_2 \neq m_1$ tal que $h = f(m_1) = f(m_2)$.
- **Resistência à colisão**: Deve ser inviável encontrar quaisquer duas mensagens $m_1 \neq m_2$ tal que seus resumos criptográficos sejam iguais.

Nota-se que a propriedade de **resistência à segunda pré-imagem** é similar à propriedade de **resistência à colisão**. Porém a primeira se diferencia da segunda quando fixa-se um resumo criptográfico h.

Algumas funções de resumo criptográfico são SHA (SAHU; GHOSH, 2017). Outras funções e um histórico podem ser encontrados em (DEBNATH; CHATTOPADHYAY; DUTTA, 2017).

Resumos criptográficos possuem diversas aplicações, como auxiliar na autenticação de uma mensagem, assim como compõe um papel fundamental na assinatura digital (SOBTI; GANESAN, 2012).

2.4 ASSINATURA DIGITAL

A assinatura digital recebe este nome por ser uma analogia muito similar à assinatura de próprio punho (ALIDOOST NIA; SAJEDI; JAMSHIDPEY, 2014). Ela serve para que o signatário prove sua identidade perante outra parte.

No mundo digital existem diversos esquemas de assinatura digital, i.e., formas de assinar e de verificar a validade da assinatura (TILBORG; JAJODIA, 2011). Eles se baseiam na criptografia assimétrica, como visto anteriormente. De maneira simples, esses esquemas se baseiam no conceito onde o signatário utiliza sua própria chave privada para assinar a mensagem – utilizando algum esquema de assinatura digital. E, então, qualquer entidade pode validar a assinatura utilizando a chave pública do signatário e verificando sua validade com base no esquema utilizado.

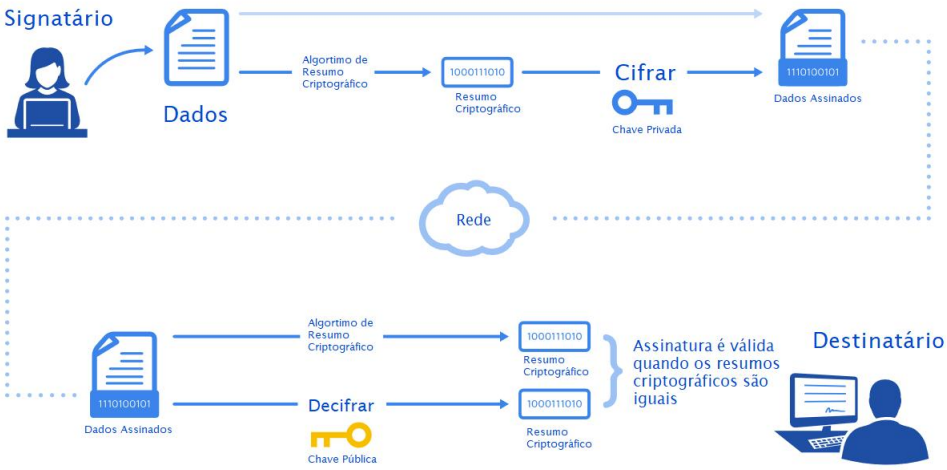

Figura 1 – Assinatura Digital

Fonte: Original

Resumos criptográficos também são utilizados nesse contexto em muitos dos esquemas de assinatura digital. Em muitos casos a mensagem em si não é assinada, mas seu resumo criptográfico é. Isso pode trazer algumas vantagens, como velocidade de assinar e de verificar a assinatura, pois, de maneira geral, o resumo criptográfico tende a ser menor que a mensagem original – desde que as propriedades do resumo criptográfico sejam garantidas como foi mencionado anteriormente.

Podemos generalizar o uso de resumos criptográficos em assinatura digital de maneira simplificada em alguns passos. Primeiro o signatário realiza o resumo criptográfico da mensagem a ser assinada e então utiliza sua chave privada para assinar – entende-se, nessa analogia, assinar por cifrar – esse resumo. O signatário envia tanto a mensagem original quanto o resumo assinado. Dessa forma, o destinatário utiliza a chave pública do signatário para "decifrar"a assinatura do resumo criptográfico e, após calcular o resumo da mensagem que recebeu, compara os dois resumos. Quando os dois são iguais então a assinatura está válida, caso contrário não. A Figura 1 ilustra bem esse cenário.

Entretando um problema que surge é o fato de que não garante-se que a assinatura é de quem diz ser, apenas que a chave pública e chave privada pertencem ao mesmo par. Para isso, surge o conceito de certificado digital, onde podemos relacionar o par de chaves à uma identidade.

2.5 CERTIFICADO DIGITAL E AUTORIDADE CERITIFICADORA

Certificados digitais são documentos que relacionam uma chave pública a uma identidade, ou seja, ao seu proprietário (GAO; LI; TU, 2004).

Além de informações sobre a chave pública, o certificado contém diversas outras informações a respeito do proprietário e de quem emitiu o certificado, usos da chave, como assinatura, cifragem etc. O ceritificado contém uma assinatura digital da entidade que emitiu, também conhecida como **Autoridade Certificadora** ou **AC**. A assinatura serve para garantir que tal AC confia no certificado emitido.

Certificados digitais seguem, principalmente, um padrão de certificado chamado X.509 (COOPER *et al.*, 2008a). Dessa forma, diferentes perfis de certificados – cada um mais apropriado para um certo caso de uso – podem ser feitos, desde que sigam o padrão.

Dessa forma, ceritificados digitais são emitidos por autoridades certificadoras, relacionando a chave pública a uma entidade, seja ela um indivíduo, uma organização ou até uma máquina.

2.6 INFRAESTRUTURA DE CHAVES PÚBLICAS

Nesse contexto, infraestrutura de chaves públicas, ou ICPs, podem ser vistas como cadeias de confiança entre diferentes ACs (ALBARQI *et al.*, 2015).

Pode-se fazer uma analogia a uma árvore de nodos, onde cada nodo representa uma AC. Dessa forma, o primeiro nodo representa a AC raiz, a qual tem seu certificado autoassinado na maioria das implementações. Abaixo dela temos as ACs intermediárias e, no último nível, as ACs folhas, as quais emitem certificados para usuários finais. A tarefa de emissão dos certificados para usuários finais também pode ser respon-

sabilidade de outra entidade, uma **AR** ou **Autoridade Registradora**. A AR tem como objetivo validar documentos dos usuários finais e emitir o certificado para os mesmos. Dessa maneira, a AC num nível acima sempre assina o certificado da AC do nível abaixo, dessa forma, construindo uma cadeia de confiança. Confiando-se na AC raiz, confia-se no sistema como um todo. Pode-se visualizar essa cadeia na Figura 2.

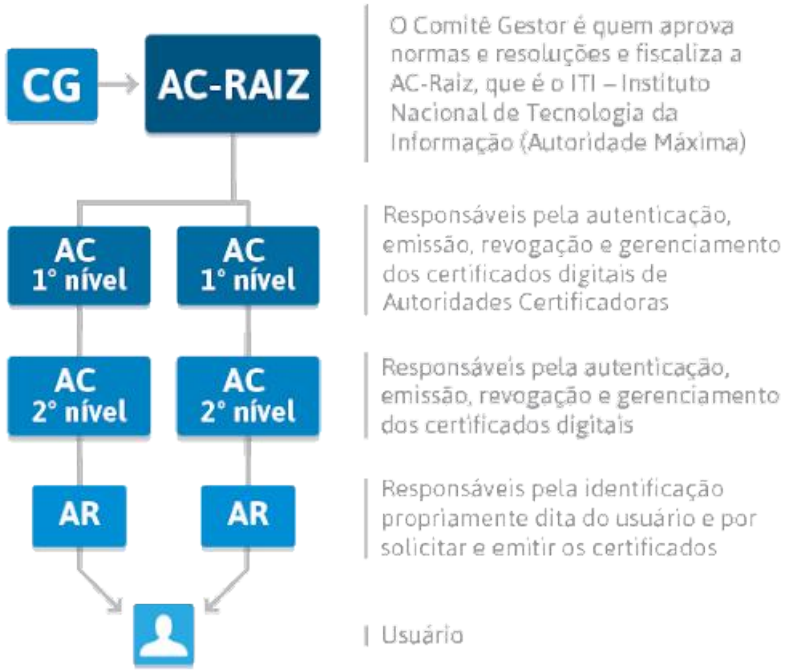

Figura 2 – Infraestrutura de Chaves Públicas

Fonte: Benefícios e Aplicações da Certificação Digital

Dessa maneira, existem ICPs para diversos contextos. Tem-se, por exemplo, a ICP-Brasil e ICP-Edu. A primeira para casos de uso mais relacionados ao governo brasileiro e a segunda para propósitos acadêmicos. Um exemplo típico de utilização de ICPs e Certificados Digitais pode ser visto em protocolos de rede, tais como explicados na seção seguinte.

2.7 PROTOCOLOS DE REDE

Os protocolos de rede são conjuntos estabelecidos de regras que determinam como os dados são transmitidos entre diferentes dispositivos na mesma rede (TRIANTAFYLLOU; SARIGIANNIDIS; LAGKAS, 2018).

Essencialmente, eles permitem que dispositivos conectados se comuniquem entre si, independentemente de quaisquer diferenças em seus processos internos, estrutura ou design. Sendo assim, os protocolos de rede desempenham um papel crítico nas comunicações digitais modernas (DIZDAREVIĆ *et al.*, 2019).

Semelhante à maneira como falar o mesmo idioma simplifica a comunicação entre duas pessoas, os protocolos de rede possibilitam que os dispositivos interajam

uns com os outros por causa de regras predeterminadas incorporadas ao software e hardware dos dispositivos.

Dessa maneira, os protocolos de rede são específico de cada contexto. Podem servir para transferir arquivos (GIEN, 1978), para buscar páginas na web (TURAU, 2003), para se conectar a uma rede (YALAGANDULA, 2000), etc.

Dentre essa variedade, tem-se o protocolo ACME (BARNES *et al.*, 2019). Esse tem como principal contexto o ciclo de vida de certificados digitais. Essencialmente, gerencia diversos processos em que um certificado digital possa passar, como será visto à fundo na próxima seção.

3 PROTOCOLO ACME

3.1 INTRODUÇÃO

Originalmente desenvolvido pelo Grupo de Pesquisa de Segurança da Internet ou Internet Security Research Group (ISRG) para sua própria AC, Let's Encrypt, uma entidade certificadora que emite gratuitamente certificados digitais, o protocolo ACME tem-se tornado um padrão comercial gradativamente. Mais recentemente, em 2019, a organização Força-Tarefa de Engenharia da Internet ou Internet Engineering Task Force (IETF) padronizou o protocolo na RFC8555 (BARNES *et al.*, 2019), favorecendo sua adoção por diversas entidades.

Como mencionado anteriormente, o protocolo tem como principal objetivo automatizar o processo de gerenciamento de certificados. Isso é feito a partir do agente ACME e do servidor ACME, as entidades do protocolo. O cliente do protocolo executa em um website, por exemplo, enquanto o servidor roda em uma AC. Sendo assim, o cliente realiza requisições – como emissão de um certificado – via HTTPS para o servidor por meio de mensagens JSON (BRAY, 2017).

De maneira geral, os principais componentes do procotolo podem ser definidos como:

- **Nonce:** Número aleatório de uso único. Tem como função garantir que o proprietário do par de chaves prove sua posse assinando tal *nonce*.
- **Token:** Código criado pelo servidor e enviado ao cliente, sendo um importante componente do desafio ACME. Pode ter seu uso variado dependendo do desafio ACME, como será visto. Entretanto, pode-se generalizar seu uso como um identificador da instância do desafio ACME juntamente com alguma informação específica da conta ACME, como a impressão digital da chave pública. Dessa forma, pode servir para criar um caminho para o proprietário da conta provar posse de um domínio.
- **Desafio:** Constitui a parte de autenticação do protocolo. Existem diferentes desafios que o protocolo suporta, como o desafio DNS e o desafio HTTP como serão vistos a seguir. Cada desafio tem suas peculiaridades, porém o propósito se mantém único: provar a posse do par de chaves da conta e de um domínio.
- **Cliente ACME:** Entidade do protocolo ACME à qual realiza requisições. Requisições podem ser vistas como pedidos para a emissão de um certificado, renovação, revogação ou até mesmo a autenticação nos passos iniciais do protocolo. Sendo essas requisições respondidas pelo servidor ACME.
- **Servidor ACME:** Entidade do protocolo à qual responde as requisições do cliente ACME. Dessa forma, autenticando o cliente, enviando certificados, renovando

ou revogando os mesmos. O servidor ACME está relacionado com alguma AC, sendo mantida no mesmo servidor (o caso deste trabalho) ou em um servidor separado.

- **Par de chaves da conta:** Cada conta ACME, ou seja, cada relação entre um cliente e um servidor possui um par de chaves assimétricas. As mesmas servem para realizar a autenticação na troca de mensagens entre essas duas entidades por meio de assinaturas digitais.

Mais informações sobre outros componentes e sobre a troca de mensagens podem ser vistos em detalhes em (BARNES *et al.*, 2019). Porém, para tudo ser configurado, primeiramente, temos a etapa de autenticação do protocolo.

3.2 AUTENTICAÇÃO

A etapa de autenticação do protocolo tem como principal objetivo estabelecer um vínculo de confiança entre o cliente e o servidor ACME. Para tal vínculo ser instituído, surge o processo de ACME *challenge* (desafio ACME), o qual garante que os certificados emitidos serão para usuários confiáveis. O desafio ACME se encaixa numa classe de protocolos conhecidos como **desafio-resposta** (KUSHWAHA *et al.*, 2020), nesse caso o desafio faz parte de um protocolo que o engloba – o protocolo ACME. Protocolos desafio-resposta são caracterizados por uma entidade que envia um desafio e uma segunda entidade o recebe e o responde. Um exemplo simples muito utilizado na web é a autenticação por senha, onde o desafio é responder o usuário e senha certos. Nesse caso o servidor web lança o desafio e o cliente que quer se autenticar precisa fornecer as informações corretas. Já na versão 2 do protocolo ACME temos dois desafios possíveis: **desafio HTTP** e **desafio DNS**.

Dessa forma, primeiramente, o cliente ACME realiza um GET (FIELDING; RESCHKE, 2014) para o caminho do diretório (geralmente `/acme/directory`) no servidor ACME, e então o servidor ACME responde com um objeto JSON contendo todos os recursos possíveis do protocolo (Resposta 1 na Figura 7) como pode ser visto a seguir na Figura 3.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "newNonce": "https://example.com/acme/new-nonce",
    "newAccount": "https://example.com/acme/new-account",
    "newOrder": "https://example.com/acme/new-order",
    "newAuthz": "https://example.com/acme/new-authz",
    "revokeCert": "https://example.com/acme/revoke-cert",
    "keyChange": "https://example.com/acme/key-change",
    "meta": {
        "termsOfService":
           "https://example.com/acme/terms/2017-5-30",
        "website": "https://www.example.com/",
        "caaIdentities": ["example.com"],
        "externalAccountRequired": false
        }
}
```

Figura 3 – Resposta ao GET pelo servidor ACME `/acme/directory`

Fonte: RFC 8555

Após isso, o cliente envia uma mensagem HEAD para o caminho do *newNonce* (geralmente `/acme/new-nonce`) para que o servidor envie um *nonce*. O *nonce* é enviado no cabeçalho da resposta HTTP no campo **Replay-Nonce**, servindo de mecanismo de **não repúdio**. O não repúdio visa garantir que o autor da mensagem não tenha capacidade de negar ter criado e assinado a mesma. Nesse caso, o *nonce* assinado na mensagem é aleatório e considera-se que, naquele instante, apenas esse cliente poderia ter gerado esse *nonce* e assinando o mesmo o cliente não tem como negar posteriormente a autoria da mensagem enviada, como pode ser visto a seguir na Figura 4 (Resposta 2 da Figura 7).

```
HTTP/1.1 200 OK
Replay-Nonce: oFvnlFP1wIhRlYS2jTaXbA
Cache-Control: no-store
Link: <https://example.com/acme/directory>;rel="index"
```

Figura 4 – Resposta ao HEAD pelo servidor ACME `/acme/new-nonce`

Fonte: RFC 8555

O *nonce* não é particular dessa resposta do servidor, todas as respostas aos POSTs do cliente a partir desse ponto possuem um *nonce* para que o cliente assine com o par de chaves que irá gerar em seguida. Como o servidor cria esses *nonces* é particular da implementação.

A geração de um par de chaves feita pelo cliente, como explicitado na RFC8555 (BARNES *et al.*, 2019), para os algoritmos de geração e assinatura de chaves criptográficas (JONES, 2015a), é necessário que a AC do servidor ACME dê suporte à curvas elípticas do tipo ES256, i.e., ECDSA (PORNIN, 2013) – um algoritmo de assinatura digital padronizado que pode ser utilizado em conjunto à uma função de hash criptográfico – com a função de resumo criptográfico SHA256 (SAHU; GHOSH, 2017). Além disso, a RFC recomenda dar suporte ao algoritmo EdDSA (JOSEFSSON; LIUSVAARA, 2017), mais especificamente, utilizando a curva Ed25519, porém esse segundo algoritmo não tem a obrigatoriedade de ser suportado. Vale a pena ressaltar que, apesar de ser obrigatório a implementação do algoritmo ES256, não é necessariamente obrigatório seu uso no protocolo, apenas o suporte pela implementação deve existir.

Após a criação do par de chaves, para finalizar a criação da conta ACME do cliente com o servidor, o cliente envia uma mensagem POST para o caminho de *newAccount* do servidor ACME (geralmente `/acme/new-account`), como pode ser visto a seguir na Figura 5.

```
POST /acme/new-account HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "jwk": {...},
        "nonce": "6S8IqOGY7eL2lsGoTZYifg",
        "url": "https://example.com/acme/new-account"
    }),
    "payload": base64url({
        "termsOfServiceAgreed": true,
        "contact": [
        "mailto:cert-admin@exemplo.org",
        "mailto:admin@exemplo.org"
        ]
    }),
    "signature": "RZPOnYoPs1PhjszF...-nh6X1qtOFPB519I"
}
```

Figura 5 – POST do cliente ACME `/acme/new-account`

Fonte: RFC 8555

Dessa mensagem, nota-se o campo *protected*, o qual tem suas informações assinadas e sua assinatura mantida no campo *signature*. Desse campo, tem-se o campo *alg* que identifica o algoritmo utilizado para assinar, o campo *jwk* que contém a chave pública da conta a ser criada, *nonce* que contém o *nonce* a ser assinado, *url*

que contém o caminho para o qual a requisição POST está sendo enviada. Além disso, tem-se outras informações – não muito relevantes para avaliar a criação da conta – no campo *payload*. Nota-se que a partir desse ponto, assim como toda a resposta do servidor contém um *nonce* que deve ser assinado pelo cliente na próxima mensagem POST, todo o POST do cliente é assinado.

Em resposta a essa mensagem o servidor envia uma mensagem que entre outras informações, possui o caminho para a conta criada (Resposta 3 da Figura 7), conforme abaixo na Figura 6.

```
1     HTTP/1.1 201 Created
2     Content-Type: application/json
3     Replay-Nonce: D8s4D2mLs8Vn-goWuPQeKA
4     Link: <https://example.com/acme/directory>;rel="index"
5     Location: https://example.com/acme/acct/evOfKhNU60wg
6
7     {
8         "status": "valid",
9         "contact": [
10        "mailto:cert-admin@exemplo.org",
11        "mailto:admin@exemplo.org"
12        ],
13        "orders":
            "https://example.com/acme/acct/evOfKhNU60wg/orders"
14    }
```

Figura 6 – Resposta ao POST pelo servidor ACME `/acme/new-account`

Fonte: RFC 8555

Essa etapa inicial de autenticação pode ser visualizada de maneira simples na Figura 7.

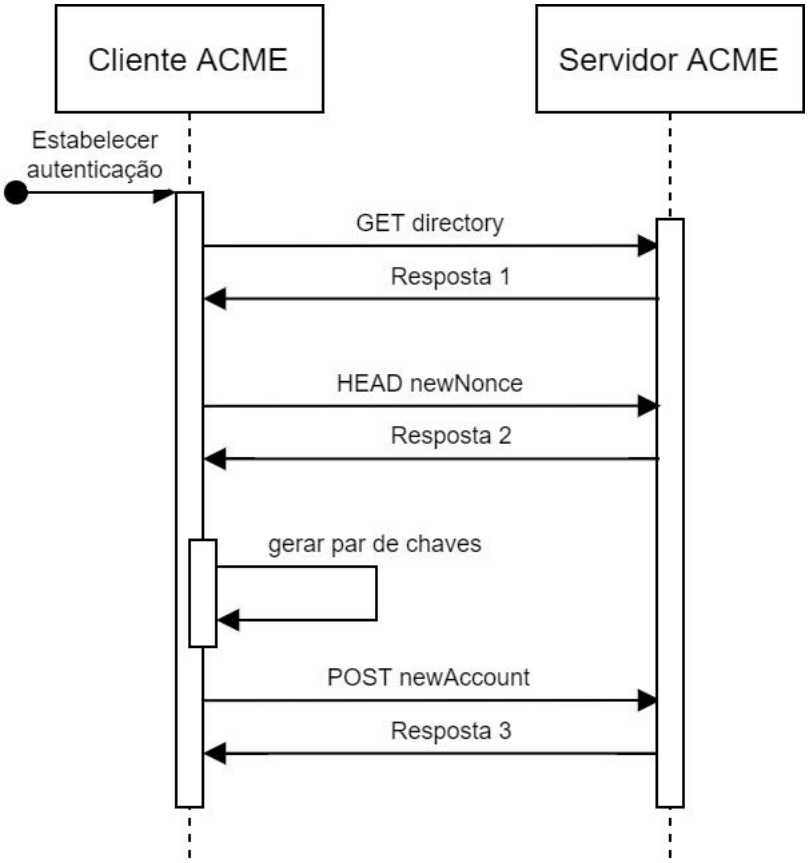

Figura 7 – Autenticação ACME

Fonte: RFC 8555

3.2.1 Desafio HTTP

Como já mencionado, os desafios servem para provar controle sobre certo domínio neste protocolo. Isso acontece quando, principalmente, o cliente ACME quer emitir um certificado para um domínio (newOrder). Dessa forma, quando o cliente manda uma mensagem para o endereço de newOrder o servidor responde com uma mensagem seguindo o seguinte formato da Figura 8:

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "pending",
    "expires": "2016-01-02T14:09:30Z",
    "identifier": {
    "type": "dns",
    "value": "www.exemplo.org"
},
    "challenges": [
        {
        "type": "http-01",
        "url":
           "https://example.com/acme/chall/prV_B7yEyA4",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        },
        {
        "type": "dns-01",
        "url":
           "https://example.com/acme/chall/Rg5dV14Gh1Q",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        }
    ]
}
```

Figura 8 – Resposta ao POST pelo servidor ACME `/acme/new-order`

Fonte: RFC 8555

Onde cada desafio do vetor *challenges* possui qual é o desafio (seu tipo), o seu endereço e o *token* para completar o desafio.

Sendo assim, tem-se como o desafio mais utilizado o HTTP, recebendo este nome devido ao protocolo HTTP (FIELDING; GETTYS *et al.*, 1999).

1. O servidor ACME fornece um *token* para o cliente (como mencionado anteriormente).

2. O cliente coloca um arquivo em seu servidor da web em *http://<dominio_exemplo>/.well-known/acme-challenge/<token>*. Esse arquivo contém o *token*, além de uma impressão digital da chave da sua conta. A impressão digital da chave é um resumo criptográfico da mesma, representando ela de maneira comprimida.

3. Depois o cliente ACME notifica ao servidor que o arquivo está pronto.

4. O servidor ACME tenta recuperar o arquivo. Potencialmente, isso pode levar várias tentativas e cada tentativa pode realizar a recuperação de diferentes formas.

5. Nesse aspecto, se as verificações de validação obtiverem as respostas corretas de seu servidor web, a validação será considerada bem-sucedida e o cliente poderá emitir seu certificado. Por outro lado, se as verificações de validação falharem, o cliente terá que tentar novamente com um novo certificado.

Dos passos acima, o qual vale a pena ressaltar seria como o cliente notifica o servidor que completou o desafio. Isso se da por meio de uma simples mensagem ao servidor com o conteúdo **{}**, conforme abaixo na Figura 9.

```
1     POST /acme/chall/prV_B7yEyA4
2     Host: example.com
3     Content-Type: application/jose+json
4
5     {
6          "protected": base64url({
7          "alg": "ES256",
8          "kid": "https://example.com/acme/acct/evOfKhNU60wg",
9          "nonce": "UQI1PoRi5OuXzxuX7V7wL0",
10         "url": "https://example.com/acme/chall/prV_B7yEyA4"
11         }),
12         "payload": base64url({}),
13         "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
14    }
```

Figura 9 – POST do cliente ACME `/acme/chall`

Fonte: RFC 8555

Portanto, as chaves – antes já autenticadas – são utilizadas nas trocas de mensagens para realizar a autenticação das mesmas por meio das assinaturas digitais. As chaves são utilizadas tanto durante os desafios quanto depois para outras requisições, como a emissão de certificados digitais. Outro ponto importante é o uso do *token* nesse desafio o qual serve para criar um caminho onde o cliente disponibilizará o arquivo contendo a impressão digital da sua chave pública e o *token* em si, provando, assim, que possui controle sob o domínio. Sendo assim, a validação do servidor se resume a verificar se o arquivo contém as informações esperadas e se está no caminho em que deveria estar. Para ser mais exato, as verificações do servidor são:

1. Constrói um URL `http://<dominio_exemplo>/.well-known/acme-challenge/<token>`, onde:

    - o campo de domínio é definido para o nome de domínio que está sendo verificado; e

- o campo do token é definido como o token no desafio

2. Verifica se o URL resultante está bem formado.
3. Cancela a referência do URL usando uma requisição HTTP GET. Este pedido deve ser enviado para a porta TCP 80 no servidor HTTP.
4. Verifica se o corpo da resposta é uma **chave de autorização** bem formada
5. Verifica se a autorização da chave fornecida pelo servidor HTTP corresponde a autorização de chave armazenada pelo servidor.

Esse desafio possui diversas vantagens. É fácil automatizar sem conhecimento extra sobre a configuração de um domínio, além de funcionar com servidores da web já prontos para uso. A Figura 10 forma simplificada – num cenário onde as informações estão corretas e o servidor consegue realizar a verificação sem problemas.

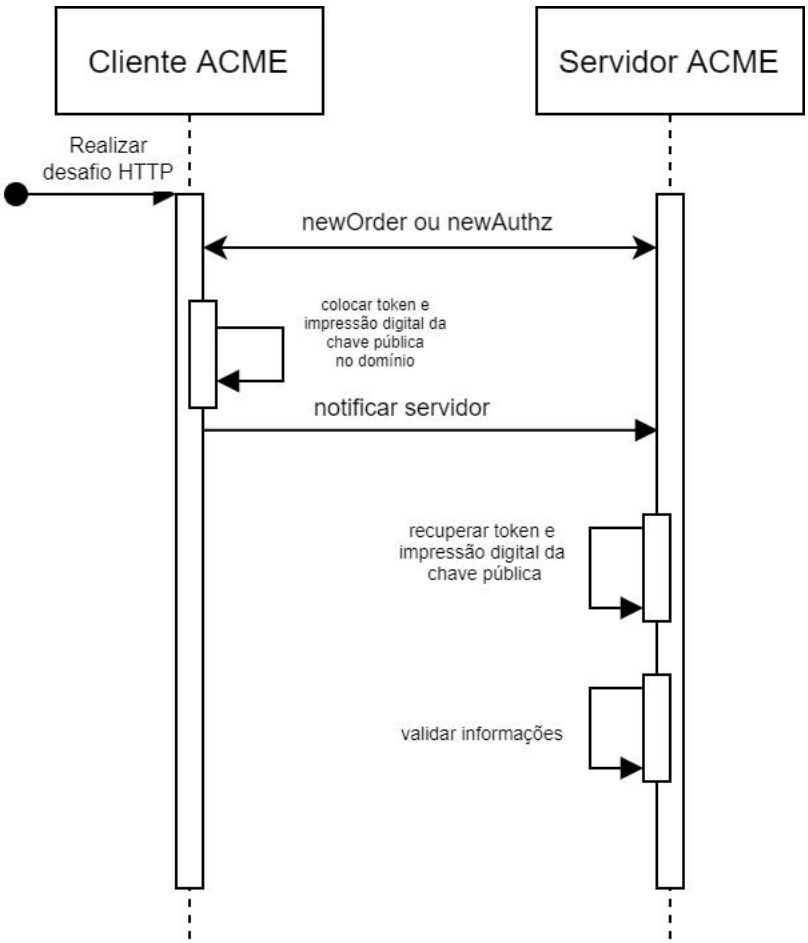

Figura 10 – Desafio HTTP ACME

Fonte: RFC 8555

3.2.2 Desafio DNS

Por outro lado, tem-se o desafio DNS, recebendo este nome devido ao protocolo DNS (MOCKAPETRIS, 1987). Este pede que o cliente prove que possui controle sob o DNS de seu nome de domínio, colocando um valor específico em um registro TXT sob esse nome de domínio. É mais difícil de configurar do que HTTP, entretanto pode funcionar em cenários que o HTTP não pode. O desafio inicia-se da mesma forma que o anterior, com o cliente escolhendo qual desafio quer realizar dentre os enviados no vetor *challenges* pelo servidor. Além disso, assim como no desafio HTTP as mensagens também são assinadas pelo par de chaves já autenticado na etapa anterior. Ou seja, a diferença entre os desafios apenas está na maneira em que o cliente prova o controle sobre tal domínio, mantendo as outras questões iguais.

1. O servidor ACME fornece ao seu cliente um *token*.
2. O cliente criará um registro TXT derivado desse *token* e da impressão digital da chave pública da sua conta, assemelhando-se ao desafio HTTP mencionado.
3. O cliente, então, colocará esse registro em *_acme-challenge.<dominio_exemplo>*.
4. Então, o servidor ACME consultará o sistema DNS para esse registro.
5. Se o servidor encontrar uma correspondência, o cliente pode prosseguir para emitir um certificado.

De maneira mais detalhista, a partir do *token* do desafio DNS – disponibilizado pelo servidor no desafio presente no vetor *challenges* – o cliente concatena o *token* com a impressão digital da chave pública e, por fim, realiza um resumo criptográfico SHA256 da concatenação. Essa concatenação de *token* com impressão digital da chave pública é referenciada na (BARNES *et al.*, 2019) por *Key Authorization*. Por fim, o cliente constrói o domínio de validação concatenando "_acme-challenge."ao domínio a ser validado num registro TXT. Por exemplo, caso o domínio se chamasse *www.exemplo.org* a estrutura do registro seria similar a seguinte:

```
1    _acme-challenge.www.exemplo.org. 300 IN TXT
       "gfj9Xq...Rg85nM"
```

Figura 11 – Registro DNS ACME

Fonte: RFC 8555

Por fim, o cliente envia uma mensagem com o conteúdo **{}**, similar ao HTTP, para notificar o servidor que o mesmo pode verificar a completude do desafio, como na Figura 12.

```
1     POST /acme/chall/Rg5dV14Gh1Q
2     Host: example.com
3     Content-Type: application/jose+json
4
5   {
6       "protected": base64url({
7           "alg": "ES256",
8           "kid":
              "https://example.com/acme/acct/evOfKhNU60wg",
9           "nonce": "SS2sSl1PtspvFZ08kNtzKd",
10          "url": "https://example.com/acme/chall/Rg5dV14Gh1Q"
11      }),
12      "payload": base64url({}),
13      "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
14  }
```

Figura 12 – POST do cliente ACME `/acme/chall`

Fonte: RFC 8555

Sendo assim, o cliente pode ter vários registros TXT para o mesmo nome. No entanto, o cliente deve certificar-se de limpar os registros TXT antigos, pois se o tamanho da resposta ficar muito grande, o servidor ACME começará a rejeitá-los.

A verificação do desafio pelo servidor se resume as seguintes etapas:

1. Calcula o resumo criptográfico SHA-256 da chave da conta ACME
2. Consulta de registros TXT para o nome de domínio de validação, como demonstrado acima
3. Verifica se o conteúdo de um dos registros TXT corresponde ao valor de resumo criptográfico

A Figura 13 forma simplificada – com os detalhes já explicitados acima.

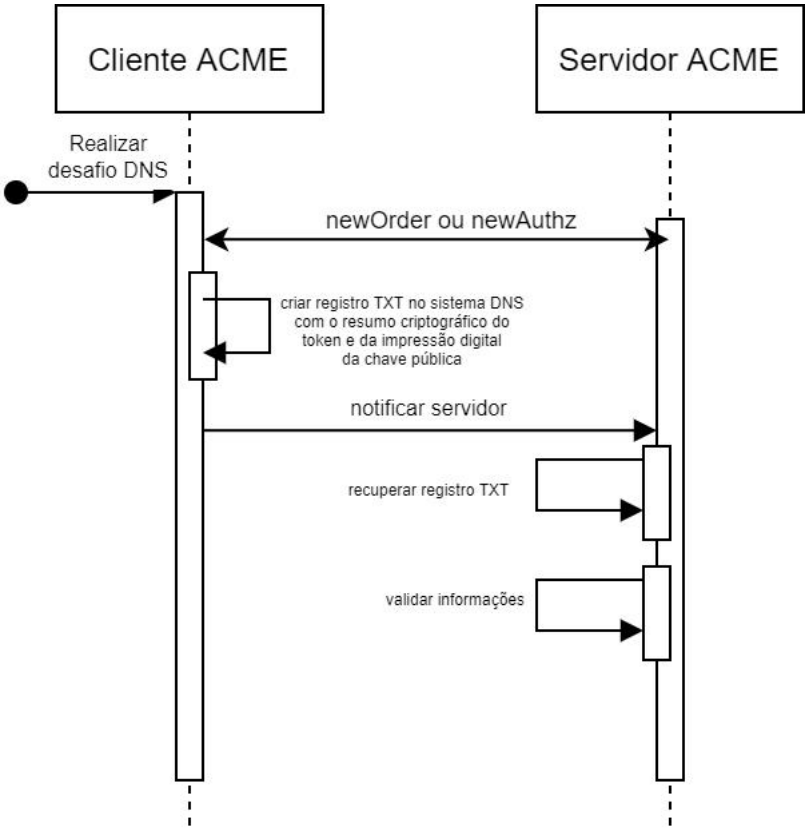

Figura 13 – Desafio DNS ACMEFonte: Original

Fonte: RFC 8555

Uma grande vantagem do uso desse desafio em relação ao HTTP vem do fato que este funciona bem mesmo se o cliente possuir vários servidores web.

3.3 EMISSÃO DE CERTIFICADOS

Foi visto como o cliente ACME realiza autenticação com servidor e como ele pode provar o controle sobre algum dominío – por meio dos desafios. Dessa forma, demonstra-se – de maneira usual – os passos para a geração de um certificado utilizando o protocolo ACME.

Primeiramente, o cliente realiza a etapa de autenticação, estabelecendo um conta ACME com o servidor, como foi explicado anteriormente. Feito isso, o cliente então envia um POST para newOrder, para obter um certificado. Geralmente o caminho para essa requisição se encontra em `/acme/newOrder`, conforme abaixo na Figura 14.

```
POST /acme/new-order HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "kid":
           "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "5XJ1L3lEkMG7tR6pA00clA",
        "url": "https://example.com/acme/new-order"
    }),
    "payload": base64url({
        "identifiers": [
            { "type": "dns", "value": "www.exemplo.org" },
            { "type": "dns", "value": "exemplo.org" }
        ],
        "notBefore": "2016-01-01T00:04:00+04:00",
        "notAfter": "2016-01-08T00:04:00+04:00"
    }),
    "signature": "H6ZXtGjTZyUnPeKn...wEA4TklBdh3e454g"
}
```

Figura 14 – POST do cliente ACME `/acme/new-order`

Fonte: RFC 8555

Com esse POST o cliente requisita um certificado que demonstre o controle sob tais domínios explicitados no vetor *identifiers*. O servidor responde, mostrando, além de outras informações, um vetor de *authorizations*, ou seja, quais autorizações o cliente deve cumprir para demonstrar que possui controle sob tais domínios (um por domínio) e um endereço *finalize*, onde, após cumprir todas as autorizações, o cliente envia a CSR (requisição de certificado assinada), como pode ser visto à seguir na Figura 15.

```
HTTP/1.1 201 Created
Replay-Nonce: MYAuvOpaoIiywTezizk5vw
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/order/TOlocE8rfgo

{
    "status": "pending",
    "expires": "2016-01-05T14:09:07.99Z",
    "notBefore": "2016-01-01T00:00:00Z",
    "notAfter": "2016-01-08T00:00:00Z",
    "identifiers": [
        { "type": "dns", "value": "www.exemplo.org" },
        { "type": "dns", "value": "exemplo.org" }
    ],
    "authorizations": [
        "https://example.com/acme/authz/PAniVnsZcis",
        "https://example.com/acme/authz/r4HqLzrSrpI"
    ],
    "finalize":
      "https://example.com/acme/order/T...o/finalize"
}
```

Figura 15 – Resposta ao POST pelo servidor ACME `/acme/new-order`

Fonte: RFC 8555

Após isso, o cliente ACME envia um POST (referenciado como POST-as-GET na (BARNES *et al.*, 2019), uma maneira de mandar uma mensagem assinada, porém apenas com o intuito de recuperar informação) para o endereço de cada uma das autorizações, conforme abaixo na Figura 16.

```
    POST /acme/authz/PAniVnsZcis HTTP/1.1
    Host: example.com
    Content-Type: application/jose+json

    {
        "protected": base64url({
            "alg": "ES256",
            "kid":
               "https://example.com/acme/acct/evOfKhNU60wg",
            "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
            "url": "https://example.com/acme/authz/PAniVnsZcis"
        }),
        "payload": "",
        "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
    }
```

Figura 16 – POST do cliente ACME `/acme/authz`

Fonte: RFC 8555

Para cada um desses POST-as-GET, o servidor envia, dentre outras informações, um vetor de *challenges* – como demonstrado na seção do desafio HTTP – para que o cliente escolha uma e a realize, conforme abaixo na Figura 17.

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "pending",
    "expires": "2016-01-02T14:09:30Z",
    "identifier": {
    "type": "dns",
    "value": "www.exemplo.org"
},
    "challenges": [
        {
        "type": "http-01",
        "url":
          "https://example.com/acme/chall/prV_B7yEyA4",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        },
        {
        "type": "dns-01",
        "url":
          "https://example.com/acme/chall/Rg5dV14Gh1Q",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        }
    ]
}
```

Figura 17 – Resposta ao POST pelo servidor ACME `/acme/authz`

Fonte: RFC 8555

Logo depois de realizar cada desafio, – como já demonstrado – o cliente envia um POST-as-GET para o endereço da autorização novamente. Dessa vez, o servidor disponibiliza outras informações a respeito do estado de cada desafio, conforme a seguir na Figura 18.

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "valid",
    "expires": "2018-09-09T14:09:01.13Z",
    "identifier": {
        "type": "dns",
        "value": "www.exemplo.org"
    },
    "challenges": [
        {
            "type": "http-01",
            "url": "https://example.com/acme/chall/p...4",
            "status": "valid",
            "validated": "2014-12-01T12:05:13.72Z",
            "token": "IlirfxKKXAsHtmzK29Pj8A"
        }
    ]
}
```

Figura 18 – Resposta ao POST pelo servidor ACME `/acme/authz`

Fonte: RFC 8555

Dessa maneira, o cliente pode verificar se já cumpriu todas as autorizações necessárias. Após isso, o cliente envia a CSR para o caminho *finalize* – como já explicado. Como pode ser visto abaixo na Figura 19.

```
1    POST /acme/order/TOlocE8rfgo/finalize HTTP/1.1
2    Host: example.com
3    Content-Type: application/jose+json
4
5    {
6        "protected": base64url({
7            "alg": "ES256",
8            "kid":
                "https://example.com/acme/acct/evOfKhNU60wg",
9            "nonce": "MSF2j2nawWHPxxkE3ZJtKQ",
10           "url":
                "https://example.com/acme/order/T...o/finalize"
11       }),
12       "payload": base64url({
13           "csr": "MIIBPTCBxAIBADBFMQ...FS6aKdZeGsysoCo4H9P",
14       }),
15       "signature": "uOrUfIIk5RyQ...nw62Ay1cl6AB"
16   }
```

Figura 19 – POST do cliente ACME `/acme/order`

Fonte: RFC 8555

Nesse ponto, a AC pode não querer emitir o certificado caso haja algum problema com a CSR, por exemplo:

- Se a CSR e os identificadores do pedido forem diferentes
- Se a conta não for autorizada para os identificadores indicados na CSR
- Se a CSR solicitar extensões que a AC não deseja incluir

Caso a CSR esteja bem formada o servidor ACME retorna informações atualizadas à respeito do pedido (Order), contendo o caminho para baixar o certificado no campo *certificate*, conforme abaixo na Figura 20.

```
HTTP/1.1 200 OK
Replay-Nonce: CGf81JWBsq8QyIgPCi9Q9X
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/order/TOlocE8rfgo

{
    "status": "valid",
    "expires": "2016-01-20T14:09:07.99Z",
    "notBefore": "2016-01-01T00:00:00Z",
    "notAfter": "2016-01-08T00:00:00Z",
    "identifiers": [
        { "type": "dns", "value": "www.exemplo.org" },
        { "type": "dns", "value": "exemplo.org" }
    ],
    "authorizations": [
        "https://example.com/acme/authz/PAniVnsZcis",
        "https://example.com/acme/authz/r4HqLzrSrpI"
    ],
    "finalize":
      "https://example.com/acme/order/T...o/finalize",
    "certificate":
      "https://example.com/acme/cert/mAt3xBGaobw"
}
```

Figura 20 – Resposta ao POST pelo servidor ACME `/acme/order`

Fonte: RFC 8555

Sendo assim, para baixar o certificado o cliente simplesmente envia um POST-as-GET para o endereço do campo *certificate*, como pode ser visto a seguir na Figura 21.

```
POST /acme/cert/mAt3xBGaobw HTTP/1.1
Host: example.com
Content-Type: application/jose+json
Accept: application/pem-certificate-chain

{
    "protected": base64url({
        "alg": "ES256",
        "kid":
           "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
        "url": "https://example.com/acme/cert/mAt3xBGaobw"
    }),
    "payload": "",
    "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
}
```

Figura 21 – POST do cliente ACME `/acme/cert`

Fonte: RFC 8555

E, dessa maneira, o servidor ACME responde com o certificado emitido, conforme abaixo na Figura 22.

```
HTTP/1.1 200 OK
Content-Type: application/pem-certificate-chain
Link: <https://example.com/acme/directory>;rel="index"

-----BEGIN CERTIFICATE-----
[End-entity certificate contents]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[Issuer certificate contents]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[Other certificate contents]
-----END CERTIFICATE-----
```

Figura 22 – Resposta ao POST pelo servidor ACME `/acme/cert`

Fonte: RFC 8555

Nota-se que a resposta do servidor não contém apenas o certificado emitido, adicionalmente contém o certificado da AC que o emitiu e outros possíveis certificados que constroem a cadeia de certificação.

A Figura 23 ilustra de maneira bem simplificada a emissão de um certificado utilizando o protocolo ACME.

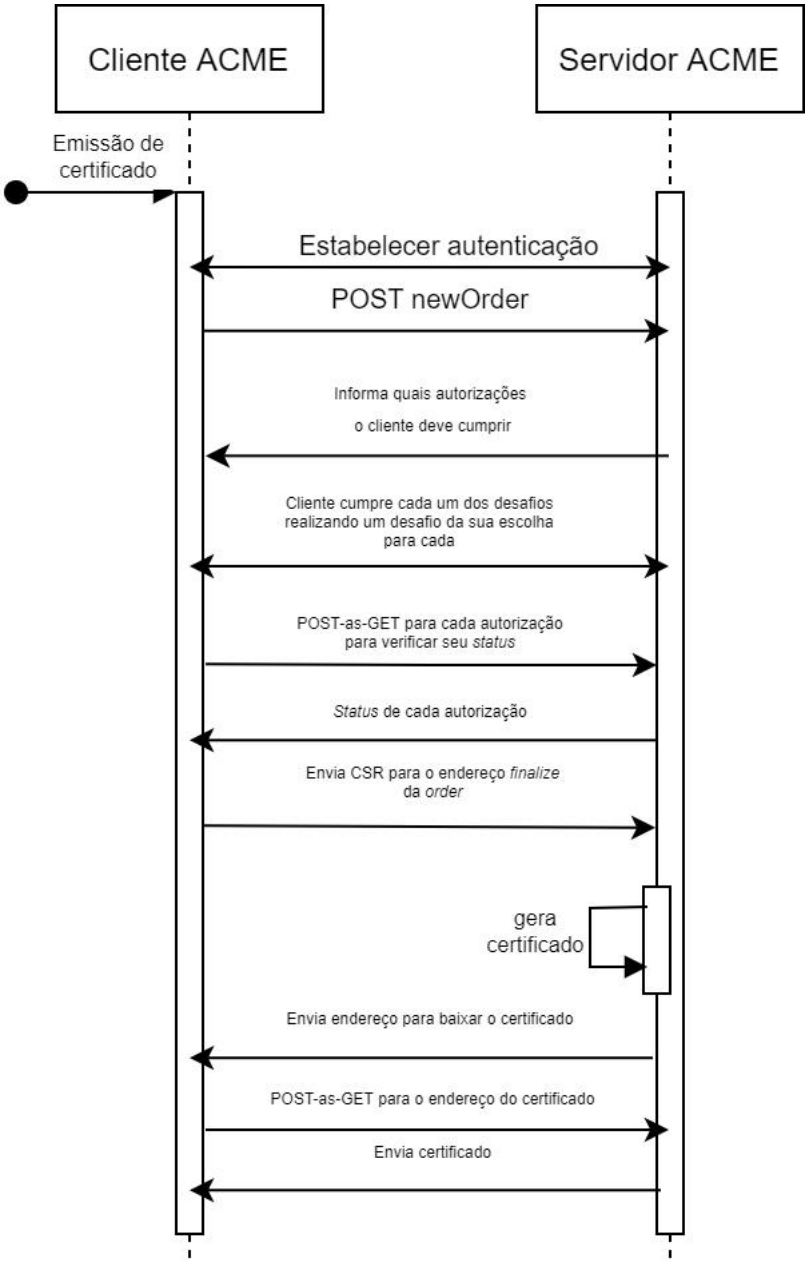

Figura 23 – Emissão de Certificado ACME

Fonte: RFC 8555

3.4 PROTOCOLO ACME PÓS-QUÂNTICO

Atualmente, na versão 2, o protocolo requisita a implementação de algoritmos de curvas elípticas, ou seja, algoritmos de criptografia clássica para assinatura digital, como mencionado anteriormente. Além disso, o protocolo não restringe a implementação de algoritmos adicionais de criptografia, dessa forma, sendo abragente neste sentido.

Sendo assim, a única mudança necessária para torná-lo pós-quântico é a integração de algoritmos pós-quânticos de assinatura digital às implementações do protocolo. Esses algoritmos podem ser tanto utilizados para realizar a troca de mensagens do protocolo – as quais são assinadas digitalmente – quanto para a assinatura e emissão

dos certificados digitais, tornando-os pós-quânticos.

Dessa maneira, resta escolher quais algoritmos pós-quânticos serão integrados às implementações do protocolo – tanto no cliente quanto no servidor ACME.

4 IMPLEMENTAÇÃO

Para escolha de quais implementações serão modificadas considerou-se, principalmente, os aspectos de: simplicidade e compatibilidade entre cliente e servidor. Sendo assim, como tem-se muito menos variedade de implementações do servidor ACME, considerou-se essa primeiro. Entre as implementações, o servidor Pebble (PEBBLE. . . , 2021) mostrou-se mais simples – feito para testes e prototipagem – sendo, dessa maneira, o escolhido. Já para a grande variedade de implementações cliente, considerou-se principalmente a compatibilidade com o servidor, i.e., implementados na mesma linguagem – neste caso, em Go – para que modificações em uma implementação sejam facilmente reproduzidas na outra. Dessa forma, escolheu-se a implementação LEGO (LEGO. . . , 2021) para o cliente, sendo a principal implementação do cliente ACME em Go disponibilizada.

Além disso, contou-se com a biblioteca LibOQS (LIBOQS. . . , 2021) a qual oferece implementações de diversos algoritmos pós-quânticos participando do processo de padronização do NIST (MOODY *et al.*, 2020). Dessa forma, a integração teve-se com alguns dos algoritmos já suportados nesta biblioteca a qual também oferece *bindings*, i.e., uma interface para utilizar funções escritas em uma linguagem em outra com facilidade – neste caso de C para Go. Outras bibliotecas em Go também tiveram que ser modificadas no processo, como será detalhado nas seções seguintes.

Neste trabalho optou-se por integrar os algoritmos de assinatura digital finalistas da terceira rodada do NIST de padronização, como já mencionado. Além desses, o algoritmo candidato SPHINCS+ também foi escolhido por cobrir casos de uso específicos, como IoT ou Internet das Coisas de maneira eficiente (BERNSTEIN; HÜLSING *et al.*, 2019). Foram escolhidos, para cada algoritmo, dois níveis de segurança para, dessa forma, ACs em diferentes níveis da cadeia de certificação terem diferentes níveis de segurança – esboçando um cenário mais realista. Sendo assim, os algoritmos escolhidos foram (segundo a nomenclatura da LibOQS):

- **Mais alto nível de segurança:**
  - dilithium5, dilithium5-aes
  - falcon-1024
  - rainbow-V-classic, rainbow-V-circumzenithal, rainbow-V-compressed
  - sphincs+-haraka-256s-simple, sphincs+-haraka-256f-simple, sphincs+-haraka-256s-robust, sphincs+-haraka-256f-robust, sphincs+-sha256-256s-simple, sphincs+-sha256-256f-simple, sphincs+-sha256-256s-robust, sphincs+-sha256-256f-robust, sphincs+-shake256-256s-simple, sphincs+-shake256-256f-simple, sphincs+-shake256-256s-robust, sphincs+-shake256-256f-robust

- **Mais baixo nível de segurança:**
  - dilithium2, dilithium2-aes
  - falcon-512
  - rainbow-I-classic, rainbow-I-circumzenithal, rainbow-I-compressed
  - sphincs+-haraka-128s-simple, sphincs+-haraka-128f-simple, sphincs+-haraka-128s-robust, sphincs+-haraka-128f-robust, sphincs+-sha256-128s-simple, sphincs+-sha256-128f-simple, sphincs+-sha256-128s-robust, sphincs+-sha256-128f-robust, sphincs+-shake256-128s-simple, sphincs+-shake256-128f-simple, sphincs+-shake256-128s-robust, sphincs+-shake256-128f-robust

Todo o código também pode ser encontrado em *github.com/zinho02/pqc-acme*.

4.1 LIBOQS

Como mencionado a LibOQS – *Library Open Quantum Safe* – oferece diversas implementações de algoritmos pós-quânticos, dos quais concentra-se os esforços neste trabalho nos algoritmos de assinatura digital.

Seguindo a própria descrição do projeto OQS, tem-se que o objetivo do mesmo é apoiar o desenvolvimento e a prototipagem de criptografia pós-quântica.

O projeto consiste em duas linhas principais de trabalho – destinando-se à prototipagem e avaliação da criptografia pós-quântica: LibOQS, uma biblioteca em C de código aberto para algoritmos criptográficos pós-quânticos e integrações de protótipos em protocolos e aplicativos, incluindo um *fork*, i.e., uma cópia seguida de modificações, da amplamente usada biblioteca OpenSSL.

Neste trabalho utilizou-se da biblioteca LibOQS para realizar a integração com os outros pacotes modificados. Mais especificamente, as *bindings* da LibOQS (LIBOQS. . . , 2021) – em C – para a linguagem Go, como será visto na seção seguinte.

4.2 MODIFICAÇÕES NA STANDARD GOLANG LIBRARY

Standard Golang Library ou a biblioteca padrão da linguagem Go (GOLANG. . . , 2021) à qual mantém diversos componentes bases utilizados por outras bibliotecas de terceiros. Ela oferece desde funcionalidades de sistemas operacionais até pacotes de testes automatizados. Contudo, concentra-se, nesse trabalho, no pacote **crypto**. Esse oferece diversas funcionalidades à respeito de funções criptográficas assimétricas – como já visto na Seção 2 –, simétricas, constantes, etc.

Criou-se uma interface para trabalhar com a *bindings* da LibOQS para as funções da assinatura digital, concentrando no arquivo **pqc.go**. Dessa forma, implementou-

se tanto a interface de **chave pública** quanto de **chave privada** e suas respectivas funções.

```
1     // PublicKey represents an PQC public key.
2     type PublicKey struct {
3         AlgName string
4         Bytes   []byte
5     }
```

Figura 24 – Estrutura da chave pública

Fonte: Original

No código a cima da Figura 24, tem-se a estrutura da **chave pública** pós-quântica genérica, i.e., pra qualquer algoritmo pós-quântico de assinatura digital oferecido pela biblioteca da LibOQS. Dessa forma, tem-se o nome do algoritmo a qual a chave pública está relacionada e seus *bytes* propriamente.

```
1     // PrivateKey represents an PQC private key.
2     type PrivateKey struct {
3         PublicKey
4         Signer oqs.Signature
5     }
```

Figura 25 – Estrutura da chave privada

Fonte: Original

Neste último código, ilustrado na Figura 25, tem-se, analogamente, a estrutura da **chave privada**. Contém-se a chave pública dentro da estrutura de chave privada para fins de recuperar a chave pública facilmente. Além disso, tem-se a estrutura **Signer** do signatário – que entre outras informações, contém os *bytes* da chave privada –, pois a estrutura é utilizada apenas para fins de assinatura digital, não sendo necessárias outras informações.

Em relação as funções que a estrutura da chave pública implementa, tem-se a verificação de igualdade entre duas chaves públicas e a verificação de uma assinatura.

```
1     // Equal reports whether public and x have the same value.
2     func (pub *PublicKey) Equal(x crypto.PublicKey) bool {
3         xx, ok := x.(*PublicKey)
4         if !ok {
5             return false
6         }
7         return bytes.Equal(pub.Bytes, xx.Bytes) && pub.AlgName ==
              xx.AlgName
8     }
```

Figura 26 – Função de verificação de igualdade de chaves públicas

Fonte: Original

No código da Figura 26, tem-se a verificação de igualdade entre duas chaves públicas. Para isso, verifica-se se os *bytes* das duas chaves são iguais e se seus algoritmos são os mesmos.

```
1     // Verifies the signature.
2     func (pub *PublicKey) Verify(data, signature []byte) bool {
3         verifier := oqs.Signature{}
4
5         if err := verifier.Init(pub.AlgName, nil); err != nil {
6             return false
7         }
8
9         isValid, err := verifier.Verify(data, signature,
              (*pub).Bytes)
10        if err != nil {
11            return false
12        }
13        return isValid
14    }
```

Figura 27 – Função de verificação da assinatura

Fonte: Original

Nesta outra função (Figura 27), tem-se a verificação de uma assinatura digital, utilizando a chave pública para tal. Verifica-se a assinatura – juntamente com a informação assinada – criando-se uma estrutura **Signature** com o algoritmo da chave pública. A partir disso, é chamado o método **Verify** da LibOQS, o qual recebe as informações assinadas, a assinatura e a chave pública. Caso a assinatura seja válida é retornado o valor **true** – verdadeiro –, caso contrário o valor **false** – falso.

Já para a chave privada, tem-se uma variedade maior de métodos implementados. Conta-se com métodos para: verificar a igualdade entre duas chaves privadas – similarmente ao da chave pública –, para recuperar a chave pública a partir da privada,

e para assinar.

```
1      // Equal reports whether private and x have the same value.
2      func (priv *PrivateKey) Equal(x crypto.PrivateKey) bool {
3          xx, ok := x.(*PrivateKey)
4          if !ok {
5              return false
6          }
7          return priv.PublicKey.Equal(&xx.PublicKey) &&
              bytes.Equal(priv.Signer.ExportSecretKey(),
              xx.Signer.ExportSecretKey())
8      }
```

Figura 28 – Função de verificação de igualdade de chaves privadas

Fonte: Original

Neste código da Figura 28, tem-se a verificação de igualdade entre duas chaves privadas. Para isso, duas chaves privadas são ditas iguais se as suas chaves públicas são iguais – como visto anteriormente – e se os bytes das chaves privadas são iguais.

```
1      // Public returns the public key corresponding to private key.
2      func (priv *PrivateKey) Public() crypto.PublicKey {
3          return &priv.PublicKey
4      }
5
6      func (priv *PrivateKey) PQCPublic() *PublicKey {
7          return priv.Public().(*PublicKey)
8      }
```

Figura 29 – Funções de recuperação da chave pública

Fonte: Original

Tem-se, também, dois métodos para recuperar a chave pública na Figura 29. O primeiro genérico do pacote **crypto**. Já o segundo, recupera a chave pública e realiza um *cast*, i.e., uma transformação, para a chave pública pós-quântica. Isso foi realizado apenas por fins de simplificar o código, já que na maioria dos casos quer-se utilizar a chave pública pós-quântica.

```
1     /* Signs data and returns the signature.
2     rand and opts are not being used.
3     */
4     func (priv *PrivateKey) Sign(rand io.Reader, data []byte, opts
          crypto.SignerOpts) ([]byte, error) {
5         signature, err := priv.Signer.Sign(data)
6         if err != nil {
7             return nil, err
8         }
9         return signature, nil
10    }
```

Figura 30 – Função de assinatura

Fonte: Original

Neste último método da chave privada (Figura 30), tem-se a realização da assinatura digital. Por questões de compatibilidade com a interface de chave privada do pacote **crypto** precisa-se de parâmetros a mais que, neste caso, não são utilizados, como o *rand* e o *opts*. A assinatura é realizada utilizando o método **Sign** da LibOQS a partir do **Signer** da estrutura de chave privada implementada, dessa maneira, recebendo as informações que se deseja assinar. O retorno pode ser uma assinatura nula e um erro caso haja algum problema ou pode ser a assinatura com sucesso sem nenhum erro – um erro nulo.

Além dos métodos implementados para as estruturas de chave pública e privada, tem-se também o método para gerá-las, i.e., o método que gera a chave privada – a qual já contém a pública, como pode ser visto na Figura 31.

```
1     // Generates a key pair and returns the private key.
2     func GenerateKey(signatureName string) (*PrivateKey, error) {
3         signer := oqs.Signature{}
4         if err := signer.Init(signatureName, nil); err != nil {
5             return nil, err
6         }
7
8         pubKeyBytes, err := signer.GenerateKeyPair()
9         if err != nil {
10            return nil, err
11        }
12
13        privKey := new(PrivateKey)
14        privKey.AlgName = signatureName
15        privKey.Bytes = pubKeyBytes
16        privKey.Signer = signer
17
18        return privKey, nil
19    }
```

Figura 31 – Função de geração de par de chaves

Fonte: Original

Para gerar, recebe-se como parâmetro o algoritmo o qual será instânciado a chave privada. Dessa forma, instância-se o **Signer** a partir do método **Init** da LibOQS, recebendo o algoritmo pós-quântico. Feito isso, gera-se o par de chaves de fato a partir do método **GenerateKeyPair** da LibOQS, retornando os *bytes* da chave pública. E, por fim, atribui-se cada um desses valores – algoritmo utilizado, bytes da chave pública e estrutura para assinar – aos da estrutura da chave privada.

E, por fim, são integrados os OIDs (OID..., 2022) dos algoritmos pós-quânticos e métodos para recuperar o nome do algoritmo a partir do OID e o OID a partir do nome do algoritmo. Como os OIDs não são oficiais, são utilizados aqueles sugeridos pela própria LibOQS, podendo ser encontrados no *github* deles (LIBOQS..., 2022). Por conta da extensibilidade, preferiu-se não mostrar em texto esta parte.

Além disso, outro importante arquivo modificado foi **x509.go**. Este implementa código referente aos certificados **X.509** (COOPER *et al.*, 2008b) – a principal versão de certificados utilizada. No mesmo foram modificadas as partes do código necessárias para integrar com a inteface pós-quântica criada.

As modificações podem ser como: modificações em funções para adicionar os casos da interface pós-quântica – tipicamente uma estrutura *switch-case* – e modificações para adicionar constantes e variáveis da interface pós-quântica, para manter a estrutura do código sem complexidades adicionais desnecessárias.

```
1   func marshalPublicKey(pub interface{}) (publicKeyBytes []byte,
        publicKeyAlgorithm pkix.AlgorithmIdentifier, err error) {
2       switch pub := pub.(type) {
3       ...
4       case *pqc.PublicKey:
5           publicKeyBytes = pub.Bytes
6           if err != nil {
7               return nil, pkix.AlgorithmIdentifier{}, err
8           }
9           publicKeyAlgorithm.Algorithm =
                pqc.GetPublicKeyOIDFromPublicKey(pub.AlgName)
10      default:
11          return nil, pkix.AlgorithmIdentifier{},
                fmt.Errorf("x509: unsupported public key type: %T",
                pub)
12      }
13      return publicKeyBytes, publicKeyAlgorithm, nil
14  }
15
16  ...
17
18  func getPublicKeyAlgorithmFromOID(oid asn1.ObjectIdentifier)
        PublicKeyAlgorithm {
19      switch {
20      ...
21      case oid.Equal(oidPublicKeyRainbowVCompressed):
22          return RainbowVCompressed
23      ...
24      case oid.Equal(oidPublicKeyDilithium2AES):
25          return Dilithium2AES
26      ...
27      case oid.Equal(oidPublicKeySphincsPlusSHAKE256128fRobust):
28          return SphincsPlusSHAKE256128fRobust
29      }
30      return UnknownPublicKeyAlgorithm
31  }
```

Figura 32 – Novos casos para funções existentes

Fonte: Original

```
1   func checkSignature(algo SignatureAlgorithm, signed, signature
        []byte, publicKey crypto.PublicKey) (err error) {
2       ...
3       switch pub := publicKey.(type) {
4       case *pqc.PublicKey:
5           if !pub.Verify(signed, signature) {
6               return errors.New("x509: " + pub.AlgName +
                    "verification failure")
7           }
8           return
9       }
10      ...
11      return ErrUnsupportedAlgorithm
12  }
13
14  func signingParamsForPublicKey(pub interface{}, requestedSigAlgo
        SignatureAlgorithm) (hashFunc crypto.Hash, sigAlgo
        pkix.AlgorithmIdentifier, err error) {
15      var pubType PublicKeyAlgorithm
16
17      switch pub := pub.(type) {
18      case *pqc.PublicKey:
19          switch pub.AlgName {
20          case "dilithium5":
21              pubType = Dilithium5
22              sigAlgo.Algorithm = oidSignatureDilithium5
23          ...
24          case "sphincs+-sha256-256s-robust":
25              pubType = SphincsPlusSHA256256sRobust
26              sigAlgo.Algorithm =
                    oidSignatureSphincsPlusSHA256256sRobust
27          ...
28          case "dilithium2":
29              pubType = Dilithium2
30              sigAlgo.Algorithm = oidSignatureDilithium2
31          ...
32          case "sphincs+-shake256-128f-robust":
33              pubType = SphincsPlusSHAKE256128fRobust
34              sigAlgo.Algorithm =
                    oidSignatureSphincsPlusSHAKE256128fRobust
35          }
36      ...
37      return
38  }
```

Figura 33 – Novos casos para funções existentes

Fonte: Original

Estes primeiros métodos, como vistos na Figura 32 e na Figura 33, se encaixam nos quais apenas foram necessários criar novos casos para a interface pós-quântica

implementada, sem muito esforço adicional. Percebe-se que foram omitidas algumas partes para não estender demais, porém sem perda didática.

```
type SignatureAlgorithm int

const (
    UnknownSignatureAlgorithm SignatureAlgorithm = iota
    ...
    PureDilithium5
    ...
    PureRainbowIClassic
    ...
    PureSphincsPlusSHAKE256128fRobust
)

...

type PublicKeyAlgorithm int

const (
    UnknownPublicKeyAlgorithm PublicKeyAlgorithm = iota
    ...
    Falcon1024
    ...
    RainbowICompressed
    SphincsPlusSHAKE256128sSimple
    ...
)
```

Figura 34 – Modificações em constantes e seus usos

Fonte: Original

```
1     var publicKeyAlgoName = [...]string{
2         ...
3         RainbowVClassic:                "Rainbow-V-Classic",
4         RainbowVCircumzenithal:         "Rainbow-V-Circumzenithal",
5         ...
6         Dilithium2:                     "Dilithium2",
7         ...
8         SphincsPlusSHAKE256128fRobust:
              "SPHINCS+-SHAKE256-128f-Robust",
9     }
10
11    ...
12
13    var (
14        ...
15        oidSignatureDilithium5                    =
              pqc.OIDSignatureDilithium5
16        oidSignatureDilithium5AES                 =
              pqc.OIDSignatureDilithium5AES
17        oidSignatureFalcon1024                    =
              pqc.OIDSignatureFalcon1024
18        ...
19        oidSignatureSphincsPlusSHAKE256128fSimple =
              pqc.OIDSignatureSphincsPlusSHAKE256128fSimple
20        ...
21    )
22
23    ...
24
25    var signatureAlgorithmDetails = []struct {
26        algo       SignatureAlgorithm
27        name       string
28        oid        asn1.ObjectIdentifier
29        pubKeyAlgo PublicKeyAlgorithm
30        hash       crypto.Hash
31    }{
32        ...
33        {PureFalcon1024, "Falcon-1024", oidSignatureFalcon1024,
              Falcon1024, crypto.Hash(0)},
34        ...
35        {PureSphincsPlusSHAKE256128fRobust,
              "SPHINCS+-SHAKE256-128f-Robust",
              oidSignatureSphincsPlusSHAKE256128fRobust,
              SphincsPlusSHAKE256128fRobust, crypto.Hash(0)},
36    }
```

Figura 35 – Modificações em constantes e seus usos

Fonte: Original

Nesta segunda parte (Figura 34 e Figura 35), as principais modificações foram

em constantes e seus usos. Percebe-se, além da criação de novas contantes para os algoritmos integrados, seus OIDs e a união de informações sobre os respectivos algoritmos como explicitado na última variável acima **signatureAlgorithmDetails**. Fora as omissões, estas foram as modificações neste arquivo.

Outro arquivo modificado na biblioteca padrão do Go foi **pkcs8.go**. Este arquivo representa uma chave pública no padrão PKCS8 (KALISKI, 2008) – um padrão para guardar informações da chave privada. Foi adicionado um atributo a estrutura pkcs8 que representa os *bytes* da chave pública correspondente a chave privada. Isso foi necessário para poder recuperar totalmente a chave privada a partir dos bytes dessa estrutura nos métodos modificados.

```
1     func ParsePKCS8PrivateKey(der []byte) (key interface{}, err
         error) {
2         var privKey pkcs8
3         if _, err := asn1.Unmarshal(der, &privKey); err != nil {
4             if _, err := asn1.Unmarshal(der, &ecPrivateKey{}); err
                 == nil {
5                 return nil, errors.New("x509: failed to parse
                     private key (use ParseECPrivateKey instead for
                     this key format)")
6             }
7             if _, err := asn1.Unmarshal(der, &pkcs1PrivateKey{});
                 err == nil {
8                 return nil, errors.New("x509: failed to parse
                     private key (use ParsePKCS1PrivateKey instead for
                     this key format)")
9             }
10            return nil, err
11        }
12        switch {
13        case privKey.Algo.Algorithm.Equal(oidPublicKeyRSA):
14            key, err = ParsePKCS1PrivateKey(privKey.PrivateKey)
15            if err != nil {
16                return nil, errors.New("x509: failed to parse RSA
                     private key embedded in PKCS#8: " + err.Error())
17            }
18            return key, nil
19        ...
20        case privKey.Algo.Algorithm.Equal(
21        oidPublicKeySphincsPlusSHA256128sSimple):
22            algName := pqc.GetPublicKeyFromPublicKeyOID(
23            oidPublicKeySphincsPlusSHA256128sSimple)
24            key := pqc.PrivateKey{
25                PublicKey: pqc.PublicKey{
26                    Bytes:   privKey.PublicKey,
27                    AlgName: algName,
28                },
29            }
30            key.Signer.Init(algName, privKey.PrivateKey)
31            return key, nil
32        ...
33    }
```

Figura 36 – Funções PKCS8

Fonte: Original

```
1     func MarshalPKCS8PrivateKey(key interface{}) ([]byte, error) {
2         var privKey pkcs8
3
4         switch k := key.(type) {
5         case *rsa.PrivateKey:
6             privKey.Algo = pkix.AlgorithmIdentifier{
7                 Algorithm:  oidPublicKeyRSA,
8                 Parameters: asn1.NullRawValue,
9             }
10            privKey.PrivateKey = MarshalPKCS1PrivateKey(k)
11        case *pqc.PrivateKey:
12            privKey.Algo = pkix.AlgorithmIdentifier{
13                Algorithm:
                      pqc.GetPublicKeyOIDFromPublicKey(k.AlgName),
14                Parameters: asn1.NullRawValue,
15            }
16            privKey.PublicKey = k.Bytes
17            privKey.PrivateKey = k.Signer.ExportSecretKey()
18        ...
19        return asn1.Marshal(privKey)
20    }
```

Figura 37 – Funções PKCS8

Fonte: Original

Estas duas funções da Figura 36 e da Figura 37 foram as únicas modificadas no arquivo – nenhuma nova foi adicionada. Na função **ParsePKCS8PrivateKey**, recupera-se a chave privada a partir de *bytes* da mesma – já serializados. Adiciona-se um caso para cada algoritmo pós-quântico integrado. Esses *bytes* os quais recebe-se nessa função são gerados a partir do método **MarshalPKCS8PrivateKey**, o qual recebe a chave privada e retorna os *bytes* referentes a ela. Para isso, criou-se um caso para a interface pós-quântica a ser tratado.

E, por fim, modificou-se o arquivo **parser.go**. Este arquivo lida com a mudança na representação de informações, neste caso, principalmente, em relação as chaves criptográficas.

```
1     func parsePublicKey(algo PublicKeyAlgorithm, keyData
         *publicKeyInfo) (interface{}, error) {
2         der := cryptobyte.String(keyData.PublicKey.RightAlign())
3         switch algo {
4         ...
5         case Dilithium5:
6             pub := &pqc.PublicKey{
7                 Bytes:   keyData.PublicKey.Bytes,
8                 AlgName: "dilithium5",
9             }
10            return pub, nil
11        ...
12        case RainbowIClassic:
13            pub := &pqc.PublicKey{
14                Bytes:   keyData.PublicKey.Bytes,
15                AlgName: "rainbow-i-classic",
16            }
17            return pub, nil
18        ...
19        case SphincsPlusSHAKE256128fRobust:
20            pub := &pqc.PublicKey{
21                Bytes:   keyData.PublicKey.Bytes,
22                AlgName: "sphincs+-shake256-128f-robust",
23            }
24            return pub, nil
25        ...
26        }
27    }
```

Figura 38 – Função parsePublicKey

Fonte: Original

Modificou-se apenas a função **parsePublicKey** (Figura 38) neste arquivo. A qual a partir de informações da chave pública a transforma em uma inteface mais genérica. Para isso, foi necessário adicionar os casos dos novos algoritmos integrados.

4.3 GO JOSE

A biblioteca Go JOSE é uma implementação em Go do conjunto de padrões *Javascript Object Signing and Encryption* ou Assinatura e Criptografia de Objetos *Javascript*. Isso inclui suporte aos padrões *JSON Web Encryption* (JONES; HILDEBRAND, 2015), *JSON Web Signature* (JONES; BRADLEY; SAKIMURA, 2015a), and *JSON Web Token* (JONES; BRADLEY; SAKIMURA, 2015b).

Esse pacote tem uma grande importância no trabalho, pois as mensagens trocadas entre o cliente e o servidor ACME são mensagens JSON. Além disso, essas mensagens são assinadas e requerem a integração do suporte aos algoritmos pós-

quânticos para que, dessa maneira, o protocolo ACME se torne pós-quântico. Logo, além de utilizar a versão modificada a cima da biblioteca padrão do Go, também alterou-se esta.

Primeiramente, tem-se o arquivo **cryptosigner.go** o qual lida com a assinatura de dados no formato JSON. Neste caso, adicionou-se o necessário para integrar com os algoritmos pós-quânticos, como pode ser visto na Figura 39 e na Figura 40.

```
1     func (s *cryptoSigner) Algs() []jose.SignatureAlgorithm {
2         switch s.signer.Public().(type) {
3         ...
4         case *pqc.PublicKey:
5             return []jose.SignatureAlgorithm{jose.Dilithium5,
                  jose.Dilithium5AES, jose.Falcon1024,
                  jose.RainbowVClassic,
6                 jose.RainbowVCircumzenithal,
                      jose.RainbowVCompressed,
                      jose.SphincsPlusHaraka256sSimple,
7                 jose.SphincsPlusHaraka256fSimple,
                      jose.SphincsPlusHaraka256sRobust,
                      jose.SphincsPlusHaraka256fRobust,
8                 jose.SphincsPlusSHA256256fSimple,
                      jose.SphincsPlusSHA256256sSimple,
                      jose.SphincsPlusSHA256256sRobust,
9                 jose.SphincsPlusSHA256256fRobust,
                      jose.SphincsPlusSHAKE256256sSimple,
                      jose.SphincsPlusSHAKE256256fSimple,
10                jose.SphincsPlusSHAKE256256sRobust,
                      jose.SphincsPlusSHAKE256256fRobust,
                      jose.Dilithium2, jose.Dilithium2AES,
                      jose.Falcon512, jose.RainbowIClassic,
11                jose.RainbowICircumzenithal,
                      jose.RainbowICompressed,
                      jose.SphincsPlusHaraka128sSimple,
12                jose.SphincsPlusHaraka128fSimple,
                      jose.SphincsPlusHaraka128sRobust,
                      jose.SphincsPlusHaraka128fRobust,
13                jose.SphincsPlusSHA256128fSimple,
                      jose.SphincsPlusSHA256128sSimple,
                      jose.SphincsPlusSHA256128sRobust,
14                jose.SphincsPlusSHA256128fRobust,
                      jose.SphincsPlusSHAKE256128sSimple,
                      jose.SphincsPlusSHAKE256128fSimple,
15                jose.SphincsPlusSHAKE256128sRobust,
                      jose.SphincsPlusSHAKE256128fRobust}
16        default:
17            return nil
18        }
19    }
```

Figura 39 – Funções modificadas do arquivo cryptosigner.go

Fonte: Original

```
1     func (s *cryptoSigner) SignPayload(payload []byte, alg
         jose.SignatureAlgorithm) ([]byte, error) {
2         var hash crypto.Hash
3         switch alg {
4         ...
5         case jose.Dilithium5, jose.Dilithium5AES, jose.Falcon1024,
             jose.RainbowVClassic,
6             jose.RainbowVCircumzenithal, jose.RainbowVCompressed,
                 jose.SphincsPlusHaraka256sSimple,
7             jose.SphincsPlusHaraka256fSimple,
                 jose.SphincsPlusHaraka256sRobust,
                 jose.SphincsPlusHaraka256fRobust,
8             jose.SphincsPlusSHA256256fSimple,
                 jose.SphincsPlusSHA256256sSimple,
                 jose.SphincsPlusSHA256256sRobust,
9             jose.SphincsPlusSHA256256fRobust,
                 jose.SphincsPlusSHAKE256256sSimple,
                 jose.SphincsPlusSHAKE256256fSimple,
10            jose.SphincsPlusSHAKE256256sRobust,
                 jose.SphincsPlusSHAKE256256fRobust, jose.Dilithium2,
                 jose.Dilithium2AES, jose.Falcon512,
                 jose.RainbowIClassic,
11            jose.RainbowICircumzenithal, jose.RainbowICompressed,
                 jose.SphincsPlusHaraka128sSimple,
12            jose.SphincsPlusHaraka128fSimple,
                 jose.SphincsPlusHaraka128sRobust,
                 jose.SphincsPlusHaraka128fRobust,
13            jose.SphincsPlusSHA256128fSimple,
                 jose.SphincsPlusSHA256128sSimple,
                 jose.SphincsPlusSHA256128sRobust,
14            jose.SphincsPlusSHA256128fRobust,
                 jose.SphincsPlusSHAKE256128sSimple,
                 jose.SphincsPlusSHAKE256128fSimple,
15            jose.SphincsPlusSHAKE256128sRobust,
                 jose.SphincsPlusSHAKE256128fRobust:
16            return s.signer.Sign(nil, payload, nil)
17        default:
18            return nil, jose.ErrUnsupportedAlgorithm
19        }
20        ...
21        return out, err
22    }
```

Figura 40 – Funções modificadas do arquivo cryptosigner.go

Fonte: Original

Na função **Algs**, a qual retorna uma lista de algoritmos de assinatura, adicionou-se os casos para os algoritmos de assinatura suportados pela interface pós-quântica. Já no método **SignPayload**, onde um array de *bytes* é assinado pelo algoritmo de

assinatura passado como parâmetro, adicionou-se o caso onde é utilizado algum dos algoritmos integrados.

Outro arquivo modificado foi **asymmetric.go** – Figura 41, 42 e 43 – o qual trata da implementação dos padrões de criptografia assimétrica, os mesmos que utilizamos para assinar digitalmente.

```
1     // A generic PQC-based encrypter/verifier
2     type pqcEncrypterVerifier struct {
3         publicKey *pqc.PublicKey
4     }
5
6     ...
7
8     // A generic PQC-based decrypter/signer
9     type pqcDecrypterSigner struct {
10        privateKey *pqc.PrivateKey
11    }
12
13    ...
14
15    // newPQCRecipient creates recipientKeyInfo based on the given
          key.
16    func newPQCRecipient(keyAlg KeyAlgorithm, publicKey
          *pqc.PublicKey) (recipientKeyInfo, error) {
17        if publicKey == nil {
18            return recipientKeyInfo{}, errors.New("invalid public
                  key")
19        }
20
21        return recipientKeyInfo{
22            keyAlg: keyAlg,
23            keyEncrypter: &pqcEncrypterVerifier{
24                publicKey: publicKey,
25            },
26        }, nil
27    }
```

Figura 41 – Funções do arquivo asymmetric.go

Fonte: Original

```
1     // newPQCSigner creates a recipientSigInfo based on the given
          key.
2     func newPQCSigner(sigAlg SignatureAlgorithm, privateKey
          *pqc.PrivateKey) (recipientSigInfo, error) {
3         if privateKey == nil {
4             return recipientSigInfo{}, errors.New("invalid private
                  key")
5         }
6
7         return recipientSigInfo{
8             sigAlg: sigAlg,
9             publicKey: staticPublicKey(&JSONWebKey{
10                Key: privateKey.Public(),
11            }),
12            signer: &pqcDecrypterSigner{
13                privateKey: privateKey,
14            },
15        }, nil
16    }
17
18    ...
19
20    // Encrypt the given payload and update the object.
21    func (ctx pqcEncrypterVerifier) encryptKey(cek []byte, alg
          KeyAlgorithm) (recipientInfo, error) {
22        encryptedKey, err := ctx.encrypt(cek, alg)
23        if err != nil {
24            return recipientInfo{}, err
25        }
26
27        return recipientInfo{
28            encryptedKey: encryptedKey,
29            header:       &rawHeader{},
30        }, nil
31    }
```

Figura 42 – Funções do arquivo asymmetric.go

Fonte: Original

```
1     func (ctx pqcEncrypterVerifier) encrypt(cek []byte, alg
         KeyAlgorithm) ([]byte, error) {
2         return nil, ErrUnsupportedAlgorithm
3     }
4
5     ...
6
7     // Decrypt the given payload and return the content encryption
         key.
8     func (ctx pqcDecrypterSigner) decryptKey(headers rawHeader,
         recipient *recipientInfo, generator keyGenerator) ([]byte,
         error) {
9         return ctx.decrypt(recipient.encryptedKey,
             headers.getAlgorithm(), generator)
10    }
11
12    ...
13
14    func (ctx pqcDecrypterSigner) decrypt(jek []byte, alg
         KeyAlgorithm, generator keyGenerator) ([]byte, error) {
15        return nil, ErrUnsupportedAlgorithm
16    }
17
18    ...
19
20    // Sign the given payload
21    func (ctx pqcDecrypterSigner) signPayload(payload []byte, alg
         SignatureAlgorithm) (Signature, error) {
22        out, _ := ctx.privateKey.Sign(nil, payload, nil)
23
24        return Signature{
25            Signature: out,
26            protected: &rawHeader{},
27        }, nil
28    }
29
30    ...
31
32    // Verify the given payload
33    func (ctx pqcEncrypterVerifier) verifyPayload(payload []byte,
         signature []byte, alg SignatureAlgorithm) error {
34        if ctx.publicKey.Verify(payload, signature) {
35            return nil
36        }
37        return ErrUnsupportedAlgorithm
38    }
```

Figura 43 – Funções do arquivo asymmetric.go

Fonte: Original

Criou-se os tipos **pqcEncrypterVerifier**, o qual guarda uma chave pública pós-quântica, assim como **pqcDecrypterSigner**, o qual guarda uma chave privada pós-quântica. O primeiro serve para realizar operações de cifragem e assinatura e o segundo operações de decifragem e verificação de assinaturas. No caso deste trabalho, só utilizou-se as operações relacionadas a assinatura.

Implementou-se o método **newPQCRecipient**, o qual cria uma estrutura para guardar informações adicionais da chave e do tipo **pqcEncrypterVerifier**. Dessa forma, também criou-se o método **newPQCSigner**, o qual guarda informações adicionais da chave privada e do tipo **pqcDecrypterSigner**.

Implementou-se, apenas por completude, os métodos **encrypt**, **encryptKey**, **decrypt** e **decryptKey**, pois não utilizou-se operações de cifragem na implementação, apenas de assinatura digital.

Implementou-se também o método **signPayload** para realizar a assinatura de um vetor de *bytes* utilizando a chave privada pós-quântica contida no tipo **pqcDecrypterSigner**. Assim, também foi implementado o método **verifyPayload**, o qual verifica se uma dada assinatura de um vetor de *bytes* está correta.

Também modificou-se o arquivo **crypter.go** que trata da cifragem de mensagens JSON e operações relacionadas que foram necessárias mudanças para completar a integração pós-quântica, como pode ser visto na Figura 44.

```
1     func makeJWERecipient(alg KeyAlgorithm, encryptionKey
         interface{}) (recipientKeyInfo, error) {
2         switch encryptionKey := encryptionKey.(type) {
3         case *pqc.PublicKey:
4             return newPQCRecipient(alg, encryptionKey)
5         ...
6         return recipientKeyInfo{}, ErrUnsupportedKeyType
7     }
8
9     ...
10
11    // newDecrypter creates an appropriate decrypter based on the
         key type
12    func newDecrypter(decryptionKey interface{}) (keyDecrypter,
         error) {
13        switch decryptionKey := decryptionKey.(type) {
14        case *pqc.PrivateKey:
15            return &pqcDecrypterSigner{
16                privateKey: decryptionKey,
17            }, nil
18        ...
19        return nil, ErrUnsupportedKeyType
20    }
```

Figura 44 – Funções do arquivo crypter.go

Fonte: Original

No método **makeJWERecipient** adicionou-se o caso onde cobre a chave pública pós-quântica, utilizando outro método implementado**newPQCRecipient** já mencionado. O mesmo acontece no método **newDecrypter**, onde adicionou-se um caso para a chave privada pós-quântica.

Outro arquivo modificado foi **jwk.go** – Figura 45, 46, 47 e 48 – o qual trata de JSON Web Key (JONES, 2015b) e sua manipulação. Ou seja, como são representadas as chaves para trabalhar em cima de dados JSON e operações à respeito das mesmas.

```
1     // rawJSONWebKey represents a public or private key in JWK
          format, used for parsing/serializing.
2     type rawJSONWebKey struct {
3         ...
4         PQCPub  *byteBuffer `json:"pqcpub,omitempty"`
5         PQCPriv *byteBuffer `json:"pqcpriv,omitempty"`
6         ...
7     }
8
9     ...
10
11    // MarshalJSON serializes the given key to its JSON
          representation.
12    func (k JSONWebKey) MarshalJSON() ([]byte, error) {
13        var raw *rawJSONWebKey
14        var err error
15
16        switch key := k.Key.(type) {
17        case *pqc.PublicKey:
18            raw = fromPQCPublicKey(key)
19        case *pqc.PrivateKey:
20            raw, err = fromPQCPrivateKey(key)
21        ...
22        }
23
24        if err != nil {
25            return nil, err
26        }
27
28        raw.Kid = k.KeyID
29        raw.Alg = k.Algorithm
30        raw.Use = k.Use
31
32        ...
33
34        return json.Marshal(raw)
35    }
```

Figura 45 – Modificações do arquivo jwk.go

Fonte: Original

```
1          // UnmarshalJSON reads a key from its JSON representation.
2      func (k *JSONWebKey) UnmarshalJSON(data []byte) (err error) {
3          var raw rawJSONWebKey
4          err = json.Unmarshal(data, &raw)
5          if err != nil {
6              return err
7          }
8
9          ...
10
11         switch raw.Kty {
12         case "dilithium5", "dilithium5-aes", "falcon-1024",
               "rainbow-v-classic", "rainbow-v-circumzenithal",
13             "rainbow-v-compressed", "sphincs+-haraka-256s-simple",
                   "sphincs+-haraka-256f-simple",
14             "sphincs+-haraka-256s-robust",
                   "sphincs+-haraka-256f-robust",
                   "sphincs+-sha256-256s-simple",
15             "sphincs+-sha256-256f-simple",
                   "sphincs+-sha256-256s-robust",
                   "sphincs+-sha256-256f-robust",
16             ..., "sphincs+-shake256-128f-robust":
17             if raw.PQCPriv != nil {
18                 key, err = raw.pqcPrivateKey()
19                 if err == nil {
20                     keyPub = key.(*pqc.PrivateKey).Public()
21                 }
22             } else {
23                 key, err = raw.pqcPublicKey()
24                 keyPub = key
25             }
26         ...
27
28         return
29     }
```

Figura 46 – Modificações do arquivo jwk.go

Fonte: Original

```
1           const pqcThumbprintTemplate = `{"pub":"%s","kty":"%s"}`
2
3       ...
4
5       func pqcThumbprintInput(k *pqc.PublicKey) (string, error) {
6           return fmt.Sprintf(pqcThumbprintTemplate,
7               newBuffer(k.Bytes).base64(),
8               newBuffer([]byte(k.AlgName)).base64()), nil
9       }
10
11      // IsPublic returns true if the JWK represents a public key (not
            symmetric, not private).
12      func (k *JSONWebKey) IsPublic() bool {
13          switch k.Key.(type) {
14          case *ecdsa.PublicKey, *rsa.PublicKey, ed25519.PublicKey,
                *pqc.PublicKey:
15              return true
16          default:
17              return false
18          }
19      }
20
21      // Public creates JSONWebKey with corresponding public key if
            JWK represents asymmetric private key.
22      func (k *JSONWebKey) Public() JSONWebKey {
23          if k.IsPublic() {
24              return *k
25          }
26          ret := *k
27          switch key := k.Key.(type) {
28          case *pqc.PrivateKey:
29              ret.Key = key.Public()
30          ...
31          return ret
32      }
33
34      // Valid checks that the key contains the expected parameters.
35      func (k *JSONWebKey) Valid() bool {
36          if k.Key == nil {
37              return false
38          }
39          switch key := k.Key.(type) {
40          case *pqc.PublicKey:
41              if key.Bytes == nil {
42                  return false
43              }
44          ...
45          return true
46      }
```

Figura 47 – Modificações do arquivo jwk.go

Fonte: Original

```
1          func (key rawJSONWebKey) pqcPublicKey() (*pqc.PublicKey,
               error) {
2          return &pqc.PublicKey{
3              AlgName: key.Kty,
4              Bytes:   key.PQCPub.bytes(),
5          }, nil
6      }
7
8      ...
9
10     func fromPQCPublicKey(pub *pqc.PublicKey) *rawJSONWebKey {
11         return &rawJSONWebKey{
12             Kty:    pub.AlgName,
13             PQCPub: newBuffer(pub.Bytes),
14         }
15     }
16
17     ...
18
19     func (key rawJSONWebKey) pqcPrivateKey() (*pqc.PrivateKey,
           error) {
20         privKey := &pqc.PrivateKey{
21             PublicKey: pqc.PublicKey{
22                 AlgName: key.Kty,
23                 Bytes:   key.PQCPub.bytes(),
24             },
25         }
26
27         privKey.Signer.Init(key.Kty, key.PQCPriv.bytes())
28
29         return privKey, nil
30     }
31
32     ...
33
34     func fromPQCPrivateKey(pqc *pqc.PrivateKey) (*rawJSONWebKey,
           error) {
35         raw := fromPQCPublicKey(&pqc.PublicKey)
36
37         raw.PQCPriv = newBuffer(pqc.Signer.ExportSecretKey())
38
39         return raw, nil
40     }
41
42     ...
```

Figura 48 – Modificações do arquivo jwk.go

Fonte: Original

No tipo **rawJSONWebKey** foram adicionados dois novos atributos: **PQCPub** e

PQCPriv, cada um com os *bytes* da chave pública e privada respectivamente. No método **MarshalJSON**, onde uma dada **JSONWebKey** é serializada pra sua representação JSON, adicionou-se casos para chave pública e privada pós-quântica. Da mesma forma, foram adicionados casos no método **UnmarshalJSON** para a chave pública e privada pós-quântica, operação inversa da anterior.

Também criou-se uma constante que representa o template do *thumbprint* – digital – de uma chave pública pós-quântica de qualquer algoritmo pós-quântica que chamou-se de **pqcThumbprintTemplate**. Esse *template*, ou modelo, que é utilizado pelo método **pqcThumbprintInput**, o qual foi implementado. Da mesma forma, foram adicionados casos à função **Thumbprint** para as chaves públicas e privadas pós-quântica para utilizarem o método anterior citado.

Para os métodos **IsPublic** e **Public** também foram adicionados casos para cobrir as chaves pós-quânticas. Assim como foi adicionado no método **Valid** um caso também.

Foram implementados os métodos **pqcPublicKey**, o qual utiliza uma **rawJSONWebKey** e recupera a chave pós-quântica dessa estrutura. Assim como foi implementado o método **fromPQCPublicKey**, o qual realiza a operação inversa. Dessa mesma maneira, foram implementados os métodos **pqcPrivateKey** e **fromPQCPrivateKey** que lida da mesma forma, porém com chaves privadas.

O arquivo **shared.go** também sofreu alterações. Esse arquivo mantém estruturas e métodos que são compartilhados por diversos outros, como nome de algoritmos, erros, etc.

```
1     // Signature algorithms
2     const (
3         ...
4         Dilithium5                    =
              SignatureAlgorithm("Dilithium5")
5         Dilithium5AES                 =
              SignatureAlgorithm("Dilithium5-AES")
6         Falcon1024                    =
              SignatureAlgorithm("Falcon-1024")
7         ...
8         SphincsPlusHaraka128fSimple   =
              SignatureAlgorithm("SPHINCS+-Haraka-128f-Simple")
9         ...
10        SphincsPlusSHAKE256128fRobust =
              SignatureAlgorithm("SPHINCS+-SHAKE256-128f-Robust")
11    )
```

Figura 49 – Adições do arquivo shared.go

Fonte: Original

Foram adicionadas constantes para os algoritmos de assinatura pós-quântica

integrados e apenas isso foi necessário nesse arquivo, como pode ser visto na Figura 49.

E, por fim, o arquivo **signing.go** foi modificado. Esse é o principal arquivo que lida com a questão de assinaturas para dados em formato JSON, ilustrado na Figura 50.

```
1     // newVerifier creates a verifier based on the key type
2     func newVerifier(verificationKey interface{}) (payloadVerifier,
         error) {
3         switch verificationKey := verificationKey.(type) {
4         case *pqc.PublicKey:
5             return &pqcEncrypterVerifier{
6                 publicKey: verificationKey,
7             }, nil
8         ...
9         return nil, ErrUnsupportedKeyType
10    }
11
12    ...
13
14    func makeJWSRecipient(alg SignatureAlgorithm, signingKey
         interface{}) (recipientSigInfo, error) {
15        switch signingKey := signingKey.(type) {
16        case *pqc.PrivateKey:
17            return newPQCSigner(alg, signingKey)
18        ...
19        if signer, ok := signingKey.(OpaqueSigner); ok {
20            return newOpaqueSigner(alg, signer)
21        }
22        return recipientSigInfo{}, ErrUnsupportedKeyType
23    }
```

Figura 50 – Funções do arquivo signing.go

Fonte: Original

No método **newVerifier**, o qual cria um verificador baseado no tipo de chave, adicionou-se um caso para a chave pública pós-quântica integrada. O mesmo foi feito no método **makeJWSRecipient**, adicionando um caso para a chave privada pós-quântica.

4.4 PEBBLE

Pebble é um pequeno servidor ACME para testes, seu uso não se enquadra para uma AC a nível de produção. Porém, como ele é feito para testes, possui diversas facilidades nesse quesito. Ele provê interfaces simples para utilizar o servidor ACME, casos de teste para novas funcionalidades, tenta garantir, ao máximo, o uso dele

apenas para testes, etc.

Ou seja, o Pebble além de ser um servidor ACME também possui uma AC de testes para a emissão, renovação e revogação dos certificados. Dessa forma, as modificações realizadas se deram em relação as chaves dos certificados – tornando-os pós-quânticos – e em relação as mensagens JSON trocadas, dando suporte aos algoritmos pós-quânticos.

O primeiro arquivo modificado foi **ca.go** o qual trata da criação de uma AC e toda a cadeia de certificação, para prototipagem, demonstrado na Figura 51 e na Figura 52.

```
1  // makeKey creates a new PQC private key and a Subject Key Identifier
2  func makeRootKey() (*pqc.PrivateKey, []byte, error) {
3      key, err := pqc.GenerateKey("sphincs+-shake256-256f-robust")
4      if err != nil {
5          return nil, nil, err
6      }
7      ski, err := makeSubjectKeyID(key.Public())
8      if err != nil {
9          return nil, nil, err
10     }
11     return key, ski, nil
12 }
13
14 func makeInterKey() (*pqc.PrivateKey, []byte, error) {
15     key, err := pqc.GenerateKey("sphincs+-shake256-128f-robust")
16     if err != nil {
17         return nil, nil, err
18     }
19     ski, err := makeSubjectKeyID(key.Public())
20     if err != nil {
21         return nil, nil, err
22     }
23     return key, ski, nil
24 }
```

Figura 51 – Funções do arquivo ca.go

Fonte: Original

```
1     func (ca *CAImpl) GetRootKey(no int) *pqc.PrivateKey {
2         chain := ca.getChain(no)
3         if chain == nil {
4             return nil
5         }
6
7         switch key := chain.root.key.(type) {
8         case *pqc.PrivateKey:
9             return key
10        }
11        return nil
12    }
13
14    ...
15
16    func (ca *CAImpl) GetIntermediateKey(no int) *pqc.PrivateKey {
17        chain := ca.getChain(no)
18        if chain == nil {
19            return nil
20        }
21
22        switch key := chain.intermediates[0].key.(type) {
23        case *pqc.PrivateKey:
24            return key
25        }
26        return nil
27    }
```

Figura 52 – Funções do arquivo ca.go

Fonte: Original

O método **makeKey**, o qual gera uma chave e é utilizado em cada um dos certificados da cadeia, foi substituído por dois métodos: **makeRootKey** e **makeInterKey**, dessa forma, pode-se ter níveis diferentes de segurança dependendo do nível em que a AC se encontra. Além disso, esses métodos agora geram uma chave pós-quântica. Da mesma forma, alteraram-se os métodos **GetRootKey** e **GetIntermediateKey** para retornar uma chave pós-quântica.

Outro arquivo modificado foi **jose.go** o qual cria uma simples abstração para alguns métodos em cima da biblioteca Go-JOSE, como já visto e ilustrado na Figura 53.

```
1     func algorithmForKey(key *jose.JSONWebKey) (string, error) {
2         switch k := key.Key.(type) {
3         case *pqc.PublicKey:
4             switch k.AlgName {
5             case "dilithium5":
6                 return string(jose.Dilithium5), nil
7             ...
8             case "rainbow-i-classic":
9                 return string(jose.RainbowIClassic), nil
10            ...
11            case "sphincs+-shake256-128f-robust":
12                return string(jose.SphincsPlusSHAKE256128fRobust),
                      nil
13            }
14        ...
15        return "", fmt.Errorf("no signature algorithms suitable for
              given key type: %T", key.Key)
16    }
```

Figura 53 – Modificações do arquivo jose.go

Fonte: Original

No método **algorithmForKey**, o qual retorna o algoritmo da chave JSON que o assina, foram adicionados casos para os algoritmos pós-quânticos.

E, por fim, o último arquivo modificado foi **wfe.go** o qual lida com as possíveis ações do protocolo ACME.

```
1     ...
2     type keyGetter func(no int) *pqc.PrivateKey
3     ...
```

Figura 54 – Adicão do arquivo wfe.go

Fonte: Original

Nesse caso, apenas foi necessário alterar o tipo **keyGetter**, o qual é uma função, para retornar uma chave pós-quântica, como visto na Figura 54.

4.5 LEGO

LEGO – sigla para *Let's Encrypt client and ACME library written in Go* ou cliente Let's Encrypt e biblioteca ACME escrita em Go – é um cliente ACME completo. Ele não só é utilizado para prototipagem como também é feito para ambientes de produção completos com diversas funcionalidades.

Ele é baseado na RFC8555 (BARNES *et al.*, 2019) da versão 2 do protocolo ACME e com esse cliente é possível: se registrar a uma AC, obter certificados do zero –

criar par de chaves, autenticar, etc – ou obter certificados a partir de uma CSR, renovar certificados, revogar certificados, entre outras diversas funcionalidades.

Por ele ser escrito na linguagem Go, ou seja, ser compatível com o Pebble para integrar as modificações mais facilmente e por ele ser muito utilizado foi escolhido como o melhor cliente para integrar ao trabalho.

A implementação no cliente LEGO pode ser dividida em duas principais partes: modificações em pacotes de mais baixo nível – que lidam com criptografia e com mensagens JSON assinadas digitalmente – e modificações nas configurações do cliente, como parâmetros para qual algoritmo será utilizado.

Primeiramente, modificou-se a parte mais baixo nível do cliente LEGO, dessa forma, começando com o arquivo **jws.go** que lida com JSON Web Signature (JONES; BRADLEY; SAKIMURA, 2015a) que foi modificado para realizar a assinatura de dados JSON com os algoritmos pós-quânticos.

```
1     // SignContent Signs a content with the JWS.
2     func (j *JWS) SignContent(url string, content []byte)
         (*jose.JSONWebSignature, error) {
3         var alg jose.SignatureAlgorithm
4         switch k := j.privKey.(type) {
5         case *pqc.PrivateKey:
6             switch k.AlgName {
7             case "dilithium5":
8                 alg = jose.Dilithium5
9             ...
10            case "rainbow-v-circumzenithal":
11                alg = jose.RainbowVCircumzenithal
12            ...
13            case "dilithium2":
14                alg = jose.Dilithium2
15            ...
16            case "sphincs+-shake256-128f-robust":
17                alg = jose.SphincsPlusSHAKE256128fRobust
18            }
19        ...
20        return signed, nil
21    }
```

Figura 55 – Modificação do arquivo jws.go

Fonte: Original

No método **SignContent** adicionou-se um caso para a chave privada pós-quântica e, para este caso, um para cada algoritmo pós-quântico integrado, como pode ser visto na Figura 55.

Outro arquivo modificado nessa etapa foi **crypto.go** o qual trata da criação e manipulação de chaves criptográficas e de arquivos formatados – principalmente em

formatos muito utilizados em certificados digitais e chaves criptográficas, como o PEM (JOSEFSSON; LEONARD, 2015).

```
// Constants for all key types we support.
const (
    ...
    Dilithium5                    = KeyType("dilithium5")
    ...
    Falcon512                     = KeyType("falcon-512")
    ...
    SphincsPlusSHAKE256128fRobust =
       KeyType("sphincs+-shake256-128f-robust")
)

...

func ParsePEMPrivateKey(key []byte) (crypto.PrivateKey, error) {
    keyBlockDER, _ := pem.Decode(key)

    if keyBlockDER.Type != "PRIVATE KEY" &&
       !strings.HasSuffix(keyBlockDER.Type, " PRIVATE KEY") {
        return nil, fmt.Errorf("unknown PEM header %q",
           keyBlockDER.Type)
    }

    ...

    if key, err := x509.ParsePKCS8PrivateKey(keyBlockDER.Bytes);
       err == nil {
        switch key := key.(type) {
        case *rsa.PrivateKey, *ecdsa.PrivateKey,
           ed25519.PrivateKey, *pqc.PrivateKey:
            return key, nil
        default:
            return nil, fmt.Errorf("found unknown private key
               type in PKCS#8 wrapping: %T", key)
        }
    }

    ...
}
```

Figura 56 – Modificações do arquivo crypto.go

Fonte: Original

```
1     func GeneratePrivateKey(keyType KeyType) (crypto.PrivateKey,
          error) {
2         switch keyType {
3         ...
4         case Dilithium5:
5             return pqc.GenerateKey("dilithium5")
6         ...
7         case SphincsPlusHaraka256sRobust:
8             return pqc.GenerateKey("sphincs+-haraka-256s-robust")
9         ...
10        case Falcon512:
11            return pqc.GenerateKey("falcon-512")
12        ...
13        case SphincsPlusSHAKE256128fRobust:
14            return pqc.GenerateKey("sphincs+-shake256-128f-robust")
15        }
16
17        return nil, fmt.Errorf("invalid KeyType: %s", keyType)
18    }
```

Figura 57 – Modificações do arquivo crypto.go

Fonte: Original

Neste arquivo – Figura 56 e 57 – foram adicionadas constantes para os algoritmos pós-quânticos integrados com seus respectivos nomes da LibOQS. No método **ParsePEMPrivateKey**, o qual utiliza um vetor de *bytes* que representa a chave privada em formato PEM e transforma para a estrutura de chave privada que criada, foi adicionado o caso para a chave privada pós-quântica. Da mesma forma, foi feito no método **GeneratePrivateKey**, o qual retorna a estrutura de chave privada com base no tipo de chave passado por parâmetro, dessa forma, adicionaram-se os algoritmos que integrados como novos casos.

Já na segunda parte, onde modificaram-se arquivos de configuração do cliente LEGO, foi alterado, primeiramente, o arquivo **setup.go** o qual cria um ambiente de inicialização do sistema LEGO.

```
1     // getKeyType the type from which private keys should be
          generated.
2     func getKeyType(ctx *cli.Context) certcrypto.KeyType {
3         keyType := ctx.GlobalString("key-type")
4         switch strings.ToUpper(keyType) {
5         ...
6         case "DILITHIUM5":
7             return certcrypto.Dilithium5
8         ...
9         ...
10        case "SPHINCS+-SHAKE256-128F-ROBUST":
11            return certcrypto.SphincsPlusSHAKE256128fRobust
12        }
13
14        log.Fatalf("Unsupported KeyType: %s", keyType)
15        return ""
16    }
```

Figura 58 – Modificações do arquivo setup.go

Fonte: Original

Nesse arquivo (Figura 58) apenas foi adicionado casos para os novos algoritmos pós-quânticos integrados no método **getKeyType**, para que o algoritmo de chave passado como parâmetro pela linha de comando tenha essas novas opções disponíveis.

E, finalmente, o arquivo **client_config.go** o qual cria configurações padrão, as quais podem ser alteradas passando diferentes parâmetros de inicialização, do lado cliente, ou seja, um ambiente padrão caso não sejam passados certos parâmetros, como pode ser visto na Figura 59.

```
1     func NewConfig(user registration.User) *Config {
2         return &Config{
3             CADirURL:   LEDirectoryProduction,
4             User:       user,
5             HTTPClient: createDefaultHTTPClient(),
6             Certificate: CertificateConfig{
7                 KeyType: certcrypto.Falcon1024,
8                 Timeout: 30 * time.Second,
9             },
10        }
11    }
```

Figura 59 – Modificações do arquivo client_config.go

Fonte: Original

No método **NewConfig** foi alterado o **KeyType** para um tipo específico pós-quântico. Porém, esse tipo é apenas usado caso não seja explicitamente dito, por

parâmetro, outro tipo. Essa modificação não é necessária, porém para um ambiente de desenvolvimento onde o algoritmo pós-quântico é irrelevante – desde que seja pós-quântico – pode facilitar o processo.

5 RESULTADOS

Realizaram-se testes para diversas funcionalidades do protocolo ACME como é detalhado em cada teste a seguir. Os testes tem como principais objetivos comparar o desempenho dos algortimos integrados assim como comparar com os algoritmos clássicos.

Para os algoritmos, foram testados todos os que foram integrados, como explicitado anteriormente neste trabalho. Já para os clássicos foram testados os algoritmos **RSA** – com a variação RSA8192 e RSA2048 – e **ECDSA** – com a variação EC384 e EC256. Escolheram-se esses dois algoritmos clássicos, pois o algoritmo RSA é o algoritmo padrão do cliente LEGO, ou seja, se não for pedido a utilização de outro algoritmo para a criação do par de chaves, este será utilizado. E o algoritmo ECDSA, pois é o principal algoritmo utilizado quando se trata de curvas elípitcas.

Para cada um dos testes realizados, mediu-se o tempo para realizar a operação (em nanosegundos), a quantidade de bytes alocados para a operação e a quantidade de alocações realizadas. Cada tabela, foi o resultado de 1000 iterações dos testes, obtendo seus valores médios. Esses testes foram baseados nos próprios testes do cliente LEGO, no entanto resultando em mais estatísticas e, também, integrando os algoritmos pós-quânticos.

5.1 CONFIGURAÇÃO DO AMBIENTE

Todos os testes realizados foram num ambiente local – na mesma máquina – com o intuito de simplicidade e isolamento dos aspectos de rede, como latência e perda de pacotes.

Além disso, todos os testes foram baseados nos testes do cliente LEGO – como mencionado. Isso se refere aos testes de unidade que já existem no próprio código LEGO (LEGO. . . , 2021). Porém, algumas modificações foram realizadas, como a mudança de testes de validação (os quais apenas testas se certa funcionalidade se comporta como esperado) para testes de *benchmark* que trazem informações de desemepenho consigo. É possível ver também que cada tabela de resultados a seguir possui um nome, como *BenchmarkCertificateService_Get*, esse nome se refere ao mesmo nome do teste original do LEGO – para manter o mesmo padrão de nomenclatura. Em relação às métricas medidas, tem-se que o tempo de execução se refere a todo o tempo necessário para a execução do teste que pode incluir criação de mensagens, chamadas de funções, etc. Já a quantidade de bytes alocados se refere à quantidade de bytes alocados por teste, ou seja, quantos bytes são alocados em memória por teste. E, por fim, a quantidade de alocações se refere a quantas vezes foi necessário realizar uma alocação na memória por teste, nota-se que é similar, porém diferente da métrica anterior a qual mede a quantidade de bytes alocados e não

quantas alocações foram necessárias para alocar todos esses bytes.

Para a realização dos testes foi utilizada apenas uma única máquina com a seguinte configuração relevante para os testes:

- Processador: AMD Ryzen 7 3800X 8-Core (16 CPUs), 3.9GHz
- Memória: 16GB
- Sistema Operacional: Ubuntu 21.10 64-bit

Além disso, dentre todas os testes LEGO, foram medidos resultados do seguinte conjunto:

- BenchmarkCertificateService_Get: Recuperação do certificado digital
- BenchmarkOrderService_New: Requisição de um novo certificado (*newOrder*)
- BenchmarkGeneratePrivateKey: Geração do par de chaves criptográficas
- BenchmarkGenerateCSR: Geração da CSR
- BenchmarkPEMEncode: Codificação das chaves criptográficas para o formato PEM
- BenchmarkParsePEMCertificate: Conversão do certificado digital em formato PEM para o código Go.
- BenchmarkChallenge: Realização do desafio HTTP

Dessa forma, as principais operações do protocolo ACME são testadas – as operações necessárias para emissão de um certificado digital – e podem ser vistas em mais detalhes em cada um dos testes a seguir.

5.2 BENCHMARKS

Um dos passos finais do protocolo ACME e um dos seus principais objetivos é a emissão de um certificado digital. Neste teste é medido o desempenho de recuperar um certificado com a utilização dos algoritmos abaixo na Tabela 1.

	tempo (ns)	**bytes**	**alocações**
RSA8192	10155533507	10017952	9939
RSA2048	257552848	4234304	11273
EC384	6855677	3507464	28697
EC256	742294	96832	662
dilithium5	1121193	250096	586
dilithium5-aes	969369	246320	571
falcon-1024	21238404	187016	583
rainbow-V-classic	4674049033	76479616	594
rainbow-V-circumzenithal	5109131929	22412088	590
rainbow-V-compressed	7338125208	20998400	600
sphincs+-haraka-256s-simple	157437177	377536	598
sphincs+-haraka-256f-simple	15808855	592256	575
sphincs+-haraka-256s-robust	192191153	382464	580
sphincs+-haraka-256f-robust	19806326	594752	581
sphincs+-sha256-256s-simple	336648316	384992	589
sphincs+-sha256-256f-simple	43610094	596192	591
sphincs+-sha256-256s-robust	1176402460	376832	585
sphincs+-sha256-256f-robust	130547908	603904	592
sphincs+-shake256-256s-simple	799759836	368768	595
sphincs+-shake256-256f-simple	81996421	590624	576
sphincs+-shake256-256s-robust	1384110987	376096	585
sphincs+-shake256-256f-robust	149463654	603200	584
dilithium2	848558	172936	582
dilithium2-aes	729076	183896	593
falcon-512	5681192	126744	579
rainbow-I-classic	617434409	6620920	592
rainbow-I-circumzenithal	123674375	2639400	589
rainbow-I-compressed	174787641	2540872	587
sphincs+-haraka-128s-simple	92914671	173584	580
sphincs+-haraka-128f-simple	5451254	268648	581
sphincs+-haraka-128s-robust	107349266	167088	573
sphincs+-haraka-128f-robust	5911755	261152	589
sphincs+-sha256-128s-simple	242568567	171616	565
sphincs+-sha256-128f-simple	13893869	267680	582
sphincs+-sha256-128s-robust	432991367	176080	575
sphincs+-sha256-128f-robust	23687277	262040	573
sphincs+-shake256-128s-simple	516400456	177568	581
sphincs+-shake256-128f-simple	27689152	247744	568
sphincs+-shake256-128s-robust	924943749	169992	587
sphincs+-shake256-128f-robust	48245022	268616	580

Tabela 1 – Resultados BenchmarkCertificateService_Get

Fonte: Original

Primeiramente, nota-se que o número de alocações dos algoritmos pós-quânticos é bem estável, com pouca variações entre os algoritmos, e bem abaixo do número de

alocações que os algoritmos clássicos executaram.

Além disso, em relação ao número de bytes alocados, percebe-se que o algoritmo Falcon teve um desempenho excelente, tendo número bem abaixo dos outros candidatos. Em seguida, o algoritmo Dilithium, junto com algumas variações do algoritmo SPHINCS+ também obtiveram ótimos desempenhos, assemelhando-se ao Falcon. Já o algoritmo Rainbow teve um número de bytes alocados pelo menos a uma ordem de grandeza acima, sendo pior nesse quesito. Os algoritmos clássicos, por sua vez, desempenharam de formas bem variadas também, Com resultados muito bons, como do algoritmo EC256, resultados intermediários como de EC384 e RSA2048, e resultados um pouco piores como do algoritmo RSA8192 – porém, com quantidade de bytes alocados abaixo do algoritmo Rainbow ainda.

E, por fim, o tempo de execução teve como seu melhor desemepenhante o algoritmo Dilithium, com uma ordem de grande abaixo do próximo candidato, o algoritmo Falcon e algumas variações do algoritmo SPHINCS+ – o qual, teve variações que desempenharam muito bem e outras com tempos bem acima, possuindo o melhor e o pior tempo no menor nível de segurança. O algoritmo Rainbow, obteve tempos maiores na sua versão comprimida em relação ao maior nível de segurança, com tempos um pouco menores na versão *circumzenithal* e tempos menores ainda na versão clássica, o que surpreende, visto que a complexidade seria inversa a essa ordem de desemepenho. Porém, ainda assim esse nível de segurança desempenhou pior que os outros algoritmos. Já no menor nível de segurança, a versão clássica obteve piores tempos, seguida da comprimida e da *circumzenithal* logo depois. Em relação ao algoritmos clássicos, percebe-se que o RSA8192 obteve o pior tempo de todos os candidatos. Os outros algoritmos clássicos tiveram tempos similares, como o RSA2048 e o EC384 que assemelharam-se a algumas variações do SPHINCS+. O algortimo EC256 obteve tempos muito bons também, assemelhando-se ao algoritmo Dilithium.

Um dos primeiros passos para a emissão de um certificado utilizando o protocolo ACME é o pedido de newOrder – como explicado em seções anteriores. Neste teste é medido o desempenho de cada algoritmo na realização dessa etapa para assinar as mensagens, como pode ser visto na Tabela 2.

	tempo (ns)	**bytes**	**alocações**
RSA8192	96677936517	97570440	91730
RSA2048	279057929	4986416	13266
EC384	17973537	6949448	57257
EC256	777645	93912	886
dilithium5	1571755	427808	778
dilithium5-aes	1380358	419808	778
falcon-1024	22030118	267640	753
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	157567926	778768	794
sphincs+-haraka-256f-simple	18161793	1316216	792
sphincs+-haraka-256s-robust	195167473	763008	765
sphincs+-haraka-256f-robust	21996401	1317408	780
sphincs+-sha256-256s-simple	340909259	766984	779
sphincs+-sha256-256f-simple	47974207	1292672	771
sphincs+-sha256-256s-robust	1185136692	754944	755
sphincs+-sha256-256f-robust	137486203	1318352	793
sphincs+-shake256-256s-simple	808838800	783080	781
sphincs+-shake256-256f-simple	89750627	1307088	777
sphincs+-shake256-256s-robust	1349992211	782176	780
sphincs+-shake256-256f-robust	160205543	1316424	793
dilithium2	1215625	263632	769
dilithium2-aes	1235105	260280	777
falcon-512	6645092	172936	762
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	93732496	270736	766
sphincs+-haraka-128f-simple	6128456	518816	765
sphincs+-haraka-128s-robust	106527981	283592	785
sphincs+-haraka-128f-robust	7010889	516528	775
sphincs+-sha256-128s-simple	243659533	291368	781
sphincs+-sha256-128f-simple	16282851	504880	766
sphincs+-sha256-128s-robust	449395053	290880	778
sphincs+-sha256-128f-robust	28265190	507312	775
sphincs+-shake256-128s-simple	521267119	292208	786
sphincs+-shake256-128f-simple	32097843	522368	768
sphincs+-shake256-128s-robust	974057123	291096	771
sphincs+-shake256-128f-robust	57236346	515000	776

Tabela 2 – Resultados BenchmarkOrderService_New

Fonte: Original

Primeiramente, nota-se que o algoritmo Rainbow não obteve resultados nesse teste. Isso se deve à interação do algoritmo com o cliente LEGO e a biblioteca de

Benchmark, onde o número de bytes alocados deve ter sido superior à algum limite, resultando em erro na obtenção de resultados.

Na questão do número de alocações, os algoritmos pós-quânticos tiveram resultados muito bons e consistentes ao longo de todos eles, mantendo números muito similares. Já os clássicos, tiveram números muito acima dos pós-quânticos, em média.

Em relação aos número de bytes alocados, o algoritmo Falcon obteve resultados ótimos, desempenhando melhor que os outros candidatos. O algoritmo Dilithium não ficou muito longe desses números, obtendo resultados muito bons também nessa questão. Por mais que o algoritmo SPHINCS+ tenha desempenhado pior nesse quesito, em alguns casos nota-se que seus resulados foram muito similares aos do Dilithium. Em comparação aos clássicos, os pós-quânticos tiveram resultados melhores na sua maioria.

Por fim, na questão de tempo de execução, o algoritmo Dilithium foi o que obteve melhor desempenho, tendo números muito abaixo dos de mais. Em seguida, o algoritmo SPHINCS+ teve melhores resultados nos casos de variações específicas. Porém em alguns casos do algoritmo SPHINCS+, o mesmo obteve resultados piores que os outros candidatos. Já o algoritmo Falcon, ficou no meio termo nesse parâmetro, assemelhando-se a algumas variações do SPHINCS+. Os algoritmos clássicos, com exceção do RSA8192 que teve tempos de execução muito acima dos de mais, tiveram resultados bem variados, ou seja, alguns com tempos bem abaixo como EC256, e outros com tempos mais altos, como RSA2048.

Um dos primeiros passos do protocolo ACME está relacionado a geração do par de chaves – em especial da chave privada a qual é utilizada para a assinatura digital – por meio do cliente ACME. O próximo teste mede justamente os parâmetros de cada algoritmo durante essa etapa de geração de par de chaves criptográficas, com os resultados na Tabela 3.

	tempo (ns)	**bytes**	**alocações**
RSA8192	42013866457	42268552	39519
RSA2048	249897259	4024784	10367
EC384	2979343	1727472	14277
EC256	38542	1816	16
dilithium5	126531	7808	5
dilithium5-aes	58165	7808	5
falcon-1024	14171837	5000	6
rainbow-V-classic	3961783928	3342616	5
rainbow-V-circumzenithal	4642912712	1949976	5
rainbow-V-compressed	4548094094	541016	5
sphincs+-haraka-256s-simple	10424595	464	5
sphincs+-haraka-256f-simple	653786	464	5
sphincs+-haraka-256s-robust	12685489	464	5
sphincs+-haraka-256f-robust	813350	464	5
sphincs+-sha256-256s-simple	24268373	464	5
sphincs+-sha256-256f-simple	1747093	464	5
sphincs+-sha256-256s-robust	94862236	464	5
sphincs+-sha256-256f-robust	5846644	464	5
sphincs+-shake256-256s-simple	59927736	464	5
sphincs+-shake256-256f-simple	3458569	464	5
sphincs+-shake256-256s-robust	102339587	464	5
sphincs+-shake256-256f-robust	6588463	464	5
dilithium2	56424	4352	5
dilithium2-aes	34273	4352	5
falcon-512	4789825	2688	5
rainbow-I-classic	103924045	270616	5
rainbow-I-circumzenithal	118922792	172312	5
rainbow-I-compressed	118168160	65880	5
sphincs+-haraka-128s-simple	10050678	368	5
sphincs+-haraka-128f-simple	196021	368	5
sphincs+-haraka-128s-robust	11704011	368	5
sphincs+-haraka-128f-robust	199918	368	5
sphincs+-sha256-128s-simple	27526821	368	5
sphincs+-sha256-128f-simple	583804	368	5
sphincs+-sha256-128s-robust	49992281	368	5
sphincs+-sha256-128f-robust	794592	368	5
sphincs+-shake256-128s-simple	58460269	368	5
sphincs+-shake256-128f-simple	964555	368	5
sphincs+-shake256-128s-robust	106019910	368	5
sphincs+-shake256-128f-robust	1757778	368	5

Tabela 3 – Resultados BenchmarkGeneratePrivateKey

Fonte: Original

Primeiramente, nota-se que o número de alocações dos algoritmos pós-quânticos se manteve igual. Isso se deve, provavelmente, pelo de fato de todos serem implemen-

tados pela mesma biblioteca, compartilhando funcionalidades em comum. Onde os algoritmos clássicos possuem ordens de grandeza maiores nessa questão.

Em relação à quantidade de bytes alocados, o algoritmo SPHINCS+ teve uma quantidade muito abaixo tanto dos algoritmos pós-quânticos quanto dos clássicos. O algoritmo Falcon já supera em uma ordem de grandeza nesse aspecto e o algoritmo Dilithium com números um pouco maiores, porém não se compara à diferença dos anteriores. Além disso, nota-se que o algoritmo Rainbow ficou muito acima dos outros pós-quânticos em relação aos bytes alocados. Percebe-se também que os algoritmos pós-quânticos tiveram resultados melhores nesse aspecto em relação aos clássicos, com algumas exceções. Nota-se também que o algoritmo EC256 teve números muito abaixo dos outros clássicos.

Na questão do tempo de execução, percebe-se que o algoritmo RSA teve os piores números em relação à todos os algoritmos. Entre os pós-quânticos, o algoritmo Dilithium obteve os melhores resultados de tempo, equanto o algoritmo Falcon e o algoritmo SPHINCS+ se mantevem num meio termo, onde o algoritmo Rainbow teve números bem acima dos outros candidatos pós-quânticos.

Um dos passos mais importantes do protocolo ACME é a geração da CSR por meio do cliente ACME, a qual, posteriormente, se tornará um certificado digital. Este seguinte teste mede os parâmetros para a geração de uma CSR utilizando cada um dos algoritmos – tanto para a geração do certificado quanto para a assinatura da mensagem que a contém, como pode ser visto na Tabela 4.

	tempo (ns)	**bytes**	**alocações**
RSA8192	66684832	193936	378
RSA2048	1441589	38536	299
EC384	3070452	1743032	14528
EC256	104447	9696	220
dilithium5	190490	20352	163
dilithium5-aes	107317	20352	163
falcon-1024	492916	11264	163
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	143949408	70304	163
sphincs+-haraka-256f-simple	13705182	119456	163
sphincs+-haraka-256s-robust	176844016	70304	163
sphincs+-haraka-256f-robust	17615438	119456	163
sphincs+-sha256-256s-simple	304702438	70288	163
sphincs+-sha256-256f-simple	39679180	119456	163
sphincs+-sha256-256s-robust	1078656368	70304	163
sphincs+-sha256-256f-robust	120294655	119456	163
sphincs+-shake256-256s-simple	704655643	70304	163
sphincs+-shake256-256f-simple	76637610	119456	163
sphincs+-shake256-256s-robust	1225975063	70304	163
sphincs+-shake256-256f-robust	136095174	119456	163
dilithium2	187888	12800	163
dilithium2-aes	93538	12800	163
falcon-512	277770	8128	163
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	81166243	21104	163
sphincs+-haraka-128f-simple	4486391	41584	163
sphincs+-haraka-128s-robust	94119698	21104	163
sphincs+-haraka-128f-robust	4887491	41584	163
sphincs+-sha256-128s-simple	216523901	21104	163
sphincs+-sha256-128f-simple	12570565	41584	163
sphincs+-sha256-128s-robust	382643459	21104	163
sphincs+-sha256-128f-robust	22361670	41584	163
sphincs+-shake256-128s-simple	479108026	21104	163
sphincs+-shake256-128f-simple	25613943	41584	163
sphincs+-shake256-128s-robust	840072674	21104	163
sphincs+-shake256-128f-robust	48709339	41584	163

Tabela 4 – Resultados BenchmarkGenerateCSR

Fonte: Original

Nota-se, primeiramente, que o algoritmo Rainbow não teve resultados. Isso se deve, pois ele gera CSRs e certificados digitais muito grandes os quais não cabem nos

parâmetros padrão do protocolo HTTP, não sendo possível enviá-la sem alterações no protocolo HTTP. Percebe-se novamente, também, o mesmo valor para o número de alocações nos algoritmos pós-quânticos pelos mesmos motivos já mencionados. Entre os algoritmos clássicos o RSA teve um ótimo desempenho nesse quesito.

Em relação ao número de bytes alocados, o algoritmo SPHINCS+ ficou acima dos outros algoritmos pós-quânticos, porém de maneira consistente, com pouca variação. Os algortimos Dilithium e Falcon obtiveram números similares entre si, porém com o algoritmo Falcon tendo melhores resultados neste parâmetro. Já os algoritmos clássicos tiveram números bem variados nesse quesito, tendo resultados comparáveis aos pós-quânticos e resultados muito inferiores ao mesmo tempo.

Na questão do tempo de execução, o algoritmo Dilithium obteve tempos impressionantes, sendo bem abaixo dos outros candidatos pós-quânticos e até dos algoritmos clássicos. O algoritmo Falcon também teve bons tempos, porém um pouco acima do anterior. Já o algoritmo SPHINCS+ teve uma certa consistência nos números, porém com tempos muito acima dos demais. Em alguns casos obteve tempos acima até do RSA8192 que geralmente tem os piores desempenhos nesses testes.

Outro aspecto importante, não apenas para o protocolo ACME, mas para várias aplicações criptográficas também, é a codificação no formato PEM. Neste seguinte teste é medido o desempenho da codificação de chaves criptográficas, com os resultados na Tabela 5.

	tempo (ns)	**bytes**	**alocações**
RSA8192	8260881523	7955088	7590
RSA2048	95381935	1480416	4112
EC384	2800559	1726008	14268
EC256	47720	5160	64
dilithium5	139771	50864	58
dilithium5-aes	77488	50856	57
falcon-1024	13934361	28328	56
rainbow-V-classic	3904954233	3344680	33
rainbow-V-circumzenithal	4537623695	1952040	33
rainbow-V-compressed	4583668947	543080	33
sphincs+-haraka-256s-simple	10498036	3800	52
sphincs+-haraka-256f-simple	666729	3800	52
sphincs+-haraka-256s-robust	12979960	3800	52
sphincs+-haraka-256f-robust	835164	3800	52
sphincs+-sha256-256s-simple	24183459	3800	52
sphincs+-sha256-256f-simple	1755658	3800	52
sphincs+-sha256-256s-robust	93613971	3800	52
sphincs+-sha256-256f-robust	5809053	3800	52
sphincs+-shake256-256s-simple	59246639	3800	52
sphincs+-shake256-256f-simple	3604395	3800	52
sphincs+-shake256-256s-robust	105818320	3800	52
sphincs+-shake256-256f-robust	6620277	3800	52
dilithium2	100494	26920	56
dilithium2-aes	54559	26920	56
falcon-512	4426888	15272	55
rainbow-I-classic	105650448	272680	33
rainbow-I-circumzenithal	118303789	174376	33
rainbow-I-compressed	117377480	67944	33
sphincs+-haraka-128s-simple	10377752	3576	52
sphincs+-haraka-128f-simple	218727	3576	52
sphincs+-haraka-128s-robust	12477908	3576	52
sphincs+-haraka-128f-robust	213318	3576	52
sphincs+-sha256-128s-simple	27787232	3576	52
sphincs+-sha256-128f-simple	468064	3576	52
sphincs+-sha256-128s-robust	50703309	3576	52
sphincs+-sha256-128f-robust	799354	3576	52
sphincs+-shake256-128s-simple	60598335	3576	52
sphincs+-shake256-128f-simple	988694	3576	52
sphincs+-shake256-128s-robust	112837474	3576	52
sphincs+-shake256-128f-robust	1777461	3576	52

Tabela 5 – Resultados BenchmarkPEMEncode

Fonte: Original

Primeiramente, no número de alocações percebe-se o baixo e estável número dos algortimos pós-quânticos, diferentemente dos clássicos, com números muito acima.

Na questão de quantidade de bytes alocados, nota-se a estabilidade e um um ótimo desempenho do algoritmo SPHINCS+ – baixo número de bytes alocados durante a operação. O algoritmo Rainbow possui um desempenho ruim neste teste, tendo resultados similares aos algoritmos clássicos. Enquanto isso, os algoritmos Dilithium e Falcon permanecem no meio termo, tendo resultados muito parecidos entre si.

Em relação ao tempo de execução da codificação, o algoritmo Dilithium demonstra um ótimo desempenho, possuindo um tempo bem abaixo dos outros candidatos. Os algoritmos Falcon e SPHINCS+, ficam no meio termo, enquanto o algoritmo Rainbow possui resultados muito ruins nesse quesito. Em comparação aos algoritmos clássicos, os pós-quânticos tiveram resultados similares na média, com exceção do RSA8192, que tem possui um tempo de execução muito acima de todos os outros – clássicos e pós-quânticos.

Outro ponto importante, tanto para o protocolo ACME quanto para outras aplicações criptográficas, é a capacidade de converter um arquivo em formato PEM para sua representação em código. Neste teste é medido o desempenho da conversão de certificados digitais no formato PEM para sua representação no cliente LEGO, como pode ser visto na Tabela 6.

	tempo (ns)	**bytes**	**alocações**
RSA8192	42067829555	42451064	40093
RSA2048	145540114	2169384	6162
EC384	12372332	7050216	58482
EC256	231626	23720	470
dilithium5	468058	78176	380
dilithium5-aes	284484	78104	378
falcon-1024	16621633	44928	379
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	155735227	243360	382
sphincs+-haraka-256f-simple	15302718	448096	382
sphincs+-haraka-256s-robust	190877647	243360	382
sphincs+-haraka-256f-robust	19550675	448160	383
sphincs+-sha256-256s-simple	329667674	243256	380
sphincs+-sha256-256f-simple	44774934	448088	381
sphincs+-sha256-256s-robust	1175953080	243360	382
sphincs+-sha256-256f-robust	135193928	448088	381
sphincs+-shake256-256s-simple	782659983	243360	382
sphincs+-shake256-256f-simple	86942822	448088	381
sphincs+-shake256-256s-robust	1321532719	243288	380
sphincs+-shake256-256f-robust	154539119	448088	381
dilithium2	236493	46360	377
dilithium2-aes	205653	46360	377
falcon-512	4856045	29768	379
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	91896425	72464	380
sphincs+-haraka-128f-simple	5011414	134600	379
sphincs+-haraka-128s-robust	103997347	72392	378
sphincs+-haraka-128f-robust	5400802	134672	381
sphincs+-sha256-128s-simple	237760159	72464	380
sphincs+-sha256-128f-simple	15264743	134672	381
sphincs+-sha256-128s-robust	430486768	72464	380
sphincs+-sha256-128f-robust	27404773	134672	381
sphincs+-shake256-128s-simple	514253342	72464	380
sphincs+-shake256-128f-simple	31012989	134672	381
sphincs+-shake256-128s-robust	972879596	72464	380
sphincs+-shake256-128f-robust	55752609	134600	379

Tabela 6 – Resultados BenchmarkParsePEMCertificate

Fonte: Original

Primeiramente, nota-se a falta de resultados do algoritmo Rainbow. Isso se deve ao fato de que seus certificados possuem tamanhos muito acima dos demais e resultam

em problemas durante a execução de *benchmarks* na linguagem Go.

Em questões do número de alocações, os algoritmos clássicos realizaram um número muito maior enquanto os pós-quânticos estiveram ordens de grandeza abaixo nesse quesito.

Em relação ao número de bytes alocados, tem-se que o algoritmo Falcon obteve melhor desempenho, porém tendo resultados similares ao algoritmo Dilithium. Enquanto o algoritmo SPHINCS+ ficou um pouco acima nesse quesito, com algumas de suas variações a uma ordem de grandeza acima dos demais. Os algoritmos clássicos desempenharam, em média, pior que os pós-quânticos com exceção do algoritmo EC256.

Já na questão do tempo de execução, o algoritmo Dilithium foi muito melhor que os outros candidatos pós-quânticos e com resultados similares ao algoritmo EC do grupo dos clássicos. Os algoritmos Falcon e SPHINCS+ tiveram resultados similares, porém com tempos muito acima do Dilithium, com resultados semelhantes aos clássicos em média, com exceção do RSA8192 que teve um tempo muito acima dos outros.

Uma das principais etapas de autenticação do protocolo ACME é a realização do desafio ACME, onde o cliente demonstra o controle sob certo domínio. No seguinte teste é medido o desempenho da realização do desafio HTTP – como explicado em seções anteriores – utilizando cada um dos algoritmos para assinar as mensagens, como pode ser visto na Tabela 7.

	tempo (ns)	**bytes**	**alocações**
RSA8192	18454799487	17560720	16896
RSA2048	171146322	2820648	7707
EC384	4062628	1774040	14553
EC256	587367	72048	504
dilithium5	931623	100296	490
dilithium5-aes	787023	97936	498
falcon-1024	16847569	84872	481
rainbow-V-classic	4004382810	16322608	510
rainbow-V-circumzenithal	4524151349	5632096	494
rainbow-V-compressed	4525478996	4232768	509
sphincs+-haraka-256s-simple	11756282	80840	491
sphincs+-haraka-256f-simple	1239946	79128	490
sphincs+-haraka-256s-robust	13526494	66328	488
sphincs+-haraka-256f-robust	1542308	81224	497
sphincs+-sha256-256s-simple	25317550	78512	489
sphincs+-sha256-256f-simple	2368252	77968	492
sphincs+-sha256-256s-robust	98091443	61944	480
sphincs+-sha256-256f-robust	6262785	69896	479
sphincs+-shake256-256s-simple	60834800	72104	481
sphincs+-shake256-256f-simple	4013196	71064	495
sphincs+-shake256-256s-robust	102447339	80328	492
sphincs+-shake256-256f-robust	7077653	70024	481
dilithium2	753593	84440	488
dilithium2-aes	480571	70328	470
falcon-512	11994110	81816	495
rainbow-I-classic	106448431	1442568	485
rainbow-I-circumzenithal	118871755	658128	486
rainbow-I-compressed	117957904	550632	484
sphincs+-haraka-128s-simple	11241893	72160	492
sphincs+-haraka-128f-simple	883957	72072	488
sphincs+-haraka-128s-robust	12684016	78208	489
sphincs+-haraka-128f-robust	772967	73344	498
sphincs+-sha256-128s-simple	28730156	69672	483
sphincs+-sha256-128f-simple	1010456	64168	493
sphincs+-sha256-128s-robust	53483761	71488	493
sphincs+-sha256-128f-robust	1449081	72616	499
sphincs+-shake256-128s-simple	58646592	78520	492
sphincs+-shake256-128f-simple	1475393	59584	479
sphincs+-shake256-128s-robust	112387053	65752	469
sphincs+-shake256-128f-robust	2451097	80744	497

Tabela 7 – Resultados BenchmarkChallenge

Fonte: Original

Novamente, o número de alocações para os algoritmos pós-quânticos se manteve similar e bem abaixo do clássicos, como explicado anteriormente.

Em relação ao número de bytes alocados, percebe-se que o algoritmo SPHINCS+ se saiu melhor em alguns casos, porém, tanto ele quanto os algoritmos Dilithium e Falcon tiveram resultados muito similares. Já o algoritmo Rainbow teve um número de bytes alocados consideravelmente superior, assemelhando-se aos algoritmos clássicos em alguns casos.

Na questão do tempo de execução, o algoritmo Dilithium obteve resultados muito bons em relação aos outros candidatos, ficando ordens de grandeza abaixo. Em seguida, o algoritmo SPHINCS+, em alguns casos, obteve resultados muito bons, apesar de outros se assemelharem muito com os resultados do algoritmo Falcon. Já o algoritmo Rainbow obteve tempos de execução muito acima dos demais, resultando num pior desemepenho nesse quesito. Comparando com o desempenho dos algoritmos clássicos, os pós-quânticos tiveram resultados muito similares em média, com exceção do RSA8192 que teve tempos de execução muito superiores.

Por uma ótica geral, pode-se dizer que entre os algoritmos pós-quânticos o algoritmo Dilithium obteve ótimos resultados na maioria dos testes. Da mesma forma, os algoritmos Falcon e SPHINCS+ também tiveram resultados interessantes e, em alguns casos, melhores que o Dilithium. Notou-se também que o algoritmo Rainbow dentre os pós-quânticos não obteve resultados muito bons, geralmente ficando com os piores números – ou até em testes que não foi possível recuperar seus resultados.

Dentre os clássicos, percebeu-se que o algoritmo EC256 teve ótimo resultados, muitas vezes sendo o melhor colocado, com sua outra variação EC384 também tendo resultados bons. Já o algoritmo RSA2048 não teve resultados muito bons, porém em muitos testes assemelhou-se aos resultados do SPHINCS+ e Rainbow. No entanto, o algoritmo RSA8192 foi o pior em todos os testes em questão de tempo, e ficando com as outras métricas com resultados ruins também.

Para resumir os principais resultados do tempo de execução de cada teste, escolheu-se os melhores representantes de cada grupo de algortimos, i.e., RSA2048, EC256, dilithium2-aes, falcon-512, sphincs+-haraka-128f-simple, rainbow-I-classic e os comparou num gráfico. Onde no eixo vertical tem-se a média dos tempos (em escala logarítmica) de todos os testes e no eixo horizontal cada um desses algortimos. Decidiu-se utilizar o algoritmo EC256 como base de comparação, pois dentre os clássicos é um dos mais utilizados atualmente e obteve bons desempenhos nos testes. Dessa forma, os resultados foram normalizados em relação à média dos tempos do EC256.

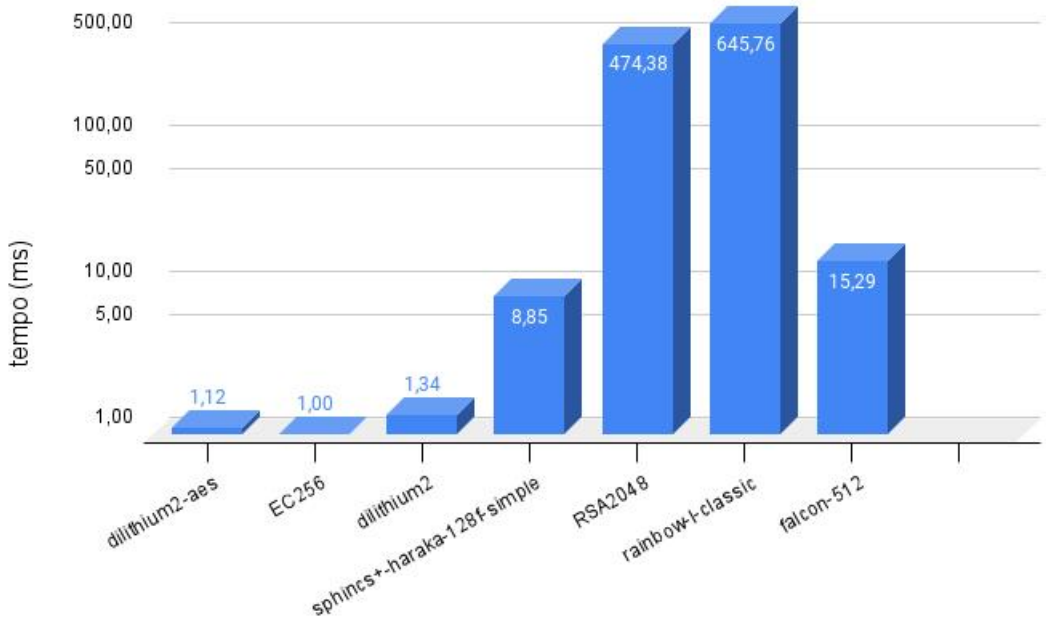

Figura 60 – Comparação dos tempos de execução

Fonte: Original

Percebe-se uma consistência nos resultados, onde os algoritmo EC256, dilithium2-aes e o algoritmo dilithium2 obtêm os melhores tempos enquanto o algoritmo RSA2048 e o algoritmo rainbow-I-classic permanecem nas piores posições. Os outros algoritmos – sphincs+-haraka-128f-simple e falcon-512 – se mantém com resultados intermediários bons.

Outra comparação feita da mesma forma foi em relação aos bytes alocados, como pode-se ver no gráfico abaixo.

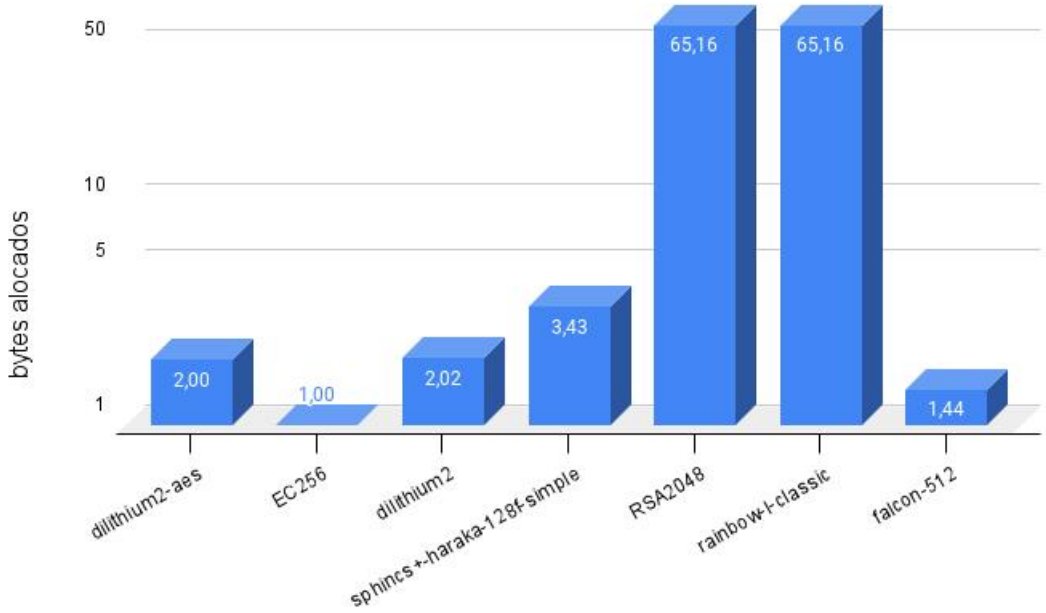

Figura 61 – Comparação dos bytes alocados

Fonte: Original

Da mesma forma que nos tempos de execução, os algoritmos RSA2048 e rainbow-I-classic tiveram os piores resultados nesse quesito também. Entretanto, os outros algoritmos mantiveram resultados muito mais similares nessa comparação, apenas com o sphincs+-haraka-128f-simple se distanciando um pouco mais.

6 CONSIDERAÇÕES FINAIS

Cada vez mais vem se tornando uma realidade os computadores quânticos e o risco que eles impõe à criptografia clássica atual. Dessa forma, foi realizada uma extensão do protocolo ACME para que o mesmo suportasse a criptografia pós-quântica para que mesmo num ambiente de computadores quânticos o protocolo ainda se manter seguro e, dessa maneira, facilitando o tráfego seguro na internet.

Como visto, mudanças formais no protocolo não foram necessárias uma vez que o mesmo suporta a integração de diversos algoritmos de assinatura, não sendo restrito a algoritmos como RSA ou de curvas elípticas. Sendo assim, os esforços foram concentrados nas implementações do protocolo ACME, integrando com bibliotecas e fazendo as devidas alterações nos códigos do cliente e servidor ACME.

Realizadas as modificações, a última etapa teve foco nos testes de desempenho para cada um dos algoritmos integrados, comparando os resultados tanto entre os próprios algoritmos pós-quânticos quanto em relação aos algoritmos clássicos.

Com isso, finaliza-se o trabalho com uma revisão geral dos resultados dos testes, elucidando os principais resultados obtidos.

6.1 TRABALHOS FUTUROS

Restringiu-se ao uso da biblioteca LibOQS, tal restrição traz consigo algumas desvantagens, como um desempenho não tão otimizado para cada algoritmo pós-quântico que foi utilizado devido a ser uma biblioteca que tem que atender a necessidade de todos os algoritmos a qual implementa. Entende-se esse conceito como um *wrapper* ou "empacotador", ou seja, cria um interface para utilizar todos os algoritmos que são suportados, diminuindo o desempenho dos mesmos – isso também é comentado em outros trabalhos, como em (CELI, 2021). Além disso, limita-se a interface disponibilizada pela LibOQS, tendo certas restrições à forma de integração com os outros repositórios.

Dessa forma, é possível contornar esses problema realizando a integração dos próprio algoritmos na biblioteca padrão do Go. Sendo assim, otimizações para cada algoritmo podem ser feitas na própria linguagem Go, obtendo resultados mais expressivos.

O uso de criptografia pós-quântica híbrida, i.e., a combinação de dois ou mais algoritmos, sendo um pós-quântico e outro clássico – no caso de dois algoritmos – de tal maneira que o esquema criptográfico se mantenha seguro mesmo que um dos algoritmos venha a apresentar vulnerabilidades. O desenvolvimento da criptografia pós-quântica prevê, em muitos casos, um estado híbrido para posteriormente avançar para um estado puramente pós-quântico (PAQUIN; STEBILA; TAMVADA, 2020). A adição de algoritmos híbridos implica em decisões de como realizar a combinação,

como construir os certificados TLS com mais de uma chave pública, dentre outras questões. Resultados em relação aos algoritmos híbridos também são de interesse quando aplicados no protocolo ACME, sendo um possível próximo passo.

REFERÊNCIAS

ABDULLAH, Ako. Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data, jun. 2017. Citado na p. 19.

ALBARQI, Aysha *et al.* Public Key Infrastructure: A Survey. **Journal of Information Security**, v. 06, p. 31–37, jan. 2015. DOI: 10.4236/jis.2015.61004. Citado na p. 22.

ALIDOOST NIA, Mehran; SAJEDI, Ali; JAMSHIDPEY, Aryo. An Introduction to Digital Signature Schemes, abr. 2014. Citado na p. 21.

BARNES, R. *et al.* **Automatic Certificate Management Environment (ACME)**. [*S.l.*], mar. 2019. DOI: 10.17487/rfc8555. Disponível em: <https://doi.org/10.17487/rfc8555>. Citado nas pp. 15, 17, 24–26, 28, 34, 38, 78.

BERNSTEIN, Daniel J.; HÜLSING, Andreas *et al.* The SPHINCS Signature Framework. *In:* PROCEEDINGS of the 2019 ACM SIGSAC Conference on Computer and Communications Security. [*S.l.*]: ACM, nov. 2019. DOI: 10.1145/3319535.3363229. Disponível em: <https://doi.org/10.1145/3319535.3363229>. Citado nas pp. 17, 47.

BERNSTEIN, Daniel J.; LANGE, Tanja. Post-quantum cryptography. **Nature**, Springer Science e Business Media LLC, v. 549, n. 7671, p. 188–194, set. 2017. DOI: 10.1038/nature23461. Disponível em: <https://doi.org/10.1038/nature23461>. Citado na p. 16.

BRAY, T. (Ed.). **The JavaScript Object Notation (JSON) Data Interchange Format**. [*S.l.*], dez. 2017. DOI: 10.17487/rfc8259. Disponível em: <https://doi.org/10.17487/rfc8259>. Citado na p. 25.

CELI, Sofía. Implementing and Measuring KEMTLS. *In:* PROGRESS in Cryptology - LATINCRYPT 2021. [*S.l.*: *s.n.*], 2021. P. 88–107. Citado na p. 102.

CHAN, Chia-ling *et al.* Monitoring TLS adoption using backbone and edge traffic. *In:* IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). [*S.l.*]: IEEE, abr. 2018. DOI: 10.1109/infcomw.2018.8406957. Disponível em: <https://doi.org/10.1109/infcomw.2018.8406957>. Citado na p. 15.

COFFMAN, Kerry G.; ODLYZKO, Andrew M. Growth of the Internet. *In:* OPTICAL Fiber Telecommunications IV-B. [*S.l.*]: Elsevier, 2002. P. 17–56. DOI:

10.1016/b978-012395173-1/50002-5. Disponível em: <https://doi.org/10.1016/b978-012395173-1/50002-5>. Citado na p. 15.

COOPER, D. *et al.* **Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile**. [*S.l.*], mai. 2008. DOI: 10.17487/rfc5280. Disponível em: <https://doi.org/10.17487/rfc5280>. Citado na p. 22.

COOPER, D. *et al.* **Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile**. [*S.l.*], mai. 2008. DOI: 10.17487/rfc5280. Disponível em: <https://doi.org/10.17487/rfc5280>. Citado na p. 53.

DEBNATH, Santanu; CHATTOPADHYAY, Abir; DUTTA, Subhamoy. Brief review on journey of secured hash algorithms. *In:* 2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix). [*S.l.*: *s.n.*], 2017. P. 1–5. DOI: 10.1109/OPTRONIX.2017.8349971. Citado na p. 21.

DIZDAREVIĆ, Jasenka *et al.* A Survey of Communication Protocols for Internet of Things and Related Challenges of Fog and Cloud Computing Integration. **ACM Computing Surveys**, Association for Computing Machinery (ACM), v. 51, n. 6, p. 1–29, fev. 2019. DOI: 10.1145/3292674. Disponível em: <https://doi.org/10.1145/3292674>. Citado na p. 23.

FIELDING, R.; GETTYS, J. *et al.* **Hypertext Transfer Protocol – HTTP/1.1**. [*S.l.*], jun. 1999. DOI: 10.17487/rfc2616. Disponível em: <https://doi.org/10.17487/rfc2616>. Citado na p. 31.

FIELDING, R.; RESCHKE, J. (Ed.). **Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content**. [*S.l.*], jun. 2014. DOI: 10.17487/rfc7231. Disponível em: <https://doi.org/10.17487/rfc7231>. Citado na p. 26.

GAO, Zhengxian; LI, Zhongxue; TU, Yaqing. Design and completion of digital certificate with authorization based on PKI. *In:* PROCEEDINGS of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004. [*S.l.*: *s.n.*], 2004. P. 462–466. DOI: 10.1109/IRI.2004.1431504. Citado na p. 22.

GIEN, Michel. A File Transfer Protocol (FTP). **Computer Networks (1976)**, Elsevier BV, v. 2, n. 4-5, p. 312–319, set. 1978. DOI: 10.1016/0376-5075(78)90009-0. Disponível em: <https://doi.org/10.1016/0376-5075(78)90009-0>. Citado na p. 24.

GOLANG. Disponível em: <https://github.com/golang/go>. Acesso em: 27 dez. 2021. Citado na p. 48.

JOHNSON, Don; MENEZES, Alfred; VANSTONE, Scott. The Elliptic Curve Digital Signature Algorithm (ECDSA). **International Journal of Information Security**, Springer Science e Business Media LLC, v. 1, n. 1, p. 36–63, ago. 2001. DOI: 10.1007/s102070100002. Disponível em: <https://doi.org/10.1007/s102070100002>. Citado na p. 20.

JONES, M. **JSON Web Algorithms (JWA)**. [*S.l.*], mai. 2015. DOI: 10.17487/rfc7518. Disponível em: <https://doi.org/10.17487/rfc7518>. Citado na p. 28.

JONES, M. **JSON Web Key (JWK)**. [*S.l.*], mai. 2015. DOI: 10.17487/rfc7517. Disponível em: <https://doi.org/10.17487/rfc7517>. Citado na p. 69.

JONES, M.; BRADLEY, J.; SAKIMURA, N. **JSON Web Signature (JWS)**. [*S.l.*], mai. 2015. DOI: 10.17487/rfc7515. Disponível em: <https://doi.org/10.17487/rfc7515>. Citado nas pp. 61, 79.

JONES, M.; BRADLEY, J.; SAKIMURA, N. **JSON Web Token (JWT)**. [*S.l.*], mai. 2015. DOI: 10.17487/rfc7519. Disponível em: <https://doi.org/10.17487/rfc7519>. Citado na p. 61.

JONES, M.; HILDEBRAND, J. **JSON Web Encryption (JWE)**. [*S.l.*], mai. 2015. DOI: 10.17487/rfc7516. Disponível em: <https://doi.org/10.17487/rfc7516>. Citado na p. 61.

JOSEFSSON, S.; LEONARD, S. **Textual Encodings of PKIX, PKCS, and CMS Structures**. [*S.l.*], abr. 2015. DOI: 10.17487/rfc7468. Disponível em: <https://doi.org/10.17487/rfc7468>. Citado na p. 80.

JOSEFSSON, S.; LIUSVAARA, I. **Edwards-Curve Digital Signature Algorithm (EdDSA)**. [*S.l.*], jan. 2017. DOI: 10.17487/rfc8032. Disponível em: <https://doi.org/10.17487/rfc8032>. Citado na p. 28.

KALISKI, B. **Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax Specification Version 1.2**. [*S.l.*], mai. 2008. DOI: 10.17487/rfc5208. Disponível em: <https://doi.org/10.17487/rfc5208>. Citado na p. 58.

KARTHIK, .S. Data Encryption and Decryption by Using Triple DES and Performance Analysis of Crypto System. *In:* citado na p. 19.

KAUR, Nirmaljeet; SODHI, Sukhman. Article: Data Encryption Standard Algorithm (DES) for Secure Data Transmission. **IJCA Proceedings on International Conference on Advances in Emerging Technology**, ICAET 2016, n. 2, p. 31–37, set. 2016. Full text available. Citado na p. 19.

KUSHWAHA, Prashant *et al.* A Brief Survey of Challenge–Response Authentication Mechanisms. *In:* ICT Analysis and Applications. [*S.l.*]: Springer Singapore, dez. 2020. P. 573–581. DOI: 10.1007/978-981-15-8354-4_57. Disponível em: <https://doi.org/10.1007/978-981-15-8354-4_57>. Citado na p. 26.

LEGO. Disponível em: <https://github.com/go-acme/lego>. Acesso em: 27 dez. 2021. Citado nas pp. 47, 84.

LET'S Encrypt Stats. Disponível em: <https://letsencrypt.org/stats/>. Acesso em: 27 ago. 2021. Citado na p. 16.

LIBOQS. Disponível em: <https://github.com/openquantum-safe/liboqs>. Acesso em: 27 dez. 2021. Citado na p. 47.

LIBOQS Go. Disponível em: <https://github.com/open-quantum-safe/liboqs-go>. Acesso em: 27 dez. 2021. Citado na p. 48.

LIBOQS OIDs. Disponível em: <https://github.com/open-quantum-safe/openssl/blob/OQS-OpenSSL_1_1_1-stable/oqs-template/oqs-sig-info.md.>. Acesso em: 6 jan. 2022. Citado na p. 53.

MARQAS, Ridwan; ALMUFTI, Saman; REBAR, Rasheed. Comparing Symmetric and Asymmetric cryptography in message encryption and decryption by using AES and RSA algorithms. **Xi'an Jianzhu Keji Daxue Xuebao/Journal of Xi'an University of Architecture Technology**, v. 12, p. 3110–3116, mar. 2020. DOI: 10.37896/JXAT12.03/262. Citado na p. 20.

MOCKAPETRIS, P.V. **Domain names - concepts and facilities**. [*S.l.*], nov. 1987. DOI: 10.17487/rfc1034. Disponível em: <https://doi.org/10.17487/rfc1034>. Citado na p. 34.

MOODY, Dustin *et al.* **Status report on the second round of the NIST post-quantum cryptography standardization process**. [*S.l.*], jul. 2020. DOI: 10.6028/nist.ir.8309. Disponível em: <https://doi.org/10.6028/nist.ir.8309>. Citado nas pp. 17, 47.

MOORE, Nicole Casal; HALDERMAN, J. Alex. **How Let's Encrypt doubled the internet's percentage of secure websites in four years**. Disponível em: <https://news.umich.edu/how-lets-encrypt-doubled-the-internets-percentage-of-secure-websites-in-four-years/>. Acesso em: 27 ago. 2021. Citado na p. 16.

NAYLOR, David *et al.* The Cost of the "S" in HTTPS. *In:* PROCEEDINGS of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. [*S.l.*]: ACM, dez. 2014. DOI: 10.1145/2674005.2674991. Disponível em: <https://doi.org/10.1145/2674005.2674991>. Citado na p. 15.

OID. Disponível em: <https://csrc.nist.gov/glossary/term/oid>. Acesso em: 6 jan. 2022. Citado na p. 53.

PAQUIN, Christian; STEBILA, Douglas; TAMVADA, Goutam. Benchmarking Post-Quantum Cryptography in TLS. *In:* PQCRYPTO. [*S.l.*: *s.n.*], 2020. P. 72–91. Citado na p. 102.

PEBBLE. Disponível em: <https://github.com/letsencrypt/pebble>. Acesso em: 27 dez. 2021. Citado na p. 47.

PETITCOLAS, Fabien. Kerckhoffs Principle. *In:* [*S.l.*: *s.n.*], jan. 2011. ISBN 978-1-4419-5905-8. DOI: 10.1007/978-1-4419-5906-5. Citado na p. 19.

PORNIN, T. **Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)**. [*S.l.*], ago. 2013. DOI: 10.17487/rfc6979. Disponível em: <https://doi.org/10.17487/rfc6979>. Citado na p. 28.

SAHU, Aradhana; GHOSH, Samarendra. Review Paper on Secure Hash Algorithm With Its Variants, mai. 2017. DOI: 10.13140/RG.2.2.13855.05289. Citado nas pp. 21, 28.

SHANNON, C. E. Communication Theory of Secrecy Systems∗. **Bell System Technical Journal**, Institute of Electrical e Electronics Engineers (IEEE), v. 28, n. 4,

p. 656–715, out. 1949. DOI: 10.1002/j.1538-7305.1949.tb00928.x. Disponível em: <https://doi.org/10.1002/j.1538-7305.1949.tb00928.x>. Citado na p. 19.

SHOR, Peter W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. **SIAM Journal on Computing**, Society for Industrial & Applied Mathematics (SIAM), v. 26, n. 5, p. 1484–1509, out. 1997. DOI: 10.1137/s0097539795293172. Disponível em: <https://doi.org/10.1137/s0097539795293172>. Citado na p. 16.

SOBTI, Rajeev; GANESAN, Geetha. Cryptographic Hash Functions: A Review. **International Journal of Computer Science Issues, ISSN (Online): 1694-0814**, Vol 9, p. 461–479, mar. 2012. Citado na p. 21.

STALLINGS, William. **Cryptography and Network Security: Principles and Practice**. 6th. USA: Prentice Hall Press, 2013. ISBN 0133354695. Citado nas pp. 19, 20.

TILBORG, Henk C. A. van; JAJODIA, Sushil (Ed.). **Encyclopedia of Cryptography and Security**. [*S.l.*]: Springer US, 2011. DOI: 10.1007/978-1-4419-5906-5. Disponível em: <https://doi.org/10.1007/978-1-4419-5906-5>. Citado na p. 21.

TRIANTAFYLLOU, Anna; SARIGIANNIDIS, Panagiotis; LAGKAS, Thomas. Network Protocols, Schemes, and Mechanisms for Internet of Things (IoT): Features, Open Challenges, and Trends. **Wireless Communications and Mobile Computing**, v. 2018, p. 1–24, set. 2018. DOI: 10.1155/2018/5349894. Citado na p. 23.

TURAU, Volker. HTTPExplorer: Exploring The Hypertext Transfer Protocol. *In:* p. 198–201. DOI: 10.1145/961565.961566. Citado na p. 24.

WARDLAW, William P. The RSA Public Key Cryptosystem. *In:* CODING Theory and Cryptography. [*S.l.*]: Springer Berlin Heidelberg, 2000. P. 101–123. DOI: 10.1007/978-3-642-59663-6_6. Disponível em: <https://doi.org/10.1007/978-3-642-59663-6_6>. Citado na p. 20.

YALAGANDULA, Praveen. DHCP: Dynamic Host Configuration Protocol, nov. 2000. Citado na p. 24.

Apêndices

.1 APÊNDICE A - ARTIGO SBC

Protocolo ACME Pós-Quântico

Matheus de Oliveira Saldanha[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 5040 – 88040-900 – Florianópolis – SC – Brasil

Abstract. *This work begins with an introduction to the main concepts of cryptography, public key infrastructure and computer networks, to a detailed explanation of the functioning of the ACME protocol. And then, an implementation of the ACME protocol is performed using post-quantum algorithms, creating an extension of it. Note that the protocol is already generic enough for the integration of new algorithms, requiring only changes in code implementations. The work is finalized with performance tests for different phases of the ACME protocol, comparing the different classical and post-quantum algorithms with each other. Better performance is observed for the Dilithium algorithm – one of the post-quantum algorithms – in most tests and poorer performance in most for the Rainbow algorithm – another post-quantum candidate.*

Resumo. *Este trabalho inicia com uma introdução aos principais conceitos de criptografia, infraestrutura de chaves públicas e redes de computadores até uma explicação detalhada do funcionamento do protocolo ACME. E então, é realizada uma implementação ao protocolo ACME com o uso de algoritmos pós-quânticos, criando uma extensão do mesmo. Nota-se que o protocolo já é genérico o suficiente para a integração de novos algoritmos, sendo apenas necessário mudanças em implementações em código. O trabalho é finalizado com a realização de testes de desempenho para diversas fases do protocolo ACME, comparando os diversos algoritmos clássicos e pós-quânticos entre si. É observado melhor desemepenho do algoritmo Dilithium – um dos algoritmos pós-quânticos – na maioria dos testes e um desempenho inferior na maioria para o algoritmo Rainbow – outro candidato pós-quântico.*

1. Introdução

Sabe-se que hoje em dia grande parte da informação é mantida na internet e distribuída na mesma, principalmente, via websites [Coffman and Odlyzko 2002]. Dessa forma, questões de integridade, sigilo e autenticidade da informação tornam-se cada vez mais relevantes no mundo moderno. Nesse aspecto, é importante garantir que essas propriedades de segurança da informação sejam garantidas.

O protocolo HTTPS – HTTP [Naylor et al. 2014] com TLS [ling Chan et al. 2018] – inicialmente foi feito na década de 1990 para transações bancárias e outros processos onde informações confidenciais estão em risco. Porém, hoje em dia, o protocolo é o mais utilizado para navegar na web, onde é feita grande parte da troca de informação da internet. Para isso, o HTTPS nos permite verificar a integridade dos dados, ou seja, permite que a informação não seja modificada no processo de transmissão. Também oferece confidencialidade, mantendo a informação em sigilo onde apenas as partes autorizadas têm acesso. E garante a autenticação do servidor,

ou seja, o servidor é quem diz ser e, também, permite – de maneira opcional – que o cliente se autentique. Entretanto, essa última questão exige certa complexidade para ser implementada, requerendo toda uma infraestrutura.

Para garantir a integridade dos dados, o protocolo HTTPS utiliza mecanismos de resumos criptográficos. Os resumos criptográficos, adicionados com outros processos, asseguram que a informação foi mantida imutável durante a troca.

Assim como resumos criptográficos sustentam a integridade de dados no HTTPS, a cifragem garante a confidencialidade da informação. Isso é feito transformando a informação clara em um informação cifrada, sendo inviável a leitura para indivíduos sem autorização (que não possuem a chave criptográfica).

Já a autenticação da informação, como mencionado anteriormente, requer diferentes abstrações para que seja assegurada. Dentre elas, temos os certificados digitais – os quais terão grande foco nesse trabalho – que podem ser vistos como identidades digitais, de uma maneira simplificada. Os certificados digitais se baseiam em diversas primitivas criptográficas, como criptografia assimétrica e assinaturas digitais, por exemplo. Além disso, a partir deles criam-se outras entidades como as Autoridades Certificadoras e Infraestruturas de Chaves Públicas, essenciais para a autenticação na internet. Esses e outros tópicos serão vistos a fundo no decorrer do trabalho.

Nesse âmbito, surge o protocolo ACME [Barnes et al. 2019]. ACME ou Ambiente Automático de Gerenciamento de Certificados (*Automated Certificate Management Environment*, em inglês) automatiza toda essa complexidade associada a um certificado digital, englobando processos de emissão, renovação e revogação dos certificados. Além de facilitar essas tarefas, o protocolo é de uso gratuito, mantido por uma comunidade e não apenas centralizado em uma organização.

O protocolo, desde sua primeira versão publicada em 2016 e sua segunda sendo publicada em 2018 já rendeu alguns resultados. Isso pode ser notado pelo fato de que de 2016 para 2018 o percentual de sites que utilizam HTTPS passou de 40% para 80% [Moore and Halderman 2021] [Let 2021]. Isso se deve, principalmente, pelos esforços da Let's Encrypt, principal entidade responsável pelo desenvolvimento do protocolo e sua larga adesão. Sendo um dos principais objetivos da Let's Encrypt a difusão do uso do HTTPS na web.

Contudo, novos riscos à segurança digital estão surgindo. O desenvolvimento de computadores quânticos é uma ameaça aos esquemas tradicionais de criptografia. Apesar de, publicamente, estar numa fase muito primitiva, o surgimento de computadores quânticos de larga escala oferece riscos à toda uma infraestrutura de segurança já bem estabelecida.

Computadores quânticos trabalham de forma diferente do computadores clássicos, não utilizando bits e sim qubits. Eles exploram propriedades da mecânica quântica, como superposição e interferência e, assim, conseguem computar certos algoritmos que computadores clássicos são incapazes de resolver eficientemente. Alguns desses algoritmos, como o algoritmo de Shor [Shor 1997], são passíveis de ameaça a criptografia clássica, a qual supõe a dificuldade de computar certos problemas matemáticos.

Dessa forma, a proteção da infraestrutura atual requer a modificação de protocolos

e de algoritmos para dar suporte à criptografia pós-quântica [Bernstein and Lange 2017], a qual utiliza algoritmos em computadores clássicos de forma a serem seguros à ataques de computadores quânticos.

Para isso, é importante destacar que os sistemas que utilizam de certificação digital para autenticação precisarão ser atualizados, gerando novamente a necessidade de intervenção. Assim, este trabalho tem objetivo de avaliar e modificar o protocolo ACME para a atualização destes sistemas para uso de criptografia pós-quantica. Espera-se que o ACME possa automatizar a atualização de sistemas para um ambiente pós-quântico facilitando a atualização para diferentes cenários e com menor esforço de intervenção humana, tais como: IoT ou Internet das Coisas (*Internet of Things*), Servidores HTTPS distribuídos, Redes de sensores, Indústria 4.0 e outros.

2. Fundamentação Teórica

2.1. Criptografia Simétrica

A criptografia simétrica tem como principal função a garantia do sigilo na troca de mensagens. Para manter a confidencialidade das mensagens ela possui dois principais componentes: um **algoritmo** e, também, uma **chave** [Stallings 2013].

O sistema funciona, de maneira simples, como uma função de dois parâmetros:

$$f(k, p) = c$$

Onde k é a chave e p o texto a ser cifrado. Já c é o texto cifrado e f o algoritmo.

Conforme o desenvolvimento da criptografia simétrica, outras questões foram incorporadas para aumentar a segurança do sistema, como modos de operação dos algoritmos.

Como forma de ilustrar algoritmos de criptografia simétrica tem-se o DES [Kaur and Sodhi 2016], 3DES [Karthik 2014] e AES [Abdullah 2017].

A chave é a segurança do sistema criptográfico. É com a ocultação da chave do meio público que o sistema deve se manter seguro, deixando com que o algoritmo seja de domínio público. Isso pode parecer contraditório em algum ponto, pois um pensamento inicial seria de que esconder o algoritmo tornasse o sistema mais seguro. Porém não é o que se tem sido afirmado com o tempo. Esconder o algoritmo faz com que ele seja menos testado pelo público, podendo assim existir falhas e vulnerabilidades que podem ser exploradas. Isso pode ser visto também nos príncipios de Kerckhoffs [Petitcolas 2011] e também nos textos de Claude Shannon [Shannon 1949]. A chave então é acordada entre as partes que vão efetuar a troca de mensagens e deve ser mantida oculta pelas mesmas.

Uma das principais vantagens desses sistemas é sua performance, sendo aplamente utilizada devido sua velocidade e tamanho de chaves. Porém, o gerenciamento de chaves, por exemplo, torna-se mais complexo à medida que aumenta o número de pessoas que se comunicam. Para cada N usuários, são necessários N usuários tomados dois a dois, ou seja, $\binom{N}{2} = \frac{N^2-N}{2}$ chaves.

2.2. Criptografia Assimétrica

Criptografia assimétrica ou criptografia de chave pública recebe este nome, pois temos um par de chaves composto de uma **chave pública** e uma **chave privada** [Stallings 2013].

A chave pública recebe este nome justamente por ser de domínio público, qualquer indivíduo pode conhecê-la sem prejudicar a segurança do sistema. Já a chave privada, assim como na criptografia simétrica, deve ser mantida em segredo, longe do público.

Para cifrar uma mensagem e torná-la sigilosa entre duas partes, a entidade que deseja enviar a mensagem a cifra com a chave pública do destinatário. Dessa forma, apenas o destinatário (que possui a chave privada correspondente) pode decifrar a mensagem. Também existem outros usos da criptografia assimétrica, como a assinatura digital que será visto mais a frente.

Para exemplificar alguns algoritmos de criptografia assimétrica, tem-se o RSA [Wardlaw 2000] e o ECDSA [Johnson et al. 2001].

Além de possibilitar outros usos das chaves além de cifrar mensagens, a criptografia de chave pública facilita o gerenciamento das chaves. Pois cada entidade necessita apenas de um par de chaves, ou seja, para N entidades precisa-se de apenas N pares de chave, diferente da criptografia simétrica onde o gerenciamento de chaves torna-se um problema, como foi visto anteriormente. Entretanto, seu desempenho – da criptografia assimétrica – tende a ser inferior quando comparado à criptografia simétrica [Marqas et al. 2020]. Porém, seus usos variam e, dessa forma, os dois modelos coexistem para diversas aplicações. Na assinatura digital, por exemplo, a criptografia assimétrica é empregada em conjunto com resumos criptográficos, conceito apresentado a seguir.

2.3. Resumo Criptográfico

Resumo criptográfico ou função *hash* criptográfica recebe este nome por representar, de certa forma, um resumo da mensagem a qual função foi aplicada. São funções, também, que são inviáveis de serem invertidas.

Pode-se definir uma função *hash* criptográfica como:

$$f(m) = h$$

onde **f** é a função, **m** a mensagem e **h** o resumo criptográfico [Stallings 2013]. E deve possuir as seguintes propriedades:

- **Resistência à pré-imagem**: Dado um valor de resumo criptográfico h, deve ser inviável encontrar qualquer mensagem m tal que $h = f(m)$. Essa propriedade está relacionada ao fato da função ser difícil de ser invertida.
- **Resistência à segunda pré-imagem**: Dada uma mensagem m_1, deve ser difícil encontrar uma mensagem $m_2 \neq m_1$ tal que $h = f(m_1) = f(m_2)$.
- **Resistência à colisão**: Deve ser inviável encontrar quaisquer duas mensagens $m_1 \neq m_2$ tal que seus resumos criptográficos sejam iguais.

Nota-se que a propriedade de **resistência à segunda pré-imagem** é similar à propriedade de **resistência à colisão**. Porém a primeira se diferencia da segunda quando fixa-se um resumo criptográfico h.

Algumas funções de resumo criptográfico são SHA [Sahu and Ghosh 2017]. Outras funções e um histórico podem ser encontrados em [Debnath et al. 2017].

Resumos criptográficos possuem diversas aplicações, como auxiliar na autenticação de uma mensagem, assim como compõe um papel fundamental na assinatura digital [Sobti and Ganesan 2012].

2.4. Assinatura Digital

A assinatura digital recebe este nome por ser uma analogia muito similar à assinatura de próprio punho [Alidoost Nia et al. 2014]. Ela serve para que o signatário prove sua identidade perante outra parte.

No mundo digital existem diversos esquemas de assinatura digital, i.e., formas de assinar e de verificar a validade da assinatura [van Tilborg and Jajodia 2011]. Eles se baseiam na criptografia assimétrica, como visto anteriormente. De maneira simples, esses esquemas se baseiam no conceito onde o signatário utiliza sua própria chave privada para assinar a mensagem – utilizando algum esquema de assinatura digital. E, então, qualquer entidade pode validar a assinatura utilizando a chave pública do signatário e verificando sua validade com base no esquema utilizado.

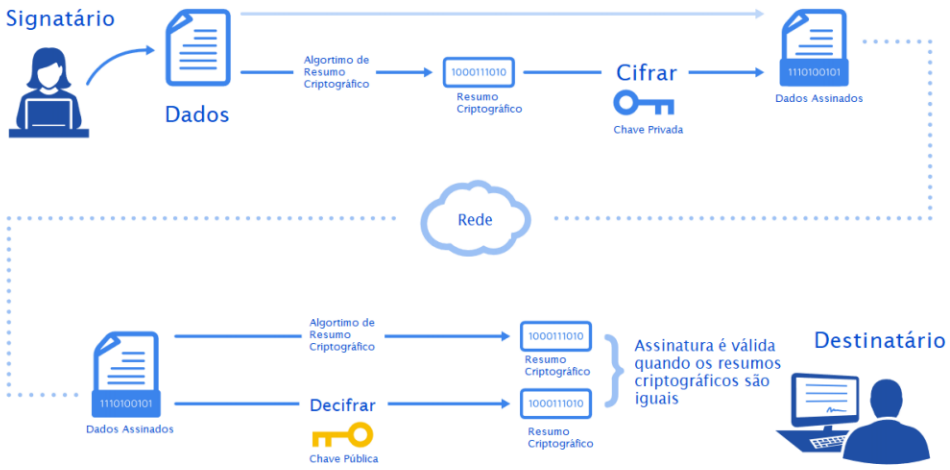

Figura 1. Assinatura DigitalFonte: Original

Resumos criptográficos também são utilizados nesse contexto em muitos dos esquemas de assinatura digital. Em muitos casos a mensagem em si não é assinada, mas seu resumo criptográfico é. Isso pode trazer algumas vantagens, como velocidade de assinar e de verificar a assinatura, pois, de maneira geral, o resumo criptográfico tende a ser menor que a mensagem original – desde que as propriedades do resumo criptográfico sejam garantidas como foi mencionado anteriormente.

Podemos generalizar o uso de resumos criptográficos em assinatura digital de maneira simplificada em alguns passos. Primeiro o signatário realiza o resumo criptográfico da mensagem a ser assinada e então utiliza sua chave privada para assinar – entende-se, nessa analogia, assinar por cifrar – esse resumo. O signatário envia tanto a mensagem original quanto o resumo assinado. Dessa forma, o destinatário utiliza a chave pública do signatário para ”decifrar”a assinatura do resumo criptográfico e, após calcular o resumo da mensagem que recebeu, compara os dois resumos. Quando os dois são iguais então a assinatura está válida, caso contrário não. A Figura 1 ilustra bem esse cenário.

Entretando um problema que surge é o fato de que não garante-se que a assinatura é de quem diz ser, apenas que a chave pública e chave privada pertencem ao mesmo par.

Para isso, surge o conceito de certificado digital, onde podemos relacionar o par de chaves à uma identidade.

2.5. Certificado Digital e Autoridade Ceritificadora

Certificados digitais são documentos que relacionam uma chave pública a uma identidade, ou seja, ao seu proprietário [Gao et al. 2004].

Além de informações sobre a chave pública, o certificado contém diversas outras informações a respeito do proprietário e de quem emitiu o certificado, usos da chave, como assinatura, cifragem etc. O ceritificado contém uma assinatura digital da entidade que emitiu, também conhecida como **Autoridade Certificadora** ou **AC**. A assinatura serve para garantir que tal AC confia no certificado emitido.

Certificados digitais seguem, principalmente, um padrão de certificado chamado X.509 [Cooper et al. 2008a]. Dessa forma, diferentes perfis de certificados – cada um mais apropriado para um certo caso de uso – podem ser feitos, desde que sigam o padrão.

Dessa forma, ceritificados digitais são emitidos por autoridades certificadoras, relacionando a chave pública a uma entidade, seja ela um indivíduo, uma organização ou até uma máquina.

2.6. Infraestrutura de Chaves Públicas

Nesse contexto, infraestrutura de chaves públicas, ou ICPs, podem ser vistas como cadeias de confiança entre diferentes ACs [Albarqi et al. 2015].

Pode-se fazer uma analogia a uma árvore de nodos, onde cada nodo representa uma AC. Dessa forma, o primeiro nodo representa a AC raiz, a qual tem seu certificado autoassinado na maioria das implementações. Abaixo dela temos as ACs intermediárias e, no último nível, as ACs folhas, as quais emitem certificados para usuários finais. A tarefa de emissão dos certificados para usuários finais também pode ser responsabilidade de outra entidade, uma **AR** ou **Autoridade Registradora**. A AR tem como objetivo validar documentos dos usuários finais e emitir o certificado para os mesmos. Dessa maneira, a AC num nível acima sempre assina o certificado da AC do nível abaixo, dessa forma, construindo uma cadeia de confiança. Confiando-se na AC raiz, confia-se no sistema como um todo. Pode-se visualizar essa cadeia na Figura 2.

Dessa maneira, existem ICPs para diversos contextos. Tem-se, por exemplo, a ICP-Brasil e ICP-Edu. A primeira para casos de uso mais relacionados ao governo brasileiro e a segunda para propósitos acadêmicos. Um exemplo típico de utilização de ICPs e Certificados Digitais pode ser visto em protocolos de rede, tais como explicados na seção seguinte.

2.7. Protocolos de Rede

Os protocolos de rede são conjuntos estabelecidos de regras que determinam como os dados são transmitidos entre diferentes dispositivos na mesma rede [Triantafyllou et al. 2018].

Essencialmente, eles permitem que dispositivos conectados se comuniquem entre si, independentemente de quaisquer diferenças em seus processos internos, estrutura ou design. Sendo assim, os protocolos de rede desempenham um papel crítico nas comunicações digitais modernas [Dizdarević et al. 2019].

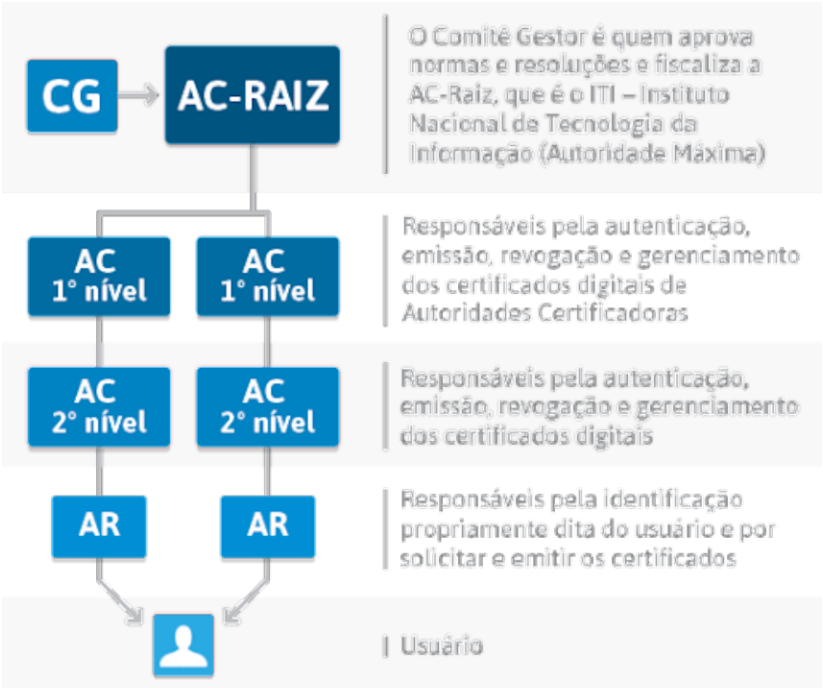

Figura 2. Infraestrutura de Chaves Públicas
Fonte: Benefícios e Aplicações da Certificação Digital

Semelhante à maneira como falar o mesmo idioma simplifica a comunicação entre duas pessoas, os protocolos de rede possibilitam que os dispositivos interajam uns com os outros por causa de regras predeterminadas incorporadas ao software e hardware dos dispositivos.

Dessa maneira, os protocolos de rede são específico de cada contexto. Podem servir para transferir arquivos [Gien 1978], para buscar páginas na web [Turau 2003], para se conectar a uma rede [Yalagandula 2000], etc.

Dentre essa variedade, tem-se o protocolo ACME [Barnes et al. 2019]. Esse tem como principal contexto o ciclo de vida de certificados digitais. Essencialmente, gerencia diversos processos em que um certificado digital possa passar, como será visto à fundo na próxima seção.

3. Protocolo ACME

3.1. Introdução

Originalmente desenvolvido pelo Grupo de Pesquisa de Segurança da Internet ou Internet Security Research Group (ISRG) para sua própria AC, Let's Encrypt, uma entidade certificadora que emite gratuitamente certificados digitais, o protocolo ACME tem-se tornado um padrão comercial gradativamente. Mais recentemente, em 2019, a organização Força-Tarefa de Engenharia da Internet ou Internet Engineering Task Force (IETF) padrozinou o protocolo na RFC8555 [Barnes et al. 2019], favorecendo sua adoção por diversas entidades.

Como mencionado anteriormente, o protocolo tem como principal objetivo automatizar o processo de gerenciamento de certificados. Isso é feito a partir do agente ACME

e do servidor ACME, as entidades do protocolo. O cliente do protocolo executa em um website, por exemplo, enquanto o servidor roda em uma AC. Sendo assim, o cliente realiza requisições – como emissão de um certificado – via HTTPS para o servidor por meio de mensagens JSON [JSO 2017].

De maneira geral, os principais componentes do procotolo podem ser definidos como:

- **Nonce:** Número aleatório de uso único. Tem como função garantir que o proprietário do par de chaves prove sua posse assinando tal *nonce*.
- **Token:** Código criado pelo servidor e enviado ao cliente, sendo um importante componente do desafio ACME. Pode ter seu uso variado dependendo do desafio ACME, como será visto. Entretanto, pode-se generalizar seu uso como um identificador da instância do desafio ACME juntamente com alguma informação específica da conta ACME, como a impressão digital da chave pública. Dessa forma, pode servir para criar um caminho para o proprietário da conta provar posse de um domínio.
- **Desafio:** Constitui a parte de autenticação do protocolo. Existem diferentes desafios que o protocolo suporta, como o desafio DNS e o desafio HTTP como serão vistos a seguir. Cada desafio tem suas pecualiaridades, porém o propósito se mantém único: provar a posse do par de chaves da conta e de um domínio.
- **Cliente ACME:** Entidade do protocolo ACME à qual realiza requisições. Requisições podem ser vistas como pedidos para a emissão de um certificado, renovação, revogação ou até mesmo a autenticação nos passos iniciais do protocolo. Sendo essas requisições respondidas pelo servidor ACME.
- **Servidor ACME:** Entidade do protocolo à qual responde as requisições do cliente ACME. Dessa forma, autenticando o cliente, enviando certificados, renovando ou revogando os mesmos. O servidor ACME está relacionado com alguma AC, sendo mantida no mesmo servidor (o caso deste trabalho) ou em um servidor separado.
- **Par de chaves da conta:** Cada conta ACME, ou seja, cada relação entre um cliente e um servidor possui um par de chaves assimétricas. As mesmas servem para realizar a autenticação na troca de mensagens entre essas duas entidades por meio de assinaturas digitais.

Mais informações sobre outros componentes e sobre a troca de mensagens podem ser vistos em detalhes em [Barnes et al. 2019]. Porém, para tudo ser configurado, primeiramente, temos a etapa de autenticação do protocolo.

3.2. Autenticação

A etapa de autenticação do protocolo tem como principal objetivo estabelecer um vínculo de confiança entre o cliente e o servidor ACME. Para tal vínculo ser instituído, surge o processo de ACME *challenge* (desafio ACME), o qual garante que os certificados emitidos serão para usuários confiáveis. O desafio ACME se encaixa numa classe de protocolos conhecidos como **desafio-resposta** [Kushwaha et al. 2020], nesse caso o desafio faz parte de um protocolo que o engloba – o protocolo ACME. Protocolos desafio-resposta são caracterizados por uma entidade que envia um desafio e uma segunda entidade o recebe e o responde. Um exemplo simples muito utilizado na web é a autenticação por senha, onde o desafio é responder o usuário e senha certos. Nesse caso o servidor web lança o desafio

e o cliente que quer se autenticar precisa fornecer as informações corretas. Já na versão 2 do protocolo ACME temos dois desafios possíveis: **desafio HTTP** e **desafio DNS**.

Dessa forma, primeiramente, o cliente ACME realiza um GET [HTT 2014] para o caminho do diretório (geralmente */acme/directory*) no servidor ACME, e então o servidor ACME responde com um objeto JSON contendo todos os recursos possíveis do protocolo (Resposta 1 na Figura 7) como pode ser visto a seguir na Figura 3.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "newNonce": "https://example.com/acme/new-nonce",
    "newAccount": "https://example.com/acme/new-
       account",
    "newOrder": "https://example.com/acme/new-order",
    "newAuthz": "https://example.com/acme/new-authz",
    "revokeCert": "https://example.com/acme/revoke-
       cert",
    "keyChange": "https://example.com/acme/key-change"
       ,
    "meta": {
        "termsOfService": "https://example.com/acme/
           terms/2017-5-30",
        "website": "https://www.example.com/",
        "caaIdentities": ["example.com"],
        "externalAccountRequired": false
        }
}
```

Figura 3. Resposta ao GET pelo servidor ACME */acme/directory*
Fonte: RFC 8555

Após isso, o cliente envia uma mensagem HEAD para o caminho do *newNonce* (geralmente */acme/new-nonce*) para que o servidor envie um *nonce*. O *nonce* é enviado no cabeçalho da resposta HTTP no campo **Replay-Nonce**, servindo de mecanismo de **não repúdio**. O não repúdio visa garantir que o autor da mensagem não tenha capacidade de negar ter criado e assinado a mesma. Nesse caso, o *nonce* assinado na mensagem é aleatório e considera-se que, naquele instante, apenas esse cliente poderia ter gerado esse *nonce* e assinando o mesmo o cliente não tem como negar posteriormente a autoria da mensagem enviada, como pode ser visto a seguir na Figura 4 (Resposta 2 da Figura 7).

```
HTTP/1.1 200 OK
Replay-Nonce: oFvnlFP1wIhRlYS2jTaXbA
Cache-Control: no-store
Link: <https://example.com/acme/directory>;rel="index"
```

Figura 4. Resposta ao HEAD pelo servidor ACME *`/acme/new-nonce`*
Fonte: RFC 8555

O *nonce* não é particular dessa resposta do servidor, todas as respostas aos POSTs do cliente a partir desse ponto possuem um *nonce* para que o cliente assine com o par de chaves que irá gerar em seguida. Como o servidor cria esses *nonces* é particular da implementação.

A geração de um par de chaves feita pelo cliente, como explicitado na RFC8555 [Barnes et al. 2019], para os algoritmos de geração e assinatura de chaves criptográficas [Jones 2015a], é necessário que a AC do servidor ACME dê suporte à curvas elípticas do tipo ES256, i.e., ECDSA [Pornin 2013] – um algoritmo de assinatura digital padronizado que pode ser utilizado em conjunto à uma função de hash criptográfico – com a função de resumo criptográfico SHA256 [Sahu and Ghosh 2017]. Além disso, a RFC recomenda dar suporte ao algoritmo EdDSA [Josefsson and Liusvaara 2017], mais especificamente, utilizando a curva Ed25519, porém esse segundo algoritmo não tem a obrigatoriedade de ser suportado. Vale a pena ressaltar que, apesar de ser obrigatório a implementação do algoritmo ES256, não é necessariamente obrigatório seu uso no protocolo, apenas o suporte pela implementação deve existir.

Após a criação do par de chaves, para finalizar a criação da conta ACME do cliente com o servidor, o cliente envia uma mensagem POST para o caminho de *newAccount* do serivdor ACME (geralmente */acme/new-account*), como pode ser visto a seguir na Figura 5.

```
POST /acme/new-account HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "jwk": {...},
        "nonce": "6S8IqOGY7eL2lsGoTZYifg",
        "url": "https://example.com/acme/new-account"
    }),
    "payload": base64url({
        "termsOfServiceAgreed": true,
        "contact": [
        "mailto:cert-admin@exemplo.org",
        "mailto:admin@exemplo.org"
        ]
    }),
    "signature": "RZPOnYoPs1PhjszF...-nh6X1qtOFPB519I"
}
```

Figura 5. POST do cliente ACME *`/acme/new-account`*
Fonte: RFC 8555

Dessa mensagem, nota-se o campo *protected*, o qual tem suas informações assinadas e sua assinatura mantida no campo *signature*. Desse campo, tem-se o campo *alg* que identifica o algoritmo utilizado para assinar, o campo *jwk* que contém a chave pública da conta a ser criada, *nonce* que contém o *nonce* a ser assinado, *url* que contém o caminho para o qual a requisição POST está sendo enviada. Além disso, tem-se outras informações – não muito relevantes para avaliar a criação da conta – no campo *payload*. Nota-se que a partir desse ponto, assim como toda a resposta do servidor contém um *nonce* que deve ser assinado pelo cliente na próxima mensagem POST, todo o POST do cliente é assinado.

Em resposta a essa mensagem o servidor envia uma mensagem que entre outras informações, possui o caminho para a conta criada (Resposta 3 da Figura 7), conforme abaixo na Figura 6.

```
HTTP/1.1 201 Created
Content-Type: application/json
Replay-Nonce: D8s4D2mLs8Vn-goWuPQeKA
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/acct/evOfKhNU60wg

{
    "status": "valid",
    "contact": [
    "mailto:cert-admin@exemplo.org",
    "mailto:admin@exemplo.org"
    ],
    "orders": "https://example.com/acme/acct/evOfKhNU60
      wg/orders"
}
```

Figura 6. Resposta ao POST pelo servidor ACME *`/acme/new-account`*
Fonte: RFC 8555

Essa etapa inicial de autenticação pode ser visualizada de maneira simples na Figura 7.

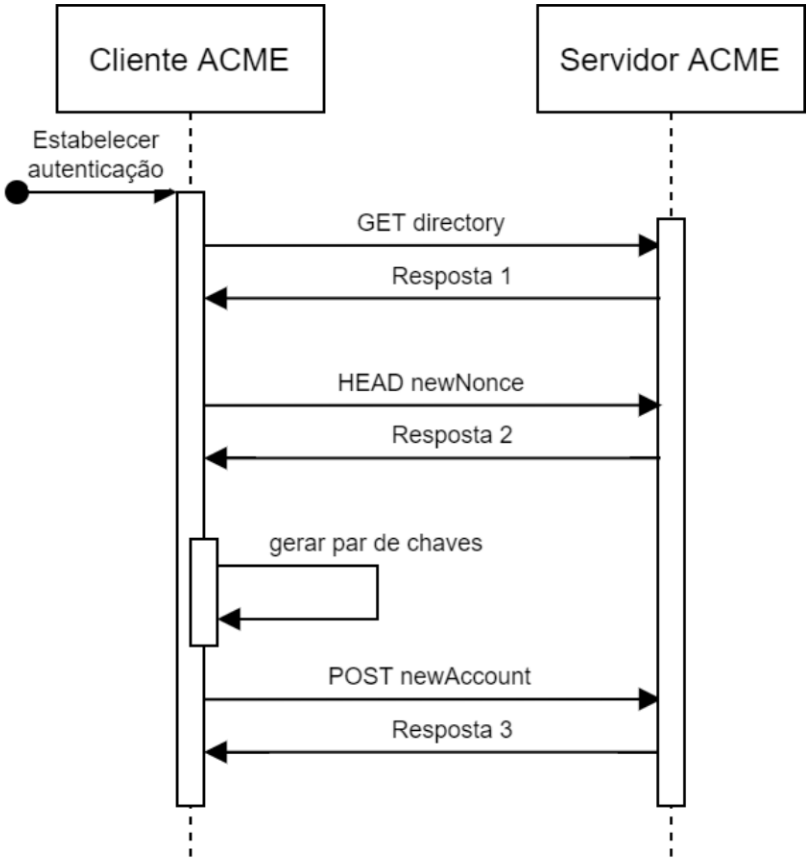

Figura 7. Autenticação ACME
Fonte: RFC 8555

3.3. Desafio HTTP

Como já mencionado, os desafios servem para provar controle sobre certo domínio neste protocolo. Isso acontece quando, principalmente, o cliente ACME quer emitir um certificado para um domínio (newOrder). Dessa forma, quando o cliente manda uma mensagem para o endereço de newOrder o servidor responde com uma mensagem seguindo o seguinte formato da Figura 8:

Onde cada desafio do vetor *challenges* possui qual é o desafio (seu tipo), o seu endereço e o *token* para completar o desafio.

Sendo assim, tem-se como o desafio mais utilizado o HTTP, recebendo este nome devido ao protocolo HTTP [Fielding et al. 1999].

1. O servidor ACME fornece um *token* para o cliente (como mencionado anteriormente).
2. O cliente coloca um arquivo em seu servidor da web em *http://¡dominio_exemplo¿/.well-known/acme-challenge/¡token¿*. Esse arquivo contém o *token*, além de uma impressão digital da chave da sua conta. A impressão digital da chave é um resumo criptográfico da mesma, representando ela de maneira comprimida.
3. Depois o cliente ACME notifica ao servidor que o arquivo está pronto.

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "pending",
    "expires": "2016-01-02T14:09:30Z",
    "identifier": {
    "type": "dns",
    "value": "www.exemplo.org"
},
    "challenges": [
        {
        "type": "http-01",
        "url": "https://example.com/acme/chall/prV_B7
           yEyA4",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        },
        {
        "type": "dns-01",
        "url": "https://example.com/acme/chall/Rg5dV14
           Gh1Q",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        }
    ]
}
```

Figura 8. Resposta ao POST pelo servidor ACME *`/acme/new-order`*
Fonte: RFC 8555

4. O servidor ACME tenta recuperar o arquivo. Potencialmente, isso pode levar várias tentativas e cada tentativa pode realizar a recuperação de diferentes formas.
5. Nesse aspecto, se as verificações de validação obtiverem as respostas corretas de seu servidor web, a validação será considerada bem-sucedida e o cliente poderá emitir seu certificado. Por outro lado, se as verificações de validação falharem, o cliente terá que tentar novamente com um novo certificado.

Dos passos acima, o qual vale a pena ressaltar seria como o cliente notifica o servidor que completou o desafio. Isso se da por meio de uma simples mensagem ao servidor com o conteúdo **{}**, conforme abaixo na Figura 9.

Portanto, as chaves – antes já autenticadas – são utilizadas nas trocas de mensagens para realizar a autenticação das mesmas por meio das assinaturas digitais. As chaves são utilizadas tanto durante os desafios quanto depois para outras requisições, como a emissão de certificados digitais. Outro ponto importante é o uso do *token* nesse desafio o qual serve para criar um caminho onde o cliente disponibilizará o arquivo contendo a impressão digital da sua chave pública e o *token* em si, provando, assim, que possui controle sob o domínio. Sendo assim, a validação do servidor se resume a verificar se o arquivo

```
POST /acme/chall/prV_B7yEyA4
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
    "alg": "ES256",
    "kid": "https://example.com/acme/acct/evOfKhNU60wg"
      ,
    "nonce": "UQI1PoRi5OuXzxuX7V7wL0",
    "url": "https://example.com/acme/chall/prV_B7yEyA4"
    }),
    "payload": base64url({}),
    "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

Figura 9. POST do cliente ACME *"/acme/chall"*
Fonte: RFC 8555

contém as informações esperadas e se está no caminho em que deveria estar. Para ser mais exato, as verificações do servidor são:

1. Constrói um URL "http://dominio_exemplo/.well-known/acme-challenge/token", onde:
    - o campo de domínio é definido para o nome de domínio que está sendo verificado; e
    - o campo do token é definido como o token no desafio
2. Verifica se o URL resultante está bem formado.
3. Cancela a referência do URL usando uma requisição HTTP GET. Este pedido deve ser enviado para a porta TCP 80 no servidor HTTP.
4. Verifica se o corpo da resposta é uma **chave de autorização** bem formada
5. Verifica se a autorização da chave fornecida pelo servidor HTTP corresponde a autorização de chave armazenada pelo servidor.

Esse desafio possui diversas vantagens. É fácil automatizar sem conhecimento extra sobre a configuração de um domínio, além de funcionar com servidores da web já prontos para uso. A Figura 10 forma simplificada – num cenário onde as informações estão corretas e o servidor consegue realizar a verificação sem problemas.

3.4. Desafio DNS

Por outro lado, tem-se o desafio DNS, recebendo este nome devido ao protocolo DNS [Mockapetris 1987]. Este pede que o cliente prove que possui controle sob o DNS de seu nome de domínio, colocando um valor específico em um registro TXT sob esse nome de domínio. É mais difícil de configurar do que HTTP, entretanto pode funcionar em cenários que o HTTP não pode. O desafio inicia-se da mesma forma que o anterior, com o cliente escolhendo qual desafio quer realizar dentre os enviados no vetor *challenges* pelo servidor. Além disso, assim como no desafio HTTP as mensagens também são assinadas

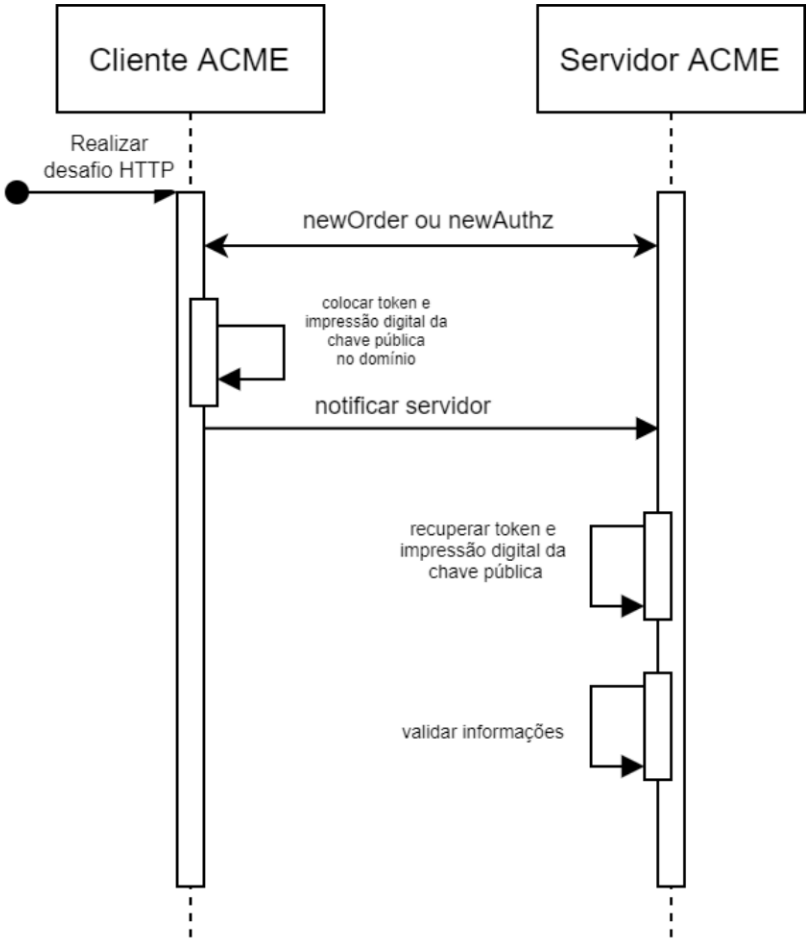

Figura 10. Desafio HTTP ACME
Fonte: RFC 8555

pelo par de chaves já autenticado na etapa anterior. Ou seja, a diferença entre os desafios apenas está na maneira em que o cliente prova o controle sobre tal dominío, mantendo as outras questões iguais.

1. O servidor ACME fornece ao seu cliente um *token*.
2. O cliente criará um registro TXT derivado desse *token* e da impressão digital da chave pública da sua conta, assemelhando-se ao desafio HTTP mencionado.
3. O cliente, então, colocará esse registro em *_acme-challenge.¡dominio_exemplo¿*.
4. Então, o servidor ACME consultará o sistema DNS para esse registro.
5. Se o servidor encontrar uma correspondência, o cliente pode prosseguir para emitir um certificado.

De maneira mais detalhista, a partir do *token* do desafio DNS – disponibilizado pelo servidor no desafio presente no vetor *challenges* – o cliente concatena o *token* com a impressão digital da chave pública e, por fim, realiza um resumo criptográfico SHA256 da concatenação. Essa concatenação de *token* com impressão digital da chave pública é referenciada na [Barnes et al. 2019] por *Key Authorization*. Por fim, o cliente constrói o dominío de validação concatenando ”_acme-challenge.”ao domínio a ser validado num registro TXT. Por exemplo, caso o domínio se chamasse *www.exemplo.org* a estrutura do registro seria similar a seguinte:

```
_acme-challenge.www.exemplo.org. 300 IN TXT "gfj9Xq...
  Rg85nM"
```

Figura 11. Registro DNS ACME
Fonte: RFC 8555

Por fim, o cliente envia uma mensagem com o conteúdo **{}**, similar ao HTTP, para notificar o servidor que o mesmo pode verificar a completude do desafio, como na Figura 12.

```
POST /acme/chall/Rg5dV14Gh1Q
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "kid": "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "SS2sSl1PtspvFZ08kNtzKd",
        "url": "https://example.com/acme/chall/Rg5dV14Gh1Q"
    }),
    "payload": base64url({}),
    "signature": "Q1bURgJoEslbD1c5...3pYdSMLio57mQNN4"
}
```

Figura 12. POST do cliente ACME *`/acme/chall`*
Fonte: RFC 8555

Sendo assim, o cliente pode ter vários registros TXT para o mesmo nome. No entanto, o cliente deve certificar-se de limpar os registros TXT antigos, pois se o tamanho da resposta ficar muito grande, o servidor ACME começará a rejeitá-los.

A verificação do desafio pelo servidor se resume as seguintes etapas:

1. Calcula o resumo criptográfico SHA-256 da chave da conta ACME
2. Consulta de registros TXT para o nome de domínio de validação, como demonstrado acima
3. Verifica se o conteúdo de um dos registros TXT corresponde ao valor de resumo criptográfico

A Figura 13 forma simplificada – com os detalhes já explicitados acima.

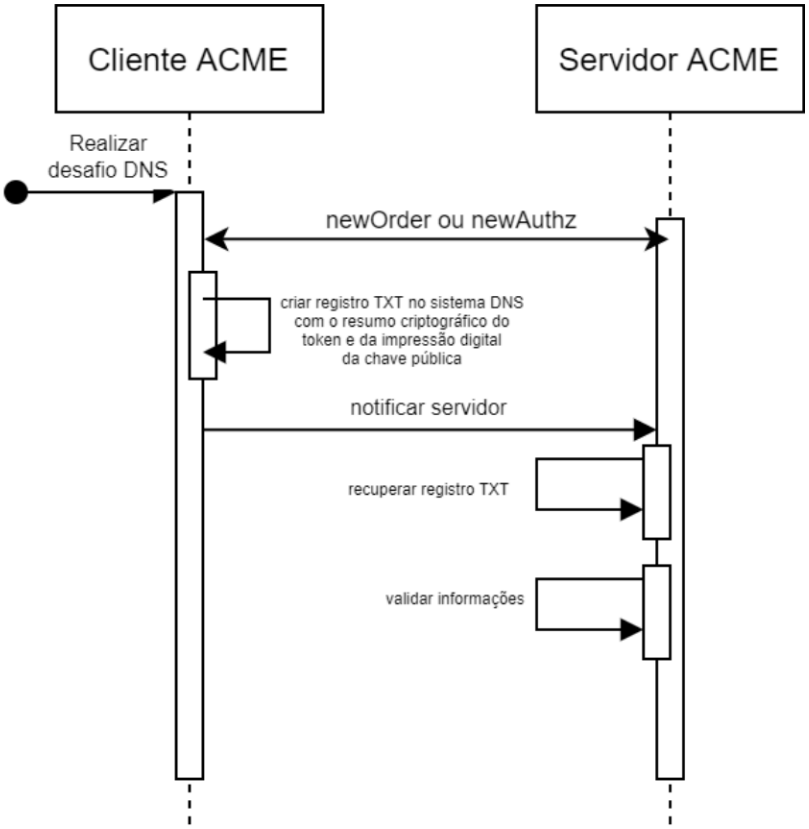

Figura 13. Desafio DNS ACME
Fonte: RFC 8555

Uma grande vantagem do uso desse desafio em relação ao HTTP vem do fato que este funciona bem mesmo se o cliente possuir vários servidores web.

3.5. Emissão de Certificados

Foi visto como o cliente ACME realiza autenticação com servidor e como ele pode provar o controle sobre algum domínio – por meio dos desafios. Dessa forma, demonstra-se – de maneira usual – os passos para a geração de um certificado utilizando o protocolo ACME.

Primeiramente, o cliente realiza a etapa de autenticação, estabelecendo um conta ACME com o servidor, como foi explicado anteriormente. Feito isso, o cliente então envia um POST para newOrder, para obter um certificado. Geralmente o caminho para essa requisição se encontra em */acme/newOrder*, conforme abaixo na Figura 14.

```
POST /acme/new-order HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "kid": "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "5XJ1L3lEkMG7tR6pA00clA",
        "url": "https://example.com/acme/new-order"
    }),
    "payload": base64url({
        "identifiers": [
            { "type": "dns", "value": "www.exemplo.org" },
            { "type": "dns", "value": "exemplo.org" }
        ],
        "notBefore": "2016-01-01T00:04:00+04:00",
        "notAfter": "2016-01-08T00:04:00+04:00"
    }),
    "signature": "H6ZXtGjTZyUnPeKn...wEA4TklBdh3e454g"
}
```

Figura 14. POST do cliente ACME *`/acme/new-order`*
Fonte: RFC 8555

Com esse POST o cliente requisita um certificado que demonstre o controle sob tais domínios explicitados no vetor *identifiers*. O servidor responde, mostrando, além de outras informações, um vetor de *authorizations*, ou seja, quais autorizações o cliente deve cumprir para demonstrar que possui controle sob tais domínios (um por domínio) e um endereço *finalize*, onde, após cumprir todas as autorizações, o cliente envia a CSR (requisição de certificado assinada), como pode ser visto à seguir na Figura 15.

```
HTTP/1.1 201 Created
Replay-Nonce: MYAuvOpaoIiywTezizk5vw
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/order/TOlocE8rfgo

{
    "status": "pending",
    "expires": "2016-01-05T14:09:07.99Z",
    "notBefore": "2016-01-01T00:00:00Z",
    "notAfter": "2016-01-08T00:00:00Z",
    "identifiers": [
        { "type": "dns", "value": "www.exemplo.org" },
        { "type": "dns", "value": "exemplo.org" }
    ],
    "authorizations": [
        "https://example.com/acme/authz/PAniVnsZcis",
        "https://example.com/acme/authz/r4HqLzrSrpI"
    ],
    "finalize": "https://example.com/acme/order/T...o/
       finalize"
}
```

Figura 15. Resposta ao POST pelo servidor ACME *`/acme/new-order`*
Fonte: RFC 8555

Após isso, o cliente ACME envia um POST (referenciado como POST-as-GET na [Barnes et al. 2019], uma maneira de mandar uma mensagem assinada, porém apenas com o intuito de recuperar informação) para o endereço de cada uma das autorizações, conforme abaixo na Figura 16.

```
POST /acme/authz/PAniVnsZcis HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "kid": "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
        "url": "https://example.com/acme/authz/PAniVnsZcis"
    }),
    "payload": "",
    "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
}
```

Figura 16. POST do cliente ACME *`/acme/authz`*
Fonte: RFC 8555

Para cada um desses POST-as-GET, o servidor envia, dentre outras informações, um vetor de *challenges* – como demonstrado na seção do desafio HTTP – para que o cliente escolha uma e a realize, conforme abaixo na Figura 17.

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "pending",
    "expires": "2016-01-02T14:09:30Z",
    "identifier": {
    "type": "dns",
    "value": "www.exemplo.org"
},
    "challenges": [
        {
        "type": "http-01",
        "url": "https://example.com/acme/chall/prV_B7
           yEyA4",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        },
        {
        "type": "dns-01",
        "url": "https://example.com/acme/chall/Rg5dV14
           Gh1Q",
        "token": "DGyRejmCefe7v4NfDGDKfA"
        }
    ]
}
```

Figura 17. Resposta ao POST pelo servidor ACME *`/acme/authz`*
Fonte: RFC 8555

Logo depois de realizar cada desafio, – como já demonstrado – o cliente envia um POST-as-GET para o endereço da autorização novamente. Dessa vez, o servidor disponibiliza outras informações a respeito do estado de cada desafio, conforme a seguir na Figura 18.

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://example.com/acme/directory>;rel="index"

{
    "status": "valid",
    "expires": "2018-09-09T14:09:01.13Z",
    "identifier": {
        "type": "dns",
        "value": "www.exemplo.org"
    },
    "challenges": [
        {
            "type": "http-01",
            "url": "https://example.com/acme/chall/p...
               4",
            "status": "valid",
            "validated": "2014-12-01T12:05:13.72Z",
            "token": "IlirfxKKXAsHtmzK29Pj8A"
        }
    ]
}
```

Figura 18. Resposta ao POST pelo servidor ACME *`/acme/authz`*
Fonte: RFC 8555

Dessa maneira, o cliente pode verificar se já cumpriu todas as autorizações necessárias. Após isso, o cliente envia a CSR para o caminho *finalize* – como já explicado. Como pode ser visto abaixo na Figura 19.

```
POST /acme/order/TOlocE8rfgo/finalize HTTP/1.1
Host: example.com
Content-Type: application/jose+json

{
    "protected": base64url({
        "alg": "ES256",
        "kid": "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "MSF2j2nawWHPxxkE3ZJtKQ",
        "url": "https://example.com/acme/order/T...o/finalize"
    }),
    "payload": base64url({
        "csr": "MIIBPTCBxAIBADBFMQ...FS6aKdZeGsysoCo4H9P",
    }),
    "signature": "uOrUfIIk5RyQ...nw62Ay1cl6AB"
}
```

Figura 19. POST do cliente ACME *`/acme/order`*
Fonte: RFC 8555

Nesse ponto, a AC pode não querer emitir o certificado caso haja algum problema com a CSR, por exemplo:

- Se a CSR e os identificadores do pedido forem diferentes
- Se a conta não for autorizada para os identificadores indicados na CSR
- Se a CSR solicitar extensões que a AC não deseja incluir

Caso a CSR esteja bem formada o servidor ACME retorna informações atualizadas à respeito do pedidor (Order), contendo o caminho para baixar o certificado no campo *certificate*, conforme abaixo na Figura 20.

```
HTTP/1.1 200 OK
Replay-Nonce: CGf81JWBsq8QyIgPCi9Q9X
Link: <https://example.com/acme/directory>;rel="index"
Location: https://example.com/acme/order/TOlocE8rfgo

{
    "status": "valid",
    "expires": "2016-01-20T14:09:07.99Z",
    "notBefore": "2016-01-01T00:00:00Z",
    "notAfter": "2016-01-08T00:00:00Z",
    "identifiers": [
        { "type": "dns", "value": "www.exemplo.org" },
        { "type": "dns", "value": "exemplo.org" }
    ],
    "authorizations": [
        "https://example.com/acme/authz/PAniVnsZcis",
        "https://example.com/acme/authz/r4HqLzrSrpI"
    ],
    "finalize": "https://example.com/acme/order/T...o/
      finalize",
    "certificate": "https://example.com/acme/cert/mAt3
      xBGaobw"
}
```

Figura 20. Resposta ao POST pelo servidor ACME *`/acme/order`*
Fonte: RFC 8555

Sendo assim, para baixar o certificado o cliente simplesmente envia um POST-as-GET para o endereço do campo *certificate*, como pode ser visto a seguir na Figura 21.

```
POST /acme/cert/mAt3xBGaobw HTTP/1.1
Host: example.com
Content-Type: application/jose+json
Accept: application/pem-certificate-chain

{
    "protected": base64url({
        "alg": "ES256",
        "kid": "https://example.com/acme/acct/evOfKhNU60wg",
        "nonce": "uQpSjlRb4vQVCjVYAyyUWg",
        "url": "https://example.com/acme/cert/mAt3xBGaobw"
    }),
    "payload": "",
    "signature": "nuSDISbWG8mMgE7H...QyVUL68yzf3Zawps"
}
```

Figura 21. POST do cliente ACME *acme/cert*
Fonte: RFC 8555

E, dessa maneira, o servidor ACME responde com o certificado emitido, conforme abaixo na Figura 22.

```
HTTP/1.1 200 OK
Content-Type: application/pem-certificate-chain
Link: <https://example.com/acme/directory>;rel="index"

-----BEGIN CERTIFICATE-----
[End-entity certificate contents]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[Issuer certificate contents]
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
[Other certificate contents]
-----END CERTIFICATE-----
```

Figura 22. Resposta ao POST pelo servidor ACME *acme/cert*
Fonte: RFC 8555

Nota-se que a resposta do servidor não contém apenas o certificado emitido, adicionalmente contém o certificado da AC que o emitiu e outros possíveis certificados que constroem a cadeia de certificação.

A Figura 23 ilustra de maneira bem simplificada a emissão de um certificado utilizando o protocolo ACME.

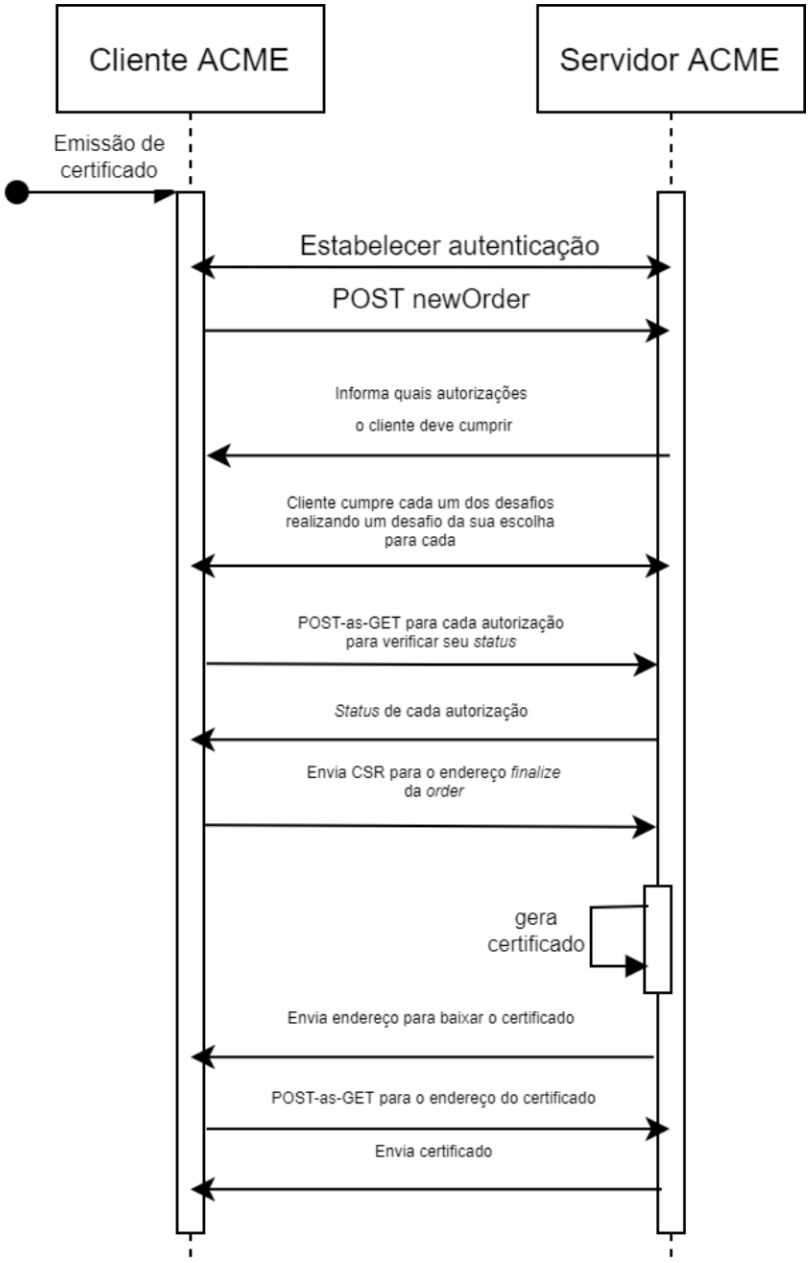

Figura 23. Emissão de Certificado ACME
Fonte: RFC 8555

3.6. Protocolo ACME Pós-Quântico

Atualmente, na versão 2, o protocolo requisita a implementação de algoritmos de curvas elípticas, ou seja, algoritmos de criptografia clássica para assinatura digital, como mencionado anteriormente. Além disso, o protocolo não restringe a implementação de algoritmos adicionais de criptografia, dessa forma, sendo abragente neste sentido.

Sendo assim, a única mudança necessária para torná-lo pós-quântico é a integração de algoritmos pós-quânticos de assinatura digital às implementações do protocolo. Esses algoritmos podem ser tanto utilizados para realizar a troca de mensagens do protocolo – as quais são assinadas digitalmente – quanto para a assinatura e emissão dos certificados

digitais, tornando-os pós-quânticos.

Dessa maneira, resta escolher quais algoritmos pós-quânticos serão integrados às implementações do protocolo – tanto no cliente quanto no servidor ACME.

4. Implementação

Para escolha de quais implementações serão modificadas considerou-se, principalmente, os aspectos de: simplicidade e compatibilidade entre cliente e servidor. Sendo assim, como tem-se muito menos variedade de implementações do servidor ACME, considerou-se essa primeiro. Entre as implementações, o servidor Pebble [Peb 2021] mostrou-se mais simples – feito para testes e prototipagem – sendo, dessa maneira, o escolhido. Já para a grande variedade de implementações cliente, considerou-se principalmente a compatibilidade com o servidor, i.e., implementados na mesma linguagem – neste caso, em Go – para que modificações em uma implementação sejam facilmente reproduzidas na outra. Dessa forma, escolheu-se a implementação LEGO [LEG 2021] para o cliente, sendo a principal implementação do cliente ACME em Go disponibilizada.

Além disso, contou-se com a biblioteca LibOQS [Lib 2021a] a qual oferece implementações de diversos algoritmos pós-quânticos participando do processo de padronização do NIST [Moody et al. 2020]. Dessa forma, a integração teve-se com alguns dos algoritmos já suportados nesta biblioteca a qual também oferece *bindings*, i.e., uma interface para utilizar funções escritas em uma linguagem em outra com facilidade – neste caso de C para Go. Outras bibliotecas em Go também tiveram que ser modificadas no processo, como será detalhado nas seções seguintes.

Neste trabalho optou-se por integrar os algoritmos de assinatura digital finalistas da terceira rodada do NIST de padronização, como já mencionado. Além desses, o algoritmo candidato SPHINCS+ também foi escolhido por cobrir casos de uso específicos, como IoT ou Internet das Coisas de maneira eficiente [Bernstein et al. 2019]. Foram escolhidos, para cada algoritmo, dois níveis de segurança para, dessa forma, ACs em diferentes níveis da cadeia de certificação terem diferentes níveis de segurança – esboçando um cenário mais realista. Sendo assim, os algoritmos escolhidos foram (segundo a nomenclatura da LibOQS):

- **Mais alto nível de segurança:**
  - dilithium5, dilithium5-aes
  - falcon-1024
  - rainbow-V-classic, rainbow-V-circumzenithal, rainbow-V-compressed
  - sphincs+-haraka-256s-simple, sphincs+-haraka-256f-simple, sphincs+-haraka-256s-robust, sphincs+-haraka-256f-robust, sphincs+-sha256-256s-simple, sphincs+-sha256-256f-simple, sphincs+-sha256-256s-robust, sphincs+-sha256-256f-robust, sphincs+-shake256-256s-simple, sphincs+-shake256-256f-simple, sphincs+-shake256-256s-robust, sphincs+-shake256-256f-robust
- **Mais baixo nível de segurança:**
  - dilithium2, dilithium2-aes
  - falcon-512
  - rainbow-I-classic, rainbow-I-circumzenithal, rainbow-I-compressed

– sphincs+-haraka-128s-simple, sphincs+-haraka-128f-simple, sphincs+-haraka-128s-robust, sphincs+-haraka-128f-robust, sphincs+-sha256-128s-simple, sphincs+-sha256-128f-simple, sphincs+-sha256-128s-robust, sphincs+-sha256-128f-robust, sphincs+-shake256-128s-simple, sphincs+-shake256-128f-simple, sphincs+-shake256-128s-robust, sphincs+-shake256-128f-robust

Todo o código também pode ser encontrado em [Pro 2022].

4.1. LibOQS

Como mencionado a LibOQS – *Library Open Quantum Safe* – oferece diversas implementações de algoritmos pós-quânticos, dos quais concentra-se os esforços neste trabalho nos algoritmos de assinatura digital.

Seguindo a própria descrição do projeto OQS, tem-se que o objetivo do mesmo é apoiar o desenvolvimento e a prototipagem de criptografia pós-quântica.

O projeto consiste em duas linhas principais de trabalho – destinando-se à prototipagem e avaliação da criptografia pós-quântica: LibOQS, uma biblioteca em C de código aberto para algoritmos criptográficos pós-quânticos e integrações de protótipos em protocolos e aplicativos, incluindo um *fork*, i.e., uma cópia seguida de modificações, da amplamente usada biblioteca OpenSSL.

Neste trabalho utilizou-se da biblioteca LibOQS para realizar a integração com os outros pacotes modificados. Mais especificamente, as *bindings* da LibOQS [Lib 2021b] – em C – para a linguagem Go, como será visto na seção seguinte.

4.2. Modificações na Standard Golang Library

Standard Golang Library ou a biblioteca padrão da linguagem Go [Gol 2021] à qual mantém diversos componentes bases utilizados por outras bibliotecas de terceiros. Ela oferece desde funcionalidades de sistemas operacionais até pacotes de testes automatizados. Contudo, concentra-se, nesse trabalho, no pacote **crypto**. Esse oferece diversas funcionalidades à respeito de funções criptográficas assimétricas – como já visto na Seção 2 –, simétricas, constantes, etc.

Criou-se uma interface para trabalhar com a *bindings* da LibOQS para as funções da assinatura digital, concentrando no arquivo **pqc.go**. Dessa forma, implementou-se tanto a interface de **chave pública** quanto de **chave privada** e suas respectivas funções.

```
1    // PublicKey represents an PQC public key.
2    type PublicKey struct {
3        AlgName string
4        Bytes   []byte
5    }
```

Figura 24. Estrutura da chave pública
Fonte: Original

No código a cima da Figura 24, tem-se a estrutura da **chave pública** pós-quântica genérica, i.e., pra qualquer algoritmo pós-quântico de assinatura digital oferecido pela

biblioteca da LibOQS. Dessa forma, tem-se o nome do algoritmo a qual a chave pública está relacionada e seus *bytes* propriamente.

```
1    // PrivateKey represents an PQC private key.
2    type PrivateKey struct {
3        PublicKey
4        Signer oqs.Signature
5    }
```

Figura 25. Estrutura da chave privada
Fonte: Original

Neste último código, ilustrado na Figura 25, tem-se, analogamente, a estrutura da **chave privada**. Contém-se a chave pública dentro da estrutura de chave privada para fins de recuperar a chave pública facilmente. Além disso, tem-se a estrutura **Signer** do signatário – que entre outras informações, contém os *bytes* da chave privada –, pois a estrutura é utilizada apenas para fins de assinatura digital, não sendo necessárias outras informações.

Em relação as funções que a estrutura da chave pública implementa, tem-se a verificação de igualdade entre duas chaves públicas e a verificação de uma assinatura.

```
1    // Equal reports whether public and x have the same value.
2    func (pub *PublicKey) Equal(x crypto.PublicKey) bool {
3        xx, ok := x.(*PublicKey)
4        if !ok {
5            return false
6        }
7        return bytes.Equal(pub.Bytes, xx.Bytes)
8        && pub.AlgName == xx.AlgName
9    }
```

Figura 26. Função de verificação de igualdade de chaves públicas
Fonte: Original

No código da Figura 26, tem-se a verificação de igualdade entre duas chaves públicas. Para isso, verifica-se se os *bytes* das duas chaves são iguais e se seus algoritmos são os mesmos.

```
1     // Verifies the signature.
2     func (pub *PublicKey) Verify(data, signature []byte) bool {
3         verifier := oqs.Signature{}
4
5         if err := verifier.Init(pub.AlgName, nil); err != nil {
6             return false
7         }
8
9         isValid, err := verifier.Verify(data, signature, (*pub).Bytes)
10        if err != nil {
11            return false
12        }
13        return isValid
14    }
```

Figura 27. Função de verificação da assinatura
Fonte: Original

Nesta outra função (Figura 27), tem-se a verificação de uma assinatura digital, utilizando a chave pública para tal. Verifica-se a assinatura – juntamente com a informação assinada – criando-se uma estrutura **Signature** com o algoritmo da chave pública. A partir disso, é chamado o método **Verify** da LibOQS, o qual recebe as informações assinadas, a assinatura e a chave pública. Caso a assinatura seja válida é retornado o valor **true** – verdadeiro –, caso contrário o valor **false** – falso.

Já para a chave privada, tem-se uma variedade maior de métodos implementados. Conta-se com métodos para: verificar a igualdade entre duas chaves privadas – similarmente ao da chave pública –, para recuperar a chave pública a partir da privada, e para assinar.

```
1     // Equal reports whether private and x have the same value.
2     func (priv *PrivateKey) Equal(x crypto.PrivateKey) bool {
3         xx, ok := x.(*PrivateKey)
4         if !ok {
5             return false
6         }
7         return priv.PublicKey.Equal(&xx.PublicKey)
8         && bytes.Equal(priv.Signer.ExportSecretKey(),
9         xx.Signer.ExportSecretKey())
10    }
```

Figura 28. Função de verificação de igualdade de chaves privadas
Fonte: Original

Neste código da Figura 28, tem-se a verificação de igualdade entre duas chaves privadas. Para isso, duas chaves privadas são ditas iguais se as suas chaves públicas são iguais – como visto anteriormente – e se os bytes das chaves privadas são iguais.

```
1     // Public returns the public key corresponding to private key.
2     func (priv *PrivateKey) Public() crypto.PublicKey {
3         return &priv.PublicKey
4     }
5
6     func (priv *PrivateKey) PQCPublic() *PublicKey {
7         return priv.Public().(*PublicKey)
8     }
```

Figura 29. Funções de recuperação da chave pública
Fonte: Original

Tem-se, também, dois métodos para recuperar a chave pública na Figura 29. O primeiro genérico do pacote **crypto**. Já o segundo, recupera a chave pública e realiza um *cast*, i.e., uma transformação, para a chave pública pós-quântica. Isso foi realizado apenas por fins de simplificar o código, já que na maioria dos casos quer-se utilizar a chave pública pós-quântica.

```
1     /* Signs data and returns the signature.
2     rand and opts are not being used.
3     */
4     func (priv *PrivateKey) Sign(rand io.Reader,
5     data []byte, opts crypto.SignerOpts) ([]byte, error) {
6         signature, err := priv.Signer.Sign(data)
7         if err != nil {
8             return nil, err
9         }
10        return signature, nil
11    }
```

Figura 30. Função de assinatura
Fonte: Original

Neste último método da chave privada (Figura 30), tem-se a realização da assinatura digital. Por questões de compatibilidade com a interface de chave privada do pacote **crypto** precisa-se de parâmetros a mais que, neste caso, não são utilizados, como o *rand* e o *opts*. A assinatura é realizada utilizando o método **Sign** da LibOQS a partir do **Signer** da estrutura de chave privada implementada, dessa maneira, recebendo as informações que se deseja assinar. O retorno pode ser uma assinatura nula e um erro caso haja algum problema ou pode ser a assinatura com sucesso sem nenhum erro – um erro nulo.

Além dos métodos implementados para as estruturas de chave pública e privada, tem-se também o método para gerá-las, i.e., o método que gera a chave privada – a qual já contém a pública, como pode ser visto na Figura 31.

```
1     // Generates a key pair and returns the private key.
2     func GenerateKey(signatureName string) (*PrivateKey, error) {
3         signer := oqs.Signature{}
4         if err := signer.Init(signatureName, nil); err != nil {
5             return nil, err
6         }
7
8         pubKeyBytes, err := signer.GenerateKeyPair()
9         if err != nil {
10            return nil, err
11        }
12
13        privKey := new(PrivateKey)
14        privKey.AlgName = signatureName
15        privKey.Bytes = pubKeyBytes
16        privKey.Signer = signer
17
18        return privKey, nil
19    }
```

Figura 31. Função de geração de par de chaves
Fonte: Original

Para gerar, recebe-se como parâmetro o algoritmo o qual será instânciado a chave privada. Dessa forma, instância-se o **Signer** a partir do método **Init** da LibOQS, recebendo o algoritmo pós-quântico. Feito isso, gera-se o par de chaves de fato a partir do método **GenerateKeyPair** da LibOQS, retornando os *bytes* da chave pública. E, por fim, atribui-se cada um desses valores – algoritmo utilizado, bytes da chave pública e estrutura para assinar – aos da estrutura da chave privada.

E, por fim, são integrados os OIDs [OID 2022] dos algoritmos pós-quânticos e métodos para recuperar o nome do algoritmo a partir do OID e o OID a partir do nome do algoritmo. Como os OIDs não são oficiais, são utilizados aqueles sugeridos pela própria LibOQS, podendo ser encontrados no *github* deles [Lib 2022]. Por conta da extensibilidade, preferiu-se não mostrar em texto esta parte.

Além disso, outro importante arquivo modificado foi **x509.go**. Este implementa código referente aos certificados **X.509** [Cooper et al. 2008b] – a principal versão de certificados utilizada. No mesmo foram modificadas as partes do código necessárias para integrar com a inteface pós-quântica criada.

As modificações podem ser como: modificações em funções para adicionar os casos da interface pós-quântica – tipicamente uma estrutura *switch-case* – e modificações para adicionar constantes e variáveis da interface pós-quântica, para manter a estrutura do código sem complexidades adicionais desnecessárias.

```
1     func marshalPublicKey(pub interface{}) (publicKeyBytes []byte,
2     publicKeyAlgorithm pkix.AlgorithmIdentifier, err error) {
3         switch pub := pub.(type) {
4         ...
5         case *pqc.PublicKey:
6             publicKeyBytes = pub.Bytes
7             if err != nil {
8                 return nil, pkix.AlgorithmIdentifier{}, err
9             }
10            publicKeyAlgorithm.Algorithm =
11            pqc.GetPublicKeyOIDFromPublicKey(pub.AlgName)
12        default:
13            return nil, pkix.AlgorithmIdentifier{}, fmt.Errorf("x509:
14            unsupported public key type: %T", pub)
15        }
16        return publicKeyBytes, publicKeyAlgorithm, nil
17    }
18
19    ...
20
21    func getPublicKeyAlgorithmFromOID(oid asn1.ObjectIdentifier)
22    PublicKeyAlgorithm {
23        switch {
24        ...
25        case oid.Equal(oidPublicKeyRainbowVCompressed):
26            return RainbowVCompressed
27        ...
28        case oid.Equal(oidPublicKeyDilithium2AES):
29            return Dilithium2AES
30        ...
31        case oid.Equal(oidPublicKeySphincsPlusSHAKE256128fRobust):
32            return SphincsPlusSHAKE256128fRobust
33        }
34        return UnknownPublicKeyAlgorithm
35    }
```

Figura 32. Novos casos para funções existentes
Fonte: Original

```
1   func checkSignature(algo SignatureAlgorithm, signed,
2   signature []byte, publicKey crypto.PublicKey) (err error) {
3       ...
4       switch pub := publicKey.(type) {
5       case *pqc.PublicKey:
6           if !pub.Verify(signed, signature) {
7               return errors.New("x509: " + pub.AlgName
8               + "verification failure")
9           }
10          return
11      }
12      ...
13      return ErrUnsupportedAlgorithm
14  }
15
16  func signingParamsForPublicKey(pub interface{}, requestedSigAlgo
17  SignatureAlgorithm) (hashFunc crypto.Hash, sigAlgo
18  pkix.AlgorithmIdentifier, err error) {
19      var pubType PublicKeyAlgorithm
20
21      switch pub := pub.(type) {
22      case *pqc.PublicKey:
23          switch pub.AlgName {
24          case "dilithium5":
25              pubType = Dilithium5
26              sigAlgo.Algorithm = oidSignatureDilithium5
27          ...
28          case "sphincs+-sha256-256s-robust":
29              pubType = SphincsPlusSHA256256sRobust
30              sigAlgo.Algorithm =
31              oidSignatureSphincsPlusSHA256256sRobust
32          ...
33          case "dilithium2":
34              pubType = Dilithium2
35              sigAlgo.Algorithm = oidSignatureDilithium2
36          ...
37          case "sphincs+-shake256-128f-robust":
38              pubType = SphincsPlusSHAKE256128fRobust
39              sigAlgo.Algorithm =
40              oidSignatureSphincsPlusSHAKE256128fRobust
41          }
42      ...
43      return
44  }
```

Figura 33. Novos casos para funções existentes
Fonte: Original

Estes primeiros métodos, como vistos na Figura 32 e na Figura 33, se encaixam nos quais apenas foram necessários criar novos casos para a interface pós-quântica implementada, sem muito esforço adicional. Percebe-se que foram omitidas algumas partes para não estender demais, porém sem perda didática.

```
type SignatureAlgorithm int

const (
    UnknownSignatureAlgorithm SignatureAlgorithm = iota
    ...
    PureDilithium5
    ...
    PureRainbowIClassic
    ...
    PureSphincsPlusSHAKE256128fRobust
)

...

type PublicKeyAlgorithm int

const (
    UnknownPublicKeyAlgorithm PublicKeyAlgorithm = iota
    ...
    Falcon1024
    ...
    RainbowICompressed
    SphincsPlusSHAKE256128sSimple
    ...
)
```

Figura 34. Modificações em constantes e seus usos
Fonte: Original

```
    var publicKeyAlgoName = [...]string{
        ...
        RainbowVClassic:                  "Rainbow-V-Classic",
        RainbowVCircumzenithal:           "Rainbow-V-Circumzenithal",
        ...
        Dilithium2:                       "Dilithium2",
        ...
        SphincsPlusSHAKE256128fRobust: "SPHINCS+-SHAKE256-128f-Robust",
    }

    ...

    var (
        ...
        oidSignatureDilithium5                        =
        pqc.OIDSignatureDilithium5
        oidSignatureDilithium5AES                     =
        pqc.OIDSignatureDilithium5AES
        oidSignatureFalcon1024                        =
        pqc.OIDSignatureFalcon1024
        ...
        oidSignatureSphincsPlusSHAKE256128fSimple =
        pqc.OIDSignatureSphincsPlusSHAKE256128fSimple
        ...
    )

    ...

    var signatureAlgorithmDetails = []struct {
        algo       SignatureAlgorithm
        name       string
        oid        asn1.ObjectIdentifier
        pubKeyAlgo PublicKeyAlgorithm
        hash       crypto.Hash
    }{
        ...
        {PureFalcon1024, "Falcon-1024", oidSignatureFalcon1024,
        Falcon1024, crypto.Hash(0)},
        ...
        {PureSphincsPlusSHAKE256128fRobust,
        "SPHINCS+-SHAKE256-128f-Robust",
        oidSignatureSphincsPlusSHAKE256128fRobust,
        SphincsPlusSHAKE256128fRobust, crypto.Hash(0)},
    }
```

Figura 35. Modificações em constantes e seus usos
Fonte: Original

Nesta segunda parte (Figura 34 e Figura 35), as principais modificações foram em constantes e seus usos. Percebe-se, além da criação de novas contantes para os algoritmos integrados, seus OIDs e a união de informações sobre os respectivos algoritmos como explicitado na última variável acima **signatureAlgorithmDetails**. Fora as omissões, estas foram as modificações neste arquivo.

Outro arquivo modificado na biblioteca padrão do Go foi **pkcs8.go**. Este arquivo representa uma chave pública no padrão PKCS8 [Kaliski 2008] – um padrão para guardar informações da chave privada. Foi adicionado um atributo a estrutura pkcs8 que representa os *bytes* da chave pública correspondente a chave privada. Isso foi necessário para poder recuperar totalmente a chave privada a partir dos bytes dessa estrutura nos métodos modificados.

```
1   func ParsePKCS8PrivateKey(der []byte)
2   (key interface{}, err error) {
3       var privKey pkcs8
4       if _, err := asn1.Unmarshal(der, &privKey); err != nil {
5           if _, err := asn1.Unmarshal(der, &ecPrivateKey{});
6           err == nil
7           {
8               return nil, errors.New("x509: failed to
9               parse private key
10              (use ParseECPrivateKey instead for this key format)")
11          }
12          if _, err := asn1.Unmarshal(der, &pkcs1PrivateKey{});
13          err == nil {
14              return nil, errors.New("x509: failed to
15              parse private key
16              (use ParsePKCS1PrivateKey instead
17              for this key format)")
18          }
19          return nil, err
20      }
21      switch {
22      case privKey.Algo.Algorithm.Equal(oidPublicKeyRSA):
23          key, err = ParsePKCS1PrivateKey(privKey.PrivateKey)
24          if err != nil {
25              return nil, errors.New("x509: failed to
26              parse RSA private
27              key embedded in PKCS#8: " + err.Error())
28          }
29          return key, nil
30      ...
31      case privKey.Algo.Algorithm.Equal(
32      oidPublicKeySphincsPlusSHA256128sSimple):
33          algName := pqc.GetPublicKeyFromPublicKeyOID(
34          oidPublicKeySphincsPlusSHA256128sSimple)
35          key := pqc.PrivateKey{
36              PublicKey: pqc.PublicKey{
37                  Bytes:    privKey.PublicKey,
38                  AlgName: algName,
39              },
40          }
41          key.Signer.Init(algName, privKey.PrivateKey)
42          return key, nil
43      ...
44  }
```

Figura 36. Funções PKCS8
Fonte: Original

```
1   func MarshalPKCS8PrivateKey(key interface{}) ([]byte, error) {
2       var privKey pkcs8
3
4       switch k := key.(type) {
5       case *rsa.PrivateKey:
6           privKey.Algo = pkix.AlgorithmIdentifier{
7               Algorithm:  oidPublicKeyRSA,
8               Parameters: asn1.NullRawValue,
9           }
10          privKey.PrivateKey = MarshalPKCS1PrivateKey(k)
11      case *pqc.PrivateKey:
12          privKey.Algo = pkix.AlgorithmIdentifier{
13              Algorithm:
14                  pqc.GetPublicKeyOIDFromPublicKey(k.AlgName),
15              Parameters: asn1.NullRawValue,
16          }
17          privKey.PublicKey = k.Bytes
18          privKey.PrivateKey = k.Signer.ExportSecretKey()
19      ...
20      return asn1.Marshal(privKey)
21  }
```

Figura 37. Funções PKCS8
Fonte: Original

Estas duas funções da Figura 36 e da Figura 37 foram as únicas modificadas no arquivo – nenhuma nova foi adicionada. Na função **ParsePKCS8PrivateKey**, recupera-se a chave privada a partir de *bytes* da mesma – já serializados. Adiciona-se um caso para cada algoritmo pós-quântico integrado. Esses *bytes* os quais recebe-se nessa função são gerados a partir do método **MarshalPKCS8PrivateKey**, o qual recebe a chave privada e retorna os *bytes* referentes a ela. Para isso, criou-se um caso para a interface pós-quântica a ser tratado.

E, por fim, modificou-se o arquivo **parser.go**. Este arquivo lida com a mudança na representação de informações, neste caso, principalmente, em relação as chaves criptográficas.

```
1     func parsePublicKey(algo PublicKeyAlgorithm,
2     keyData *publicKeyInfo) (interface{}, error) {
3         der := cryptobyte.String(keyData.PublicKey.RightAlign())
4         switch algo {
5         ...
6         case Dilithium5:
7             pub := &pqc.PublicKey{
8                 Bytes:   keyData.PublicKey.Bytes,
9                 AlgName: "dilithium5",
10            }
11            return pub, nil
12        ...
13        case RainbowIClassic:
14            pub := &pqc.PublicKey{
15                Bytes:   keyData.PublicKey.Bytes,
16                AlgName: "rainbow-i-classic",
17            }
18            return pub, nil
19        ...
20        case SphincsPlusSHAKE256128fRobust:
21            pub := &pqc.PublicKey{
22                Bytes:   keyData.PublicKey.Bytes,
23                AlgName: "sphincs+-shake256-128f-robust",
24            }
25            return pub, nil
26        ...
27        }
28    }
```

Figura 38. Função parsePublicKey
Fonte: Original

Modificou-se apenas a função **parsePublicKey** (Figura 38) neste arquivo. A qual a partir de informações da chave pública a transforma em uma inteface mais genérica. Para isso, foi necessário adicionar os casos dos novos algoritmos integrados.

4.3. Go JOSE

A biblioteca Go JOSE é uma implementação em Go do conjunto de padrões *Javascript Object Signing and Encryption* ou Assinatura e Criptografia de Objetos *Javascript*. Isso inclui suporte aos padrões *JSON Web Encryption* [Jones and Hildebrand 2015], *JSON Web Signature* [Jones et al. 2015a], and *JSON Web Token* [Jones et al. 2015b].

Esse pacote tem uma grande importância no trabalho, pois as mensagens trocadas entre o cliente e o servidor ACME são mensagens JSON. Além disso, essas mensagens são assinadas e requerem a integração do suporte aos algoritmos pós-quânticos para que, dessa maneira, o protocolo ACME se torne pós-quântico. Logo, além de utilizar a versão modificada a cima da biblioteca padrão do Go, também alterou-se esta.

Primeiramente, tem-se o arquivo **cryptosigner.go** o qual lida com a assinatura de dados no formato JSON. Neste caso, adicionou-se o necessário para integrar com os algoritmos pós-quânticos, como pode ser visto na Figura 39 e na Figura 40.

```
1     func (s *cryptoSigner) Algs() []jose.SignatureAlgorithm {
2         switch s.signer.Public().(type) {
3         ...
4         case *pqc.PublicKey:
5             return []jose.SignatureAlgorithm{jose.Dilithium5,
6             jose.Dilithium5AES, jose.Falcon1024, jose.RainbowVClassic,
7                 jose.RainbowVCircumzenithal, jose.RainbowVCompressed,
8                 jose.SphincsPlusHaraka256sSimple,
9                 jose.SphincsPlusHaraka256fSimple,
10                jose.SphincsPlusHaraka256sRobust,
11                jose.SphincsPlusHaraka256fRobust,
12                jose.SphincsPlusSHA256256fSimple,
13                jose.SphincsPlusSHA256256sSimple,
14                jose.SphincsPlusSHA256256sRobust,
15                jose.SphincsPlusSHA256256fRobust,
16                jose.SphincsPlusSHAKE256256sSimple,
17                jose.SphincsPlusSHAKE256256fSimple,
18                jose.SphincsPlusSHAKE256256sRobust,
19                jose.SphincsPlusSHAKE256256fRobust, jose.Dilithium2,
20                jose.Dilithium2AES, jose.Falcon512,
21                jose.RainbowIClassic,
22                jose.RainbowICircumzenithal, jose.RainbowICompressed,
23                jose.SphincsPlusHaraka128sSimple,
24                jose.SphincsPlusHaraka128fSimple,
25                jose.SphincsPlusHaraka128sRobust,
26                jose.SphincsPlusHaraka128fRobust,
27                jose.SphincsPlusSHA256128fSimple,
28                jose.SphincsPlusSHA256128sSimple,
29                jose.SphincsPlusSHA256128sRobust,
30                jose.SphincsPlusSHA256128fRobust,
31                jose.SphincsPlusSHAKE256128sSimple,
32                jose.SphincsPlusSHAKE256128fSimple,
33                jose.SphincsPlusSHAKE256128sRobust,
34                jose.SphincsPlusSHAKE256128fRobust}
35        default:
36            return nil
37        }
38    }
```

Figura 39. Funções modificadas do arquivo cryptosigner.go
Fonte: Original

```
1     func (s *cryptoSigner) SignPayload(payload []byte, alg
2     jose.SignatureAlgorithm) ([]byte, error) {
3         var hash crypto.Hash
4         switch alg {
5         ...
6         case jose.Dilithium5, jose.Dilithium5AES, jose.Falcon1024,
7         jose.RainbowVClassic,
8             jose.RainbowVCircumzenithal, jose.RainbowVCompressed,
9             jose.SphincsPlusHaraka256sSimple,
10            jose.SphincsPlusHaraka256fSimple,
11            jose.SphincsPlusHaraka256sRobust,
12            jose.SphincsPlusHaraka256fRobust,
13            jose.SphincsPlusSHA256256fSimple,
14            jose.SphincsPlusSHA256256sSimple,
15            jose.SphincsPlusSHA256256sRobust,
16            jose.SphincsPlusSHA256256fRobust,
17            jose.SphincsPlusSHAKE256256sSimple,
18            jose.SphincsPlusSHAKE256256fSimple,
19            jose.SphincsPlusSHAKE256256sRobust,
20            jose.SphincsPlusSHAKE256256fRobust, jose.Dilithium2,
21            jose.Dilithium2AES, jose.Falcon512, jose.RainbowIClassic,
22            jose.RainbowICircumzenithal, jose.RainbowICompressed,
23            jose.SphincsPlusHaraka128sSimple,
24            jose.SphincsPlusHaraka128fSimple,
25            jose.SphincsPlusHaraka128sRobust,
26            jose.SphincsPlusHaraka128fRobust,
27            jose.SphincsPlusSHA256128fSimple,
28            jose.SphincsPlusSHA256128sSimple,
29            jose.SphincsPlusSHA256128sRobust,
30            jose.SphincsPlusSHA256128fRobust,
31            jose.SphincsPlusSHAKE256128sSimple,
32            jose.SphincsPlusSHAKE256128fSimple,
33            jose.SphincsPlusSHAKE256128sRobust,
34            jose.SphincsPlusSHAKE256128fRobust:
35            return s.signer.Sign(nil, payload, nil)
36        default:
37            return nil, jose.ErrUnsupportedAlgorithm
38        }
39        ...
40        return out, err
41    }
```

Figura 40. Funções modificadas do arquivo cryptosigner.go
Fonte: Original

Na função **Algs**, a qual retorna uma lista de algoritmos de assinatura, adicionou-se os casos para os algoritmos de assinatura suportados pela interface pós-quântica. Já no método **SignPayload**, onde um array de *bytes* é assinado pelo algoritmo de assinatura passado como parâmetro, adicionou-se o caso onde é utilizado algum dos algoritmos integrados.

Outro arquivo modificado foi **asymmetric.go** – Figura 41, 42 e 43 – o qual trata da implementação dos padrões de criptografia assimétrica.

```
    // A generic PQC-based encrypter/verifier
    type pqcEncrypterVerifier struct {
        publicKey *pqc.PublicKey
    }

    ...

    // A generic PQC-based decrypter/signer
    type pqcDecrypterSigner struct {
        privateKey *pqc.PrivateKey
    }

    ...

    // newPQCRecipient creates recipientKeyInfo based on the given key.
    func newPQCRecipient(keyAlg KeyAlgorithm, publicKey *pqc.PublicKey)
    (recipientKeyInfo, error) {
        if publicKey == nil {
            return recipientKeyInfo{}, errors.New("invalid public key")
        }

        return recipientKeyInfo{
            keyAlg: keyAlg,
            keyEncrypter: &pqcEncrypterVerifier{
                publicKey: publicKey,
            },
        }, nil
    }
```

Figura 41. Funções do arquivo asymmetric.go
Fonte: Original

```
1     // newPQCSigner creates a recipientSigInfo based on the given key.
2     func newPQCSigner(sigAlg SignatureAlgorithm, privateKey
3     *pqc.PrivateKey) (recipientSigInfo, error) {
4         if privateKey == nil {
5             return recipientSigInfo{},
6             errors.New("invalid private key")
7         }
8
9         return recipientSigInfo{
10            sigAlg: sigAlg,
11            publicKey: staticPublicKey(&JSONWebKey{
12                Key: privateKey.Public(),
13            }),
14            signer: &pqcDecrypterSigner{
15                privateKey: privateKey,
16            },
17        }, nil
18    }
19
20    ...
21
22    // Encrypt the given payload and update the object.
23    func (ctx pqcEncrypterVerifier) encryptKey(cek []byte, alg
24    KeyAlgorithm) (recipientInfo, error) {
25        encryptedKey, err := ctx.encrypt(cek, alg)
26        if err != nil {
27            return recipientInfo{}, err
28        }
29
30        return recipientInfo{
31            encryptedKey: encryptedKey,
32            header:       &rawHeader{},
33        }, nil
34    }
```

Figura 42. Funções do arquivo asymmetric.go
Fonte: Original

```
    func (ctx pqcEncrypterVerifier) encrypt(cek []byte,
    alg KeyAlgorithm) ([]byte, error) {
        return nil, ErrUnsupportedAlgorithm
    }

    ...

    // Decrypt the given payload and return the content encryption key.
    func (ctx pqcDecrypterSigner) decryptKey(headers rawHeader,
    recipient *recipientInfo, generator keyGenerator)
    ([]byte, error) {
        return ctx.decrypt(recipient.encryptedKey,
        headers.getAlgorithm(), generator)
    }

    ...

    func (ctx pqcDecrypterSigner) decrypt(jek []byte, alg KeyAlgorithm,
    generator keyGenerator) ([]byte, error) {
        return nil, ErrUnsupportedAlgorithm
    }

    ...

    // Sign the given payload
    func (ctx pqcDecrypterSigner) signPayload(payload []byte, alg
    SignatureAlgorithm) (Signature, error) {
        out, _ := ctx.privateKey.Sign(nil, payload, nil)

        return Signature{
            Signature: out,
            protected: &rawHeader{},
        }, nil
    }

    ...

    // Verify the given payload
    func (ctx pqcEncrypterVerifier) verifyPayload(payload []byte,
    signature []byte, alg SignatureAlgorithm) error {
        if ctx.publicKey.Verify(payload, signature) {
            return nil
        }
        return ErrUnsupportedAlgorithm
    }
```

Figura 43. Funções do arquivo asymmetric.go
Fonte: Original

Criou-se os tipos **pqcEncrypterVerifier**, o qual guarda uma chave pública pós-quântica, assim como **pqcDecrypterSigner**, o qual guarda uma chave privada pós-quântica. O primeiro serve para realizar operações de cifragem e assinatura e o segundo operações de decifragem e verificação de assinaturas. No caso deste trabalho, só utilizou-

se as operações relacionadas a assinatura.

Implementou-se o método **newPQCRecipient**, o qual cria uma estrutura para guardar informações adicionais da chave e do tipo **pqcEncrypterVerifier**. Dessa forma, também criou-se o método **newPQCSigner**, o qual guarda informações adicionais da chave privada e do tipo **pqcDecrypterSigner**.

Implementou-se, apenas por completude, os métodos **encrypt**, **encryptKey**, **decrypt** e **decryptKey**, pois não utilizou-se operações de cifragem na implementação, apenas de assinatura digital.

Implementou-se também o método **signPayload** para realizar a assinatura de um vetor de *bytes* utilizando a chave privada pós-quântica contida no tipo **pqcDecryptSigner**. Assim, também foi implementado o método **verifyPayload**, o qual verifica se uma dada assinatura de um vetor de *bytes* está correta.

Também modificou-se o arquivo **crypter.go** que trata da cifragem de mensagens JSON e operações relacionadas que foram necessárias mudanças para completar a integração pós-quântica, como pode ser visto na Figura 44.

```
1     func makeJWERecipient(alg KeyAlgorithm,
2     encryptionKey interface{}) (recipientKeyInfo, error) {
3         switch encryptionKey := encryptionKey.(type) {
4         case *pqc.PublicKey:
5             return newPQCRecipient(alg, encryptionKey)
6         ...
7         return recipientKeyInfo{}, ErrUnsupportedKeyType
8     }
9
10    ...
11
12    // newDecrypter creates an appropriate decrypter
13    // based on the key type
14    func newDecrypter(decryptionKey interface{})
15    (keyDecrypter, error) {
16        switch decryptionKey := decryptionKey.(type) {
17        case *pqc.PrivateKey:
18            return &pqcDecrypterSigner{
19                privateKey: decryptionKey,
20            }, nil
21        ...
22        return nil, ErrUnsupportedKeyType
23    }
```

Figura 44. Funções do arquivo crypter.go
Fonte: Original

No método **makeJWERecipient** adicionou-se o caso onde cobre a chave pública pós-quântica, utilizando outro método implementado**newPQCRecipient** já mencionado. O mesmo acontece no método **newDecrypter**, onde adicionou-se um caso para a chave privada pós-quântica.

Outro arquivo modificado foi **jwk.go** – Figura 45, 46, 47 e 48 – o qual trata de

JSON Web Key [Jones 2015b] e sua manipulação. Ou seja, como são representadas as chaves para trabalhar em cima de dados JSON e operações à respeito das mesmas.

```
1     // rawJSONWebKey represents a public or private key in JWK format,
2     // used for parsing/serializing.
3     type rawJSONWebKey struct {
4         ...
5         PQCPub  *byteBuffer `json:"pqcpub,omitempty"`
6         PQCPriv *byteBuffer `json:"pqcpriv,omitempty"`
7         ...
8     }
9
10    ...
11
12    // MarshalJSON serializes the given key to its JSON representation.
13    func (k JSONWebKey) MarshalJSON() ([]byte, error) {
14        var raw *rawJSONWebKey
15        var err error
16
17        switch key := k.Key.(type) {
18        case *pqc.PublicKey:
19            raw = fromPQCPublicKey(key)
20        case *pqc.PrivateKey:
21            raw, err = fromPQCPrivateKey(key)
22        ...
23        }
24
25        if err != nil {
26            return nil, err
27        }
28
29        raw.Kid = k.KeyID
30        raw.Alg = k.Algorithm
31        raw.Use = k.Use
32
33        ...
34
35        return json.Marshal(raw)
36    }
```

Figura 45. Modificações do arquivo jwk.go
Fonte: Original

```
1     // UnmarshalJSON reads a key from its JSON representation.
2     func (k *JSONWebKey) UnmarshalJSON(data []byte) (err error) {
3         var raw rawJSONWebKey
4         err = json.Unmarshal(data, &raw)
5         if err != nil {
6             return err
7         }
8
9         ...
10
11        switch raw.Kty {
12        case "dilithium5", "dilithium5-aes", "falcon-1024",
13        "rainbow-v-classic", "rainbow-v-circumzenithal",
14            "rainbow-v-compressed", "sphincs+-haraka-256s-simple",
15            "sphincs+-haraka-256f-simple",
16            "sphincs+-haraka-256s-robust",
17            "sphincs+-haraka-256f-robust",
18            "sphincs+-sha256-256s-simple",
19            "sphincs+-sha256-256f-simple",
20            "sphincs+-sha256-256s-robust",
21            "sphincs+-sha256-256f-robust",
22            ..., "sphincs+-shake256-128f-robust":
23            if raw.PQCPriv != nil {
24                key, err = raw.pqcPrivateKey()
25                if err == nil {
26                    keyPub = key.(*pqc.PrivateKey).Public()
27                }
28            } else {
29                key, err = raw.pqcPublicKey()
30                keyPub = key
31            }
32        ...
33
34        return
35    }
```

Figura 46. Modificações do arquivo jwk.go
Fonte: Original

```
        const pqcThumbprintTemplate = `{"pub":"%s","kty":"%s"}`

    ...

    func pqcThumbprintInput(k *pqc.PublicKey) (string, error) {
        return fmt.Sprintf(pqcThumbprintTemplate,
            newBuffer(k.Bytes).base64(),
            newBuffer([]byte(k.AlgName)).base64()), nil
    }

    // IsPublic returns true if the JWK represents a public key (not
    symmetric, not private).
    func (k *JSONWebKey) IsPublic() bool {
        switch k.Key.(type) {
        case *ecdsa.PublicKey, *rsa.PublicKey, ed25519.PublicKey,
        *pqc.PublicKey:
            return true
        default:
            return false
        }
    }

    // Public creates JSONWebKey with corresponding public key if JWK
    represents asymmetric private key.
    func (k *JSONWebKey) Public() JSONWebKey {
        if k.IsPublic() {
            return *k
        }
        ret := *k
        switch key := k.Key.(type) {
        case *pqc.PrivateKey:
            ret.Key = key.Public()
        ...
        return ret
    }

    // Valid checks that the key contains the expected parameters.
    func (k *JSONWebKey) Valid() bool {
        if k.Key == nil {
            return false
        }
        switch key := k.Key.(type) {
        case *pqc.PublicKey:
            if key.Bytes == nil {
                return false
            }
        ...
        return true
    }
```

Figura 47. Modificações do arquivo jwk.go
Fonte: Original

```
        func (key rawJSONWebKey) pqcPublicKey()
        (*pqc.PublicKey, error) {
        return &pqc.PublicKey{
            AlgName: key.Kty,
            Bytes:   key.PQCPub.bytes(),
        }, nil
    }

    ...

    func fromPQCPublicKey(pub *pqc.PublicKey) *rawJSONWebKey {
        return &rawJSONWebKey{
            Kty:     pub.AlgName,
            PQCPub: newBuffer(pub.Bytes),
        }
    }

    ...

    func (key rawJSONWebKey) pqcPrivateKey() (*pqc.PrivateKey, error) {
        privKey := &pqc.PrivateKey{
            PublicKey: pqc.PublicKey{
                AlgName: key.Kty,
                Bytes:   key.PQCPub.bytes(),
            },
        }

        privKey.Signer.Init(key.Kty, key.PQCPriv.bytes())

        return privKey, nil
    }

    ...

    func fromPQCPrivateKey(pqc *pqc.PrivateKey)
    (*rawJSONWebKey, error) {
        raw := fromPQCPublicKey(&pqc.PublicKey)

        raw.PQCPriv = newBuffer(pqc.Signer.ExportSecretKey())

        return raw, nil
    }

    ...
```

Figura 48. Modificações do arquivo jwk.go
Fonte: Original

No tipo **rawJSONWebKey** foram adicionados dois novos atributos: **PQCPub** e **PQCPriv**, cada um com os *bytes* da chave pública e privada respectivamente. No método **MarshalJSON**, onde uma dada **JSONWebKey** é serializada pra sua representação JSON, adicionou-se casos para chave pública e privada pós-quântica. Da mesma forma, foram adicionados casos no método **UnmarshalJSON** para a chave pública e privada pós-

quântica, operação inversa da anterior.

Também criou-se uma constante que representa o template do *thumbprint* – digital – de uma chave pública pós-quântica de qualquer algoritmo pós-quântica que chamou-se de **pqcThumbprintTemplate**. Esse *template*, ou modelo, que é utilizado pelo método **pqcThumbprintInput**, o qual foi implementado. Da mesma forma, foram adicionados casos à função **Thumbprint** para as chaves públicas e privadas pós-quântica para utilizarem o método anterior citado.

Para os métodos **IsPublic** e **Public** também foram adicionados casos para cobrir as chaves pós-quânticas. Assim como foi adicionado no método **Valid** um caso também.

Foram implementados os métodos **pqcPublicKey**, o qual utiliza uma **rawJSONWebKey** e recupera a chave pós-quântica dessa estrutura. Assim como foi implementado o método **fromPQCPublicKey**, o qual realiza a operação inversa. Dessa mesma maneira, foram implementados os métodos **pqcPrivateKey** e **fromPQCPrivateKey** que lida da mesma forma, porém com chaves privadas.

O arquivo **shared.go** também sofreu alterações. Esse arquivo mantém estruturas e métodos que são compartilhados por diversos outros, como nome de algoritmos, erros, etc.

```
1   // Signature algorithms
2   const (
3       ...
4       Dilithium5                  =
5       SignatureAlgorithm("Dilithium5")
6       Dilithium5AES               =
7       SignatureAlgorithm("Dilithium5-AES")
8       Falcon1024                  =
9       SignatureAlgorithm("Falcon-1024")
10      ...
11      SphincsPlusHaraka128fSimple    =
12      SignatureAlgorithm("SPHINCS+-Haraka-128f-Simple")
13      ...
14      SphincsPlusSHAKE256128fRobust =
15      SignatureAlgorithm("SPHINCS+-SHAKE256-128f-Robust")
16  )
```

Figura 49. Adições do arquivo shared.go
Fonte: Original

Foram adicionadas constantes para os algoritmos de assinatura pós-quântica integrados e apenas isso foi necessário nesse arquivo, como pode ser visto na Figura 49.

E, por fim, o arquivo **signing.go** foi modificado. Esse é o principal arquivo que lida com a questão de assinaturas para dados em formato JSON, ilustrado na Figura 50.

```
1     // newVerifier creates a verifier based on the key type
2     func newVerifier(verificationKey interface{})
3     (payloadVerifier, error) {
4         switch verificationKey := verificationKey.(type) {
5         case *pqc.PublicKey:
6             return &pqcEncrypterVerifier{
7                 publicKey: verificationKey,
8             }, nil
9         ...
10        return nil, ErrUnsupportedKeyType
11    }
12
13    ...
14
15    func makeJWSRecipient(alg SignatureAlgorithm,
16    signingKey interface{}) (recipientSigInfo, error) {
17        switch signingKey := signingKey.(type) {
18        case *pqc.PrivateKey:
19            return newPQCSigner(alg, signingKey)
20        ...
21        if signer, ok := signingKey.(OpaqueSigner); ok {
22            return newOpaqueSigner(alg, signer)
23        }
24        return recipientSigInfo{}, ErrUnsupportedKeyType
25    }
```

Figura 50. Funções do arquivo signing.go
Fonte: Original

No método **newVerifier**, o qual cria um verificador baseado no tipo de chave, adicionou-se um caso para a chave pública pós-quântica integrada. O mesmo foi feito no método **makeJWSRecipient**, adicionando um caso para a chave privada pós-quântica.

4.4. Pebble

Pebble é um pequeno servidor ACME para testes, seu uso não se enquadra para uma AC a nível de produção. Porém, como ele é feito para testes, possui diversas facilidades nesse quesito. Ele provê interfaces simples para utilizar o servidor ACME, casos de teste para novas funcionalidades, tenta garantir, ao máximo, o uso dele apenas para testes, etc.

Ou seja, o Pebble além de ser um servidor ACME também possui uma AC de testes para a emissão, renovação e revogação dos certificados. Dessa forma, as modificações realizadas se deram em relação as chaves dos certificados – tornando-os pós-quânticos – e em relação as mensagens JSON trocadas, dando suporte aos algoritmos pós-quânticos.

O primeiro arquivo modificado foi **ca.go** o qual trata da criação de uma AC e toda a cadeia de certificação, para prototipagem, demonstrado na Figura 51 e na Figura 52.

```go
// makeKey creates a new PQC private key and a Subject Key Identifier
func makeRootKey() (*pqc.PrivateKey, []byte, error) {
    key, err := pqc.GenerateKey("sphincs+-shake256-256f-robust")
    if err != nil {
        return nil, nil, err
    }
    ski, err := makeSubjectKeyID(key.Public())
    if err != nil {
        return nil, nil, err
    }
    return key, ski, nil
}

func makeInterKey() (*pqc.PrivateKey, []byte, error) {
    key, err := pqc.GenerateKey("sphincs+-shake256-128f-robust")
    if err != nil {
        return nil, nil, err
    }
    ski, err := makeSubjectKeyID(key.Public())
    if err != nil {
        return nil, nil, err
    }
    return key, ski, nil
}
```

Figura 51. Funções do arquivo ca.go
Fonte: Original

```
1   func (ca *CAImpl) GetRootKey(no int) *pqc.PrivateKey {
2       chain := ca.getChain(no)
3       if chain == nil {
4           return nil
5       }
6
7       switch key := chain.root.key.(type) {
8       case *pqc.PrivateKey:
9           return key
10      }
11      return nil
12  }
13
14  ...
15
16  func (ca *CAImpl) GetIntermediateKey(no int) *pqc.PrivateKey {
17      chain := ca.getChain(no)
18      if chain == nil {
19          return nil
20      }
21
22      switch key := chain.intermediates[0].key.(type) {
23      case *pqc.PrivateKey:
24          return key
25      }
26      return nil
27  }
```

Figura 52. Funções do arquivo ca.go
Fonte: Original

O método **makeKey**, o qual gera uma chave e é utilizado em cada um dos certificados da cadeia, foi substituído por dois métodos: **makeRootKey** e **makeInterKey**, dessa forma, pode-se ter níveis diferentes de segurança dependendo do nível em que a AC se encontra. Além disso, esses métodos agora geram uma chave pós-quântica. Da mesma forma, alteraram-se os métodos **GetRootKey** e **GetIntermediateKey** para retornar uma chave pós-quântica.

Outro arquivo modificado foi **jose.go** o qual cria uma simples abstração para alguns métodos em cima da biblioteca Go-JOSE, como já visto e ilustrado na Figura 53.

```
func algorithmForKey(key *jose.JSONWebKey) (string, error) {
    switch k := key.Key.(type) {
    case *pqc.PublicKey:
        switch k.AlgName {
        case "dilithium5":
            return string(jose.Dilithium5), nil
        ...
        case "rainbow-i-classic":
            return string(jose.RainbowIClassic), nil
        ...
        case "sphincs+-shake256-128f-robust":
            return string(jose.SphincsPlusSHAKE256128fRobust), nil
        }
    ...
    return "", fmt.Errorf("no signature algorithms
    suitable for given key type: %T", key.Key)
}
```

Figura 53. Modificações do arquivo jose.go
Fonte: Original

No método **algorithmForKey**, o qual retorna o algoritmo da chave JSON que o assina, foram adicionados casos para os algoritmos pós-quânticos.

E, por fim, o último arquivo modificado foi **wfe.go** o qual lida com as possíveis ações do protocolo ACME.

```
...
type keyGetter func(no int) *pqc.PrivateKey
...
```

Figura 54. Adição do arquivo wfe.go
Fonte: Original

Nesse caso, apenas foi necessário alterar o tipo **keyGetter**, o qual é uma função, para retornar uma chave pós-quântica, como visto na Figura 54.

4.5. LEGO

LEGO – sigla para *Let's Encrypt client and ACME library written in Go* ou cliente Let's Encrypt e biblioteca ACME escrita em Go – é um cliente ACME completo. Ele não só é utilizado para prototipagem como também é feito para ambientes de produção completos com diversas funcionalidades.

Ele é baseado na RFC8555 [Barnes et al. 2019] da versão 2 do protocolo ACME e com esse cliente é possível: se registrar a uma AC, obter certificados do zero – criar par de chaves, autenticar, etc – ou obter certificados a partir de uma CSR, renovar certificados, revogar certificados, entre outras diversas funcionalidades.

Por ele ser escrito na linguagem Go, ou seja, ser compatível com o Pebble para integrar as modificações mais facilmente e por ele ser muito utilizado foi escolhido como o melhor cliente para integrar ao trabalho.

A implementação no cliente LEGO pode ser dividida em duas principais partes: modificações em pacotes de mais baixo nível – que lidam com criptografia e com mensagens JSON assinadas digitalmente – e modificações nas configurações do cliente, como parâmetros para qual algoritmo será utilizado.

Primeiramente, modificou-se a parte mais baixo nível do cliente LEGO, dessa forma, começando com o arquivo **jws.go** que lida com JSON Web Signature [Jones et al. 2015a] que foi modificado para realizar a assinatura de dados JSON com os algoritmos pós-quânticos.

```
1   // SignContent Signs a content with the JWS.
2   func (j *JWS) SignContent(url string, content []byte)
3   (*jose.JSONWebSignature, error) {
4       var alg jose.SignatureAlgorithm
5       switch k := j.privKey.(type) {
6       case *pqc.PrivateKey:
7           switch k.AlgName {
8           case "dilithium5":
9               alg = jose.Dilithium5
10          ...
11          case "rainbow-v-circumzenithal":
12              alg = jose.RainbowVCircumzenithal
13          ...
14          case "dilithium2":
15              alg = jose.Dilithium2
16          ...
17          case "sphincs+-shake256-128f-robust":
18              alg = jose.SphincsPlusSHAKE256128fRobust
19          }
20      ...
21      return signed, nil
22  }
```

Figura 55. Modificação do arquivo jws.go
Fonte: Original

No método **SignContent** adicionou-se um caso para a chave privada pós-quântica e, para este caso, um para cada algoritmo pós-quântico integrado, como pode ser visto na Figura 55.

Outro arquivo modificado nessa etapa foi **crypto.go** o qual trata da criação e manipulação de chaves criptográficas e de arquivos formatados – principalmente em formatos muito utilizados em certificados digitais e chaves criptográficas, como o PEM [Josefsson and Leonard 2015].

```
    // Constants for all key types we support.
    const (
        ...
        Dilithium5                      = KeyType("dilithium5")
        ...
        Falcon512                       = KeyType("falcon-512")
        ...
        SphincsPlusSHAKE256128fRobust =
        KeyType("sphincs+-shake256-128f-robust")
    )

    ...

    func ParsePEMPrivateKey(key []byte) (crypto.PrivateKey, error) {
        keyBlockDER, _ := pem.Decode(key)

        if keyBlockDER.Type != "PRIVATE KEY" &&
        !strings.HasSuffix(keyBlockDER.Type, " PRIVATE KEY") {
            return nil, fmt.Errorf("unknown PEM header %q",
            keyBlockDER.Type)
        }

        ...

        if key, err := x509.ParsePKCS8PrivateKey(keyBlockDER.Bytes);
        err == nil {
            switch key := key.(type) {
            case *rsa.PrivateKey, *ecdsa.PrivateKey,
            ed25519.PrivateKey, *pqc.PrivateKey:
                return key, nil
            default:
                return nil, fmt.Errorf("found unknown
                private key type in PKCS#8 wrapping: %T", key)
            }
        }

        ...
    }
```

Figura 56. Modificações do arquivo crypto.go
Fonte: Original

```
1   func GeneratePrivateKey(keyType KeyType)
2   (crypto.PrivateKey, error) {
3       switch keyType {
4       ...
5       case Dilithium5:
6           return pqc.GenerateKey("dilithium5")
7       ...
8       case SphincsPlusHaraka256sRobust:
9           return pqc.GenerateKey("sphincs+-haraka-256s-robust")
10      ...
11      case Falcon512:
12          return pqc.GenerateKey("falcon-512")
13      ...
14      case SphincsPlusSHAKE256128fRobust:
15          return pqc.GenerateKey("sphincs+-shake256-128f-robust")
16      }
17
18      return nil, fmt.Errorf("invalid KeyType: %s", keyType)
19  }
```

Figura 57. Modificações do arquivo crypto.go
Fonte: Original

Neste arquivo – Figura 56 e 57 – foram adicionadas constantes para os algoritmos pós-quânticos integrados com seus respectivos nomes da LibOQS. No método **ParsePEMPrivateKey**, o qual utiliza um vetor de *bytes* que representa a chave privada em formato PEM e transforma para a estrutura de chave privada que criada, foi adicionado o caso para a chave privada pós-quântica. Da mesma forma, foi feito no método **GeneratePrivateKey**, o qual retorna a estrutura de chave privada com base no tipo de chave passado por parâmetro, dessa forma, adicionaram-se os algoritmos que integrados como novos casos.

Já na segunda parte, onde modificaram-se arquivos de configuração do cliente LEGO, foi alterado, primeiramente, o arquivo **setup.go** o qual cria um ambiente de inicialização do sistema LEGO.

```
1     // getKeyType the type from which private keys should be generated.
2     func getKeyType(ctx *cli.Context) certcrypto.KeyType {
3         keyType := ctx.GlobalString("key-type")
4         switch strings.ToUpper(keyType) {
5         ...
6         case "DILITHIUM5":
7             return certcrypto.Dilithium5
8         ...
9         ...
10        case "SPHINCS+-SHAKE256-128F-ROBUST":
11            return certcrypto.SphincsPlusSHAKE256128fRobust
12        }
13
14        log.Fatalf("Unsupported KeyType: %s", keyType)
15        return ""
16    }
```

Figura 58. Modificações do arquivo setup.go
Fonte: Original

Nesse arquivo (Figura 58) apenas foi adicionado casos para os novos algoritmos pós-quânticos integrados no método **getKeyType**, para que o algoritmo de chave passado como parâmetro pela linha de comando tenha essas novas opções disponíveis.

E, finalmente, o arquivo **client_config.go** o qual cria configurações padrão, as quais podem ser alteradas passando diferentes parâmetros de inicialização, do lado cliente, ou seja, um ambiente padrão caso não sejam passados certos parâmetros, como pode ser visto na Figura 59.

```
1     func NewConfig(user registration.User) *Config {
2         return &Config{
3             CADirURL:    LEDirectoryProduction,
4             User:        user,
5             HTTPClient: createDefaultHTTPClient(),
6             Certificate: CertificateConfig{
7                 KeyType: certcrypto.Falcon1024,
8                 Timeout: 30 * time.Second,
9             },
10        }
11    }
```

Figura 59. Modificações do arquivo client_config.go
Fonte: Original

No método **NewConfig** foi alterado o **KeyType** para um tipo específico pós-quântico. Porém, esse tipo é apenas usado caso não seja explicitamente dito, por parâmetro, outro tipo. Essa modificação não é necessária, porém para um ambiente de desenvolvimento onde o algoritmo pós-quântico é irrelevante – desde que seja pós-quântico – pode facilitar o processo.

5. Resultados

Realizaram-se testes para diversas funcionalidades do protocolo ACME como é detalhado em cada teste a seguir. Os testes tem como principais objetivos comparar o desempenho dos algortimos integrados assim como comparar com os algoritmos clássicos.

Para os algoritmos, foram testados todos os que foram integrados, como explicitado anteriormente neste trabalho. Já para os clássicos foram testados os algoritmos **RSA** – com a variação RSA8192 e RSA2048 – e **ECDSA** – com a variação EC384 e EC256. Escolheram-se esses dois algoritmos clássicos, pois o algoritmo RSA é o algoritmo padrão do cliente LEGO, ou seja, se não for pedido a utilização de outro algoritmo para a criação do par de chaves, este será utilizado. E o algoritmo ECDSA, pois é o principal algoritmo utilizado quando se trata de curvas elípitcas.

Para cada um dos testes realizados, mediu-se o tempo para realizar a operação (em nanosegundos), a quantidade de bytes alocados para a operação e a quantidade de alocações realizadas. Cada tabela, foi o resultado de 1000 iterações dos testes, obtendo seus valores médios. Esses testes foram baseados nos próprios testes do cliente LEGO, no entanto resultando em mais estatísticas e, também, integrando os algoritmos pós-quânticos.

5.1. Configuração do Ambiente

Todos os testes realizados foram num ambiente local – na mesma máquina – com o intuito de simplicidade e isolamento dos aspectos de rede, como latência e perda de pacotes.

Além disso, todos os testes foram baseados nos testes do cliente LEGO – como mencionado. Isso se refere aos testes de unidade que já existem no próprio código LEGO [LEG 2021]. Porém, algumas modificações foram realizadas, como a mudança de testes de validação (os quais apenas testas se certa funcionalidade se comporta como esperado) para testes de *benchmark* que trazem informações de desemepenho consigo. É possível ver também que cada tabela de resultados a seguir possui um nome, como *Benchmark-CertificateService_Get*, esse nome se refere ao mesmo nome do teste original do LEGO – para manter o mesmo padrão de nomenclatura. Em relação às métricas medidas, tem-se que o tempo de execução se refere a todo o tempo necessário para a execução do teste que pode incluir criação de mensagens, chamadas de funções, etc. Já a quantidade de bytes alocados se refere à quantidade de bytes alocados por teste, ou seja, quantos bytes são alocados em memória por teste. E, por fim, a quantidade de alocações se refere a quantas vezes foi necessário realizar uma alocação na memória por teste, nota-se que é similar, porém diferente da métrica anterior a qual mede a quantidade de bytes alocados e não quantas alocações foram necessárias para alocar todos esses bytes.

Para a realização dos testes foi utilizada apenas uma única máquina com a seguinte configuração relevante para os testes:

- Processador: AMD Ryzen 7 3800X 8-Core (16 CPUs), 3.9GHz
- Memória: 16GB
- Sistema Operacional: Ubuntu 21.10 64-bit

Além disso, dentre todas os testes LEGO, foram medidos resultados do seguinte conjunto:

- BenchmarkCertificateService_Get: Recuperação do certificado digital

- BenchmarkOrderService_New: Requisição de um novo certificado (*newOrder*)
- BenchmarkGeneratePrivateKey: Geração do par de chaves criptográficas
- BenchmarkGenerateCSR: Geração da CSR
- BenchmarkPEMEncode: Codificação das chaves criptográficas para o formato PEM
- BenchmarkParsePEMCertificate: Conversão do certificado digital em formato PEM para o código Go.
- BenchmarkChallenge: Realização do desafio HTTP

Dessa forma, as principais operações do protocolo ACME são testadas – as operações necessárias para emissão de um certificado digital – e podem ser vistas em mais detalhes em cada um dos testes a seguir.

5.2. Benchmarks

Um dos passos finais do protocolo ACME e um dos seus principais objetivos é a emissão de um certificado digital. Neste teste é medido o desempenho de recuperar um certificado com a utilização dos algoritmos abaixo na Tabela 1. Primeiramente, nota-se que o número de alocações dos algoritmos pós-quânticos é bem estável, com pouca variações entre os algoritmos, e bem abaixo do número de alocações que os algoritmos clássicos executaram.

Além disso, em relação ao número de bytes alocados, percebe-se que o algoritmo Falcon teve um desempenho excelente, tendo número bem abaixo dos outros candidatos. Em seguida, o algoritmo Dilithium, junto com algumas variações do algoritmo SPHINCS+ também obtiveram ótimos desempenhos, assemelhando-se ao Falcon. Já o algoritmo Rainbow teve um número de bytes alocados pelo menos a uma ordem de grandeza acima, sendo pior nesse quesito. Os algoritmos clássicos, por sua vez, desempenharam de formas bem variadas também, Com resultados muito bons, como do algoritmo EC256, resultados intermediários como de EC384 e RSA2048, e resultados um pouco piores como do algoritmo RSA8192 – porém, com quantidade de bytes alocados abaixo do algoritmo Rainbow ainda.

E, por fim, o tempo de execução teve como seu melhor desemepenhante o algoritmo Dilithium, com uma ordem de grande abaixo do próximo candidato, o algoritmo Falcon e algumas variações do algoritmo SPHINCS+ – o qual, teve variações que desempenharam muito bem e outras com tempos bem acima, possuindo o melhor e o pior tempo no menor nível de segurança. O algoritmo Rainbow, obteve tempos maiores na sua versão comprimida em relação ao maior nível de segurança, com tempos um pouco menores na versão *circumzenithal* e tempos menores ainda na versão clássica, o que surpreende, visto que a complexidade seria inversa a essa ordem de desemepenho. Porém, ainda assim esse nível de segurança desempenhou pior que os outros algoritmos. Já no menor nível de segurança, a versão clássica obteve piores tempos, seguida da comprimida e da *circumzenithal* logo depois. Em relação ao algoritmos clássicos, percebe-se que o RSA8192 obteve o pior tempo de todos os candidatos. Os outros algoritmos clássicos tiveram tempos similares, como o RSA2048 e o EC384 que assemelharam-se a algumas variações do SPHINCS+. O algortimo EC256 obteve tempos muito bons também, assemelhando-se ao algoritmo Dilithium.

Um dos primeiros passos para a emissão de um certificado utilizando o protocolo ACME é o pedido de newOrder – como explicado em seções anteriores. Neste teste é

	tempo (ns)	bytes	alocações
RSA8192	10155533507	10017952	9939
RSA2048	257552848	4234304	11273
EC384	6855677	3507464	28697
EC256	742294	96832	662
dilithium5	1121193	250096	586
dilithium5-aes	969369	246320	571
falcon-1024	21238404	187016	583
rainbow-V-classic	4674049033	76479616	594
rainbow-V-circumzenithal	5109131929	22412088	590
rainbow-V-compressed	7338125208	20998400	600
sphincs+-haraka-256s-simple	157437177	377536	598
sphincs+-haraka-256f-simple	15808855	592256	575
sphincs+-haraka-256s-robust	192191153	382464	580
sphincs+-haraka-256f-robust	19806326	594752	581
sphincs+-sha256-256s-simple	336648316	384992	589
sphincs+-sha256-256f-simple	43610094	596192	591
sphincs+-sha256-256s-robust	1176402460	376832	585
sphincs+-sha256-256f-robust	130547908	603904	592
sphincs+-shake256-256s-simple	799759836	368768	595
sphincs+-shake256-256f-simple	81996421	590624	576
sphincs+-shake256-256s-robust	1384110987	376096	585
sphincs+-shake256-256f-robust	149463654	603200	584
dilithium2	848558	172936	582
dilithium2-aes	729076	183896	593
falcon-512	5681192	126744	579
rainbow-I-classic	617434409	6620920	592
rainbow-I-circumzenithal	123674375	2639400	589
rainbow-I-compressed	174787641	2540872	587
sphincs+-haraka-128s-simple	92914671	173584	580
sphincs+-haraka-128f-simple	5451254	268648	581
sphincs+-haraka-128s-robust	107349266	167088	573
sphincs+-haraka-128f-robust	5911755	261152	589
sphincs+-sha256-128s-simple	242568567	171616	565
sphincs+-sha256-128f-simple	13893869	267680	582
sphincs+-sha256-128s-robust	432991367	176080	575
sphincs+-sha256-128f-robust	23687277	262040	573
sphincs+-shake256-128s-simple	516400456	177568	581
sphincs+-shake256-128f-simple	27689152	247744	568
sphincs+-shake256-128s-robust	924943749	169992	587
sphincs+-shake256-128f-robust	48245022	268616	580

Tabela 1. Resultados BenchmarkCertificateService_Get
Fonte: Original

medido o desempenho de cada algoritmo na realização dessa etapa para assinar as mensagens, como pode ser visto na Tabela 2.

	tempo (ns)	bytes	alocações
RSA8192	96677936517	97570440	91730
RSA2048	279057929	4986416	13266
EC384	17973537	6949448	57257
EC256	777645	93912	886
dilithium5	1571755	427808	778
dilithium5-aes	1380358	419808	778
falcon-1024	22030118	267640	753
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	157567926	778768	794
sphincs+-haraka-256f-simple	18161793	1316216	792
sphincs+-haraka-256s-robust	195167473	763008	765
sphincs+-haraka-256f-robust	21996401	1317408	780
sphincs+-sha256-256s-simple	340909259	766984	779
sphincs+-sha256-256f-simple	47974207	1292672	771
sphincs+-sha256-256s-robust	1185136692	754944	755
sphincs+-sha256-256f-robust	137486203	1318352	793
sphincs+-shake256-256s-simple	808838800	783080	781
sphincs+-shake256-256f-simple	89750627	1307088	777
sphincs+-shake256-256s-robust	1349992211	782176	780
sphincs+-shake256-256f-robust	160205543	1316424	793
dilithium2	1215625	263632	769
dilithium2-aes	1235105	260280	777
falcon-512	6645092	172936	762
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	93732496	270736	766
sphincs+-haraka-128f-simple	6128456	518816	765
sphincs+-haraka-128s-robust	106527981	283592	785
sphincs+-haraka-128f-robust	7010889	516528	775
sphincs+-sha256-128s-simple	243659533	291368	781
sphincs+-sha256-128f-simple	16282851	504880	766
sphincs+-sha256-128s-robust	449395053	290880	778
sphincs+-sha256-128f-robust	28265190	507312	775
sphincs+-shake256-128s-simple	521267119	292208	786
sphincs+-shake256-128f-simple	32097843	522368	768
sphincs+-shake256-128s-robust	974057123	291096	771
sphincs+-shake256-128f-robust	57236346	515000	776

Tabela 2. Resultados BenchmarkOrderService_New
Fonte: Original

Primeiramente, nota-se que o algoritmo Rainbow não obteve resultados nesse

teste. Isso se deve à interação do algoritmo com o cliente LEGO e a biblioteca de Benchmark, onde o número de bytes alocados deve ter sido superior à algum limite, resultando em erro na obtenção de resultados.

Na questão do número de alocações, os algoritmos pós-quânticos tiveram resultados muito bons e consistentes ao longo de todos eles, mantendo números muito similares. Já os clássicos, tiveram números muito acima dos pós-quânticos, em média.

Em relação aos número de bytes alocados, o algoritmo Falcon obteve resultados ótimos, desempenhando melhor que os outros candidatos. O algoritmo Dilithium não ficou muito longe desses números, obtendo resultados muito bons também nessa questão. Por mais que o algoritmo SPHINCS+ tenha desempenhado pior nesse quesito, em alguns casos nota-se que seus resulados foram muito similares aos do Dilithium. Em comparação aos clássicos, os pós-quânticos tiveram resultados melhores na sua maioria.

Por fim, na questão de tempo de execução, o algoritmo Dilithium foi o que obteve melhor desempenho, tendo números muito abaixo dos de mais. Em seguida, o algoritmo SPHINCS+ teve melhores resultados nos casos de variações específicas. Porém em alguns casos do algoritmo SPHINCS+, o mesmo obteve resultados piores que os outros candidatos. Já o algoritmo Falcon, ficou no meio termo nesse parâmetro, assemelhando-se a algumas variações do SPHINCS+. Os algoritmos clássicos, com exceção do RSA8192 que teve tempos de execução muito acima dos de mais, tiveram resultados bem variados, ou seja, alguns com tempos bem abaixo como EC256, e outros com tempos mais altos, como RSA2048.

Um dos primeiros passos do protocolo ACME está relacionado a geração do par de chaves – em especial da chave privada a qual é utilizada para a assinatura digital – por meio do cliente ACME. O próximo teste mede justamente os parâmetros de cada algoritmo durante essa etapa de geração de par de chaves criptográficas, com os resultados na Tabela 3.

	tempo (ns)	bytes	alocações
RSA8192	42013866457	42268552	39519
RSA2048	249897259	4024784	10367
EC384	2979343	1727472	14277
EC256	38542	1816	16
dilithium5	126531	7808	5
dilithium5-aes	58165	7808	5
falcon-1024	14171837	5000	6
rainbow-V-classic	3961783928	3342616	5
rainbow-V-circumzenithal	4642912712	1949976	5
rainbow-V-compressed	4548094094	541016	5
sphincs+-haraka-256s-simple	10424595	464	5
sphincs+-haraka-256f-simple	653786	464	5
sphincs+-haraka-256s-robust	12685489	464	5
sphincs+-haraka-256f-robust	813350	464	5
sphincs+-sha256-256s-simple	24268373	464	5
sphincs+-sha256-256f-simple	1747093	464	5
sphincs+-sha256-256s-robust	94862236	464	5
sphincs+-sha256-256f-robust	5846644	464	5
sphincs+-shake256-256s-simple	59927736	464	5
sphincs+-shake256-256f-simple	3458569	464	5
sphincs+-shake256-256s-robust	102339587	464	5
sphincs+-shake256-256f-robust	6588463	464	5
dilithium2	56424	4352	5
dilithium2-aes	34273	4352	5
falcon-512	4789825	2688	5
rainbow-I-classic	103924045	270616	5
rainbow-I-circumzenithal	118922792	172312	5
rainbow-I-compressed	118168160	65880	5
sphincs+-haraka-128s-simple	10050678	368	5
sphincs+-haraka-128f-simple	196021	368	5
sphincs+-haraka-128s-robust	11704011	368	5
sphincs+-haraka-128f-robust	199918	368	5
sphincs+-sha256-128s-simple	27526821	368	5
sphincs+-sha256-128f-simple	583804	368	5
sphincs+-sha256-128s-robust	49992281	368	5
sphincs+-sha256-128f-robust	794592	368	5
sphincs+-shake256-128s-simple	58460269	368	5
sphincs+-shake256-128f-simple	964555	368	5
sphincs+-shake256-128s-robust	106019910	368	5
sphincs+-shake256-128f-robust	1757778	368	5

Tabela 3. Resultados BenchmarkGeneratePrivateKey
Fonte: Original

Primeiramente, nota-se que o número de alocações dos algoritmos pós-quânticos

se manteve igual. Isso se deve, provavelmente, pelo de fato de todos serem implementados pela mesma biblioteca, compartilhando funcionalidades em comum. Onde os algoritmos clássicos possuem ordens de grandeza maiores nessa questão.

Em relação à quantidade de bytes alocados, o algoritmo SPHINCS+ teve uma quantidade muito abaixo tanto dos algoritmos pós-quânticos quanto dos clássicos. O algoritmo Falcon já supera em uma ordem de grandeza nesse aspecto e o algoritmo Dilithium com números um pouco maiores, porém não se compara à diferença dos anteriores. Além disso, nota-se que o algoritmo Rainbow ficou muito acima dos outros pós-quânticos em relação aos bytes alocados. Percebe-se também que os algoritmos pós-quânticos tiveram resultados melhores nesse aspecto em relação aos clássicos, com algumas exceções. Nota-se também que o algoritmo EC256 teve números muito abaixo dos outros clássicos.

Na questão do tempo de execução, percebe-se que o algoritmo RSA teve os piores números em relação à todos os algoritmos. Entre os pós-quânticos, o algoritmo Dilithium obteve os melhores resultados de tempo, equanto o algoritmo Falcon e o algoritmo SPHINCS+ se mantevem num meio termo, onde o algoritmo Rainbow teve números bem acima dos outros candidatos pós-quânticos.

Um dos passos mais importantes do protocolo ACME é a geração da CSR por meio do cliente ACME, a qual, posteriormente, se tornará um certificado digital. Este seguinte teste mede os parâmetros para a geração de uma CSR utilizando cada um dos algoritmos – tanto para a geração do certificado quanto para a assinatura da mensagem que a contém, como pode ser visto na Tabela 4.

	tempo (ns)	bytes	alocações
RSA8192	66684832	193936	378
RSA2048	1441589	38536	299
EC384	3070452	1743032	14528
EC256	104447	9696	220
dilithium5	190490	20352	163
dilithium5-aes	107317	20352	163
falcon-1024	492916	11264	163
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	143949408	70304	163
sphincs+-haraka-256f-simple	13705182	119456	163
sphincs+-haraka-256s-robust	176844016	70304	163
sphincs+-haraka-256f-robust	17615438	119456	163
sphincs+-sha256-256s-simple	304702438	70288	163
sphincs+-sha256-256f-simple	39679180	119456	163
sphincs+-sha256-256s-robust	1078656368	70304	163
sphincs+-sha256-256f-robust	120294655	119456	163
sphincs+-shake256-256s-simple	704655643	70304	163
sphincs+-shake256-256f-simple	76637610	119456	163
sphincs+-shake256-256s-robust	1225975063	70304	163
sphincs+-shake256-256f-robust	136095174	119456	163
dilithium2	187888	12800	163
dilithium2-aes	93538	12800	163
falcon-512	277770	8128	163
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	81166243	21104	163
sphincs+-haraka-128f-simple	4486391	41584	163
sphincs+-haraka-128s-robust	94119698	21104	163
sphincs+-haraka-128f-robust	4887491	41584	163
sphincs+-sha256-128s-simple	216523901	21104	163
sphincs+-sha256-128f-simple	12570565	41584	163
sphincs+-sha256-128s-robust	382643459	21104	163
sphincs+-sha256-128f-robust	22361670	41584	163
sphincs+-shake256-128s-simple	479108026	21104	163
sphincs+-shake256-128f-simple	25613943	41584	163
sphincs+-shake256-128s-robust	840072674	21104	163
sphincs+-shake256-128f-robust	48709339	41584	163

Tabela 4. Resultados BenchmarkGenerateCSR
Fonte: Original

Nota-se, primeiramente, que o algoritmo Rainbow não teve resultados. Isso se

deve, pois ele gera CSRs e certificados digitais muito grandes os quais não cabem nos parâmetros padrão do protocolo HTTP, não sendo possível enviá-la sem alterações no protocolo HTTP. Percebe-se novamente, também, o mesmo valor para o número de alocações nos algoritmos pós-quânticos pelos mesmos motivos já mencionados. Entre os algoritmos clássicos o RSA teve um ótimo desempenho nesse quesito.

Em relação ao número de bytes alocados, o algoritmo SPHINCS+ ficou acima dos outros algoritmos pós-quânticos, porém de maneira consistente, com pouca variação. Os algortimos Dilithium e Falcon obtiveram números similares entre si, porém com o algoritmo Falcon tendo melhores resultados neste parâmetro. Já os algoritmos clássicos tiveram números bem variados nesse quesito, tendo resultados comparáveis aos pós-quânticos e resultados muito inferiores ao mesmo tempo.

Na questão do tempo de execução, o algoritmo Dilithium obteve tempos impressionantes, sendo bem abaixo dos outros candidatos pós-quânticos e até dos algoritmos clássicos. O algoritmo Falcon também teve bons tempos, porém um pouco acima do anterior. Já o algoritmo SPHINCS+ teve uma certa consistência nos números, porém com tempos muito acima dos demais. Em alguns casos obteve tempos acima até do RSA8192 que geralmente tem os piores desempenhos nesses testes.

Outro aspecto importante, não apenas para o protocolo ACME, mas para várias aplicações criptográficas também, é a codificação no formato PEM. Neste seguinte teste é medido o desempenho da codificação de chaves criptográficas, com os resultados na Tabela 5.

	tempo (ns)	bytes	alocações
RSA8192	8260881523	7955088	7590
RSA2048	95381935	1480416	4112
EC384	2800559	1726008	14268
EC256	47720	5160	64
dilithium5	139771	50864	58
dilithium5-aes	77488	50856	57
falcon-1024	13934361	28328	56
rainbow-V-classic	3904954233	3344680	33
rainbow-V-circumzenithal	4537623695	1952040	33
rainbow-V-compressed	4583668947	543080	33
sphincs+-haraka-256s-simple	10498036	3800	52
sphincs+-haraka-256f-simple	666729	3800	52
sphincs+-haraka-256s-robust	12979960	3800	52
sphincs+-haraka-256f-robust	835164	3800	52
sphincs+-sha256-256s-simple	24183459	3800	52
sphincs+-sha256-256f-simple	1755658	3800	52
sphincs+-sha256-256s-robust	93613971	3800	52
sphincs+-sha256-256f-robust	5809053	3800	52
sphincs+-shake256-256s-simple	59246639	3800	52
sphincs+-shake256-256f-simple	3604395	3800	52
sphincs+-shake256-256s-robust	105818320	3800	52
sphincs+-shake256-256f-robust	6620277	3800	52
dilithium2	100494	26920	56
dilithium2-aes	54559	26920	56
falcon-512	4426888	15272	55
rainbow-I-classic	105650448	272680	33
rainbow-I-circumzenithal	118303789	174376	33
rainbow-I-compressed	117377480	67944	33
sphincs+-haraka-128s-simple	10377752	3576	52
sphincs+-haraka-128f-simple	218727	3576	52
sphincs+-haraka-128s-robust	12477908	3576	52
sphincs+-haraka-128f-robust	213318	3576	52
sphincs+-sha256-128s-simple	27787232	3576	52
sphincs+-sha256-128f-simple	468064	3576	52
sphincs+-sha256-128s-robust	50703309	3576	52
sphincs+-sha256-128f-robust	799354	3576	52
sphincs+-shake256-128s-simple	60598335	3576	52
sphincs+-shake256-128f-simple	988694	3576	52
sphincs+-shake256-128s-robust	112837474	3576	52
sphincs+-shake256-128f-robust	1777461	3576	52

Tabela 5. Resultados BenchmarkPEMEncode
Fonte: Original

Primeiramente, no número de alocações percebe-se o baixo e estável número dos

algortimos pós-quânticos, diferentemente dos clássicos, com números muito acima.

Na questão de quantidade de bytes alocados, nota-se a estabilidade e um um ótimo desempenho do algoritmo SPHINCS+ – baixo número de bytes alocados durante a operação. O algoritmo Rainbow possui um desempenho ruim neste teste, tendo resultados similares aos algoritmos clássicos. Enquanto isso, os algoritmos Dilithium e Falcon permanecem no meio termo, tendo resultados muito parecidos entre si.

Em relação ao tempo de execução da codificação, o algoritmo Dilithium demonstra um ótimo desempenho, possuindo um tempo bem abaixo dos outros candidatos. Os algoritmos Falcon e SPHINCS+, ficam no meio termo, enquanto o algoritmo Rainbow possui resultados muito ruins nesse quesito. Em comparação aos algoritmos clássicos, os pós-quânticos tiveram resultados similares na média, com exceção do RSA8192, que tem possui um tempo de execução muito acima de todos os outros – clássicos e pós-quânticos.

Outro ponto importante, tanto para o protocolo ACME quanto para outras aplicações criptográficas, é a capacidade de converter um arquivo em formato PEM para sua representação em código. Neste teste é medido o desempenho da conversão de certificados digitais no formato PEM para sua representação no cliente LEGO, como pode ser visto na Tabela 6.

	tempo (ns)	bytes	alocações
RSA8192	42067829555	42451064	40093
RSA2048	145540114	2169384	6162
EC384	12372332	7050216	58482
EC256	231626	23720	470
dilithium5	468058	78176	380
dilithium5-aes	284484	78104	378
falcon-1024	16621633	44928	379
rainbow-V-classic	-	-	-
rainbow-V-circumzenithal	-	-	-
rainbow-V-compressed	-	-	-
sphincs+-haraka-256s-simple	155735227	243360	382
sphincs+-haraka-256f-simple	15302718	448096	382
sphincs+-haraka-256s-robust	190877647	243360	382
sphincs+-haraka-256f-robust	19550675	448160	383
sphincs+-sha256-256s-simple	329667674	243256	380
sphincs+-sha256-256f-simple	44774934	448088	381
sphincs+-sha256-256s-robust	1175953080	243360	382
sphincs+-sha256-256f-robust	135193928	448088	381
sphincs+-shake256-256s-simple	782659983	243360	382
sphincs+-shake256-256f-simple	86942822	448088	381
sphincs+-shake256-256s-robust	1321532719	243288	380
sphincs+-shake256-256f-robust	154539119	448088	381
dilithium2	236493	46360	377
dilithium2-aes	205653	46360	377
falcon-512	4856045	29768	379
rainbow-I-classic	-	-	-
rainbow-I-circumzenithal	-	-	-
rainbow-I-compressed	-	-	-
sphincs+-haraka-128s-simple	91896425	72464	380
sphincs+-haraka-128f-simple	5011414	134600	379
sphincs+-haraka-128s-robust	103997347	72392	378
sphincs+-haraka-128f-robust	5400802	134672	381
sphincs+-sha256-128s-simple	237760159	72464	380
sphincs+-sha256-128f-simple	15264743	134672	381
sphincs+-sha256-128s-robust	430486768	72464	380
sphincs+-sha256-128f-robust	27404773	134672	381
sphincs+-shake256-128s-simple	514253342	72464	380
sphincs+-shake256-128f-simple	31012989	134672	381
sphincs+-shake256-128s-robust	972879596	72464	380
sphincs+-shake256-128f-robust	55752609	134600	379

Tabela 6. Resultados BenchmarkParsePEMCertificate
Fonte: Original

Primeiramente, nota-se a falta de resultados do algoritmo Rainbow. Isso se deve

ao fato de que seus certificados possuem tamanhos muito acima dos demais e resultam em problemas durante a execução de *benchmarks* na linguagem Go.

Em questões do número de alocações, os algoritmos clássicos realizaram um número muito maior enquanto os pós-quânticos estiveram ordens de grandeza abaixo nesse quesito.

Em relação ao número de bytes alocados, tem-se que o algoritmo Falcon obteve melhor desempenho, porém tendo resultados similares ao algoritmo Dilithium. Enquanto o algoritmo SPHINCS+ ficou um pouco acima nesse quesito, com algumas de suas variações a uma ordem de grandeza acima dos demais. Os algoritmos clássicos desempenharam, em média, pior que os pós-quânticos com exceção do algoritmo EC256.

Já na questão do tempo de execução, o algoritmo Dilithium foi muito melhor que os outros candidatos pós-quânticos e com resultados similares ao algoritmo EC do grupo dos clássicos. Os algoritmos Falcon e SPHINCS+ tiveram resultados similares, porém com tempos muito acima do Dilithium, com resultados semelhantes aos clássicos em média, com exceção do RSA8192 que teve um tempo muito acima dos outros.

Uma das principais etapas de autenticação do protocolo ACME é a realização do desafio ACME, onde o cliente demonstra o controle sob certo domínio. No seguinte teste é medido o desempenho da realização do desafio HTTP – como explicado em seções anteriores – utilizando cada um dos algoritmos para assinar as mensagens, como pode ser visto na Tabela 7.

	tempo (ns)	bytes	alocações
RSA8192	18454799487	17560720	16896
RSA2048	171146322	2820648	7707
EC384	4062628	1774040	14553
EC256	587367	72048	504
dilithium5	931623	100296	490
dilithium5-aes	787023	97936	498
falcon-1024	16847569	84872	481
rainbow-V-classic	4004382810	16322608	510
rainbow-V-circumzenithal	4524151349	5632096	494
rainbow-V-compressed	4525478996	4232768	509
sphincs+-haraka-256s-simple	11756282	80840	491
sphincs+-haraka-256f-simple	1239946	79128	490
sphincs+-haraka-256s-robust	13526494	66328	488
sphincs+-haraka-256f-robust	1542308	81224	497
sphincs+-sha256-256s-simple	25317550	78512	489
sphincs+-sha256-256f-simple	2368252	77968	492
sphincs+-sha256-256s-robust	98091443	61944	480
sphincs+-sha256-256f-robust	6262785	69896	479
sphincs+-shake256-256s-simple	60834800	72104	481
sphincs+-shake256-256f-simple	4013196	71064	495
sphincs+-shake256-256s-robust	102447339	80328	492
sphincs+-shake256-256f-robust	7077653	70024	481
dilithium2	753593	84440	488
dilithium2-aes	480571	70328	470
falcon-512	11994110	81816	495
rainbow-I-classic	106448431	1442568	485
rainbow-I-circumzenithal	118871755	658128	486
rainbow-I-compressed	117957904	550632	484
sphincs+-haraka-128s-simple	11241893	72160	492
sphincs+-haraka-128f-simple	883957	72072	488
sphincs+-haraka-128s-robust	12684016	78208	489
sphincs+-haraka-128f-robust	772967	73344	498
sphincs+-sha256-128s-simple	28730156	69672	483
sphincs+-sha256-128f-simple	1010456	64168	493
sphincs+-sha256-128s-robust	53483761	71488	493
sphincs+-sha256-128f-robust	1449081	72616	499
sphincs+-shake256-128s-simple	58646592	78520	492
sphincs+-shake256-128f-simple	1475393	59584	479
sphincs+-shake256-128s-robust	112387053	65752	469
sphincs+-shake256-128f-robust	2451097	80744	497

Tabela 7. Resultados BenchmarkChallenge
Fonte: Original

Novamente, o número de alocações para os algoritmos pós-quânticos se manteve

similar e bem abaixo do clássicos, como explicado anteriormente.

Em relação ao número de bytes alocados, percebe-se que o algoritmo SPHINCS+ se saiu melhor em alguns casos, porém, tanto ele quanto os algoritmos Dilithium e Falcon tiveram resultados muito similares. Já o algoritmo Rainbow teve um número de bytes alocados consideravelmente superior, assemelhando-se aos algoritmos clássicos em alguns casos.

Na questão do tempo de execução, o algoritmo Dilithium obteve resultados muito bons em relação aos outros candidatos, ficando ordens de grandeza abaixo. Em seguida, o algoritmo SPHINCS+, em alguns casos, obteve resultados muito bons, apesar de outros se assemelharem muito com os resultados do algoritmo Falcon. Já o algoritmo Rainbow obteve tempos de execução muito acima dos demais, resultando num pior desemepenho nesse quesito. Comparando com o desempenho dos algoritmos clássicos, os pós-quânticos tiveram resultados muito similares em média, com exceção do RSA8192 que teve tempos de execução muito superiores.

Por uma ótica geral, pode-se dizer que entre os algoritmos pós-quânticos o algoritmo Dilithium obteve ótimos resultados na maioria dos testes. Da mesma forma, os algoritmos Falcon e SPHINCS+ também tiveram resultados interessantes e, em alguns casos, melhores que o Dilithium. Notou-se também que o algoritmo Rainbow dentre os pós-quânticos não obteve resultados muito bons, geralmente ficando com os piores números – ou até em testes que não foi possível recuperar seus resultados.

Dentre os clássicos, percebeu-se que o algoritmo EC256 teve ótimo resultados, muitas vezes sendo o melhor colocado, com sua outra variação EC384 também tendo resultados bons. Já o algoritmo RSA2048 não teve resultados muito bons, porém em muitos testes assemelhou-se aos resultados do SPHINCS+ e Rainbow. No entanto, o algoritmo RSA8192 foi o pior em todos os testes em questão de tempo, e ficando com as outras métricas com resultados ruins também.

Para resumir os principais resultados do tempo de execução de cada teste, escolheu-se os melhores representantes de cada grupo de algortimos, i.e., RSA2048, EC256, dilithium2-aes, falcon-512, sphincs+-haraka-128f-simple, rainbow-I-classic e os comparou num gráfico. Onde no eixo vertical tem-se a média dos tempos (em escala logarítmica) de todos os testes e no eixo horizontal cada um desses algortimos. Decidiu-se utilizar o algoritmo EC256 como base de comparação, pois dentre os clássicos é um dos mais utilizados atualmente e obteve bons desempenhos nos testes. Dessa forma, os resultados foram normalizados em relação à média dos tempos do EC256.

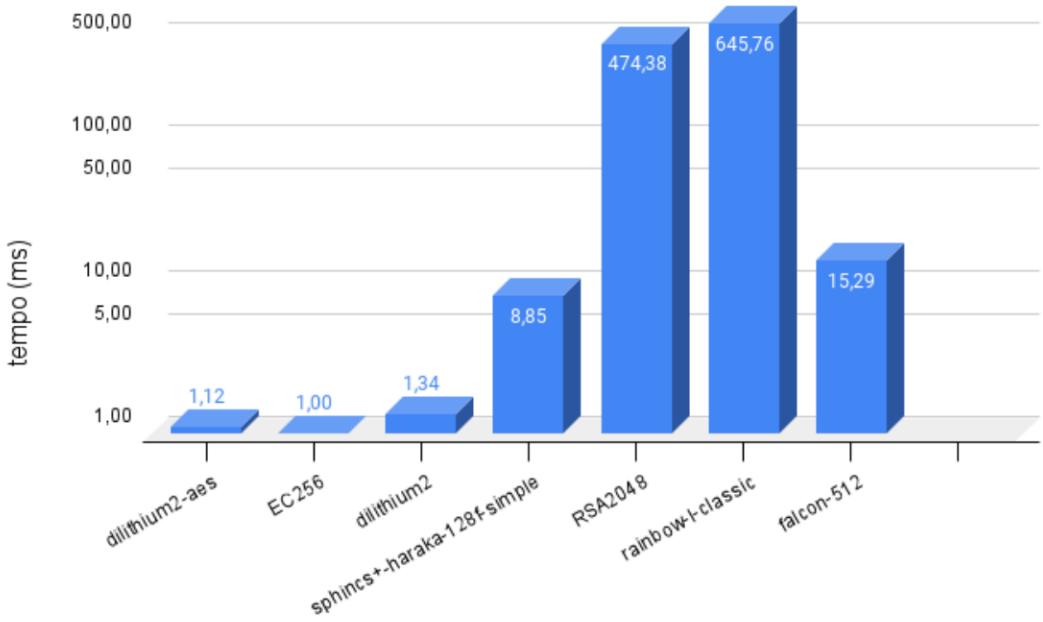

Figura 60. Comparação dos tempos de execução
Fonte: Original

Percebe-se uma consistência nos resultados, onde os algoritmo EC256, dilithium2-aes e o algoritmo dilithium2 obtêm os melhores tempos enquanto o algoritmo RSA2048 e o algoritmo rainbow-I-classic permanecem nas piores posições. Os outros algoritmos – sphincs+-haraka-128f-simple e falcon-512 – se mantém com resultados intermediários bons.

Outra comparação feita da mesma forma foi em relação aos bytes alocados, como pode-se ver no gráfico abaixo.

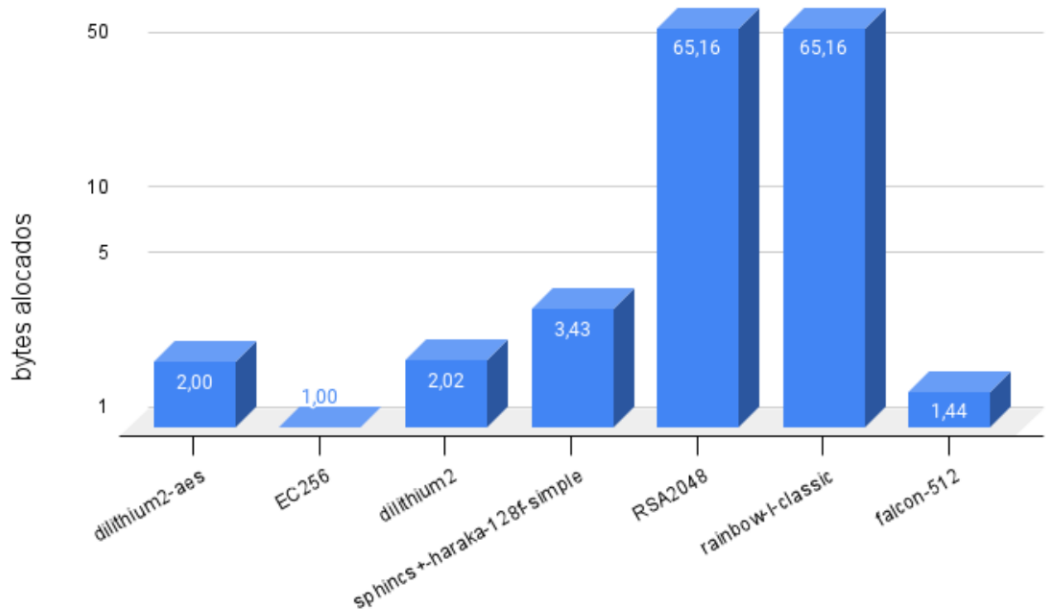

Figura 61. Comparação dos bytes alocados
Fonte: Original

Da mesma forma que nos tempos de execução, os algoritmos RSA2048 e rainbow-I-classic tiveram os piores resultados nesse quesito também. Entretanto, os outros algoritmos mantiveram resultados muito mais similares nessa comparação, apenas com o sphincs+-haraka-128f-simple se distanciando um pouco mais.

6. Considerações Finais

Cada vez mais vem se tornando uma realidade os computadores quânticos e o risco que eles impõe à criptografia clássica atual. Dessa forma, foi realizada uma extensão do protocolo ACME para que o mesmo suportasse a criptografia pós-quântica para que mesmo num ambiente de computadores quânticos o protocolo ainda se manter seguro e, dessa maneira, facilitando o tráfego seguro na internet.

Como visto, mudanças formais no protocolo não foram necessárias uma vez que o mesmo suporta a integração de diversos algoritmos de assinatura, não sendo restrito a algoritmos como RSA ou de curvas elípticas. Sendo assim, os esforços foram concentrados nas implementações do protocolo ACME, integrando com bibliotecas e fazendo as devidas alterações nos códigos do cliente e servidor ACME.

Realizadas as modificações, a última etapa teve foco nos testes de desempenho para cada um dos algoritmos integrados, comparando os resultados tanto entre os próprios algoritmos pós-quânticos quanto em relação aos algoritmos clássicos.

Com isso, finaliza-se o trabalho com uma revisão geral dos resultados dos testes, elucidando os principais resultados obtidos.

6.1. Trabalhos Futuros

Restringiu-se ao uso da biblioteca LibOQS, tal restrição traz consigo algumas desvantagens, como um desempenho não tão otimizado para cada algoritmo pós-quântico que foi utilizado devido a ser uma biblioteca que tem que atender a necessidade de todos os algoritmos a qual implementa. Entende-se esse conceito como um *wrapper* ou "empacotador", ou seja, cria um interface para utilizar todos os algoritmos que são suportados, diminuindo o desempenho dos mesmos – isso também é comentado em outros trabalhos, como em [Celi 2021]. Além disso, limita-se a interface disponibilizada pela LibOQS, tendo certas restrições à forma de integração com os outros repositórios.

Dessa forma, é possível contornar esses problema realizando a integração dos próprio algoritmos na biblioteca padrão do Go. Sendo assim, otimizações para cada algoritmo podem ser feitas na própria linguagem Go, obtendo resultados mais expressivos.

O uso de criptografia pós-quântica híbrida, i.e., a combinação de dois ou mais algoritmos, sendo um pós-quântico e outro clássico – no caso de dois algoritmos – de tal maneira que o esquema criptográfico se mantenha seguro mesmo que um dos algoritmos venha a apresentar vulnerabilidades. O desenvolvimento da criptografia pós-quântica prevê, em muitos casos, um estado híbrido para posteriormente avançar para um estado puramente pós-quântico [Paquin et al. 2020]. A adição de algoritmos híbridos implica em decisões de como realizar a combinação, como construir os certificados TLS com mais de uma chave pública, dentre outras questões. Resultados em relação aos algoritmos híbridos também são de interesse quando aplicados no protocolo ACME, sendo um possível próximo passo.

Referências

(2014). Hypertext transfer protocol (HTTP/1.1): Semantics and content. Technical report.

(2017). The JavaScript object notation (JSON) data interchange format. Technical report.

(2021). Golang. Github do Go.

(2021). Lego. Github do LEGO.

(2021). Let's encrypt stats. Website da Let's Encrypt.

(2021a). Liboqs. Github do projeto OQS.

(2021b). Liboqs go. Github do projeto OQS.

(2021). Pebble. Github do Pebble.

(2022). Liboqsoids. Github do projeto OQS.

(2022). Oid. Website do NIST.

(2022). Protocolo acme pós-quântico. Github do Protocolo ACME Pós-Quântico.

Abdullah, A. (2017). Advanced encryption standard (aes) algorithm to encrypt and decrypt data.

Albarqi, A., Alzaid, E., Alghamdi, F., Asiri, S., and Kar, J. (2015). Public key infrastructure: A survey. *Journal of Information Security*, 06:31–37.

Alidoost Nia, M., Sajedi, A., and Jamshidpey, A. (2014). An introduction to digital signature schemes.

Barnes, R., Hoffman-Andrews, J., McCarney, D., and Kasten, J. (2019). Automatic certificate management environment (ACME). Technical report.

Bernstein, D. J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., and Schwabe, P. (2019). The SPHINCS signature framework. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

Bernstein, D. J. and Lange, T. (2017). Post-quantum cryptography. *Nature*, 549(7671):188–194.

Celi, S. (2021). Implementing and measuring kemtls. In *Progress in Cryptology - LATINCRYPT 2021*, pages 88–107.

Coffman, K. G. and Odlyzko, A. M. (2002). Growth of the internet. In *Optical Fiber Telecommunications IV-B*, pages 17–56. Elsevier.

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. (2008a). Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Technical report.

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. (2008b). Internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Technical report.

Debnath, S., Chattopadhyay, A., and Dutta, S. (2017). Brief review on journey of secured hash algorithms. In *2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix)*, pages 1–5.

Dizdarević, J., Carpio, F., Jukan, A., and Masip-Bruin, X. (2019). A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys*, 51(6):1–29.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – HTTP/1.1. Technical report.

Gao, Z., Li, Z., and Tu, Y. (2004). Design and completion of digital certificate with authorization based on pki. In *Proceedings of the 2004 IEEE International Conference on Information Reuse and Integration, 2004. IRI 2004.*, pages 462–466.

Gien, M. (1978). A file transfer protocol (FTP). *Computer Networks (1976)*, 2(4-5):312–319.

Johnson, D., Menezes, A., and Vanstone, S. (2001). The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63.

Jones, M. (2015a). JSON web algorithms (JWA). Technical report.

Jones, M. (2015b). JSON web key (JWK). Technical report.

Jones, M., Bradley, J., and Sakimura, N. (2015a). JSON web signature (JWS). Technical report.

Jones, M., Bradley, J., and Sakimura, N. (2015b). JSON web token (JWT). Technical report.

Jones, M. and Hildebrand, J. (2015). JSON web encryption (JWE). Technical report.

Josefsson, S. and Leonard, S. (2015). Textual encodings of PKIX, PKCS, and CMS structures. Technical report.

Josefsson, S. and Liusvaara, I. (2017). Edwards-curve digital signature algorithm (EdDSA). Technical report.

Kaliski, B. (2008). Public-key cryptography standards (PKCS) #8: Private-key information syntax specification version 1.2. Technical report.

Karthik, S. (2014). Data encryption and decryption by using triple des and performance analysis of crypto system.

Kaur, N. and Sodhi, S. (2016). Article: Data encryption standard algorithm (des) for secure data transmission. *IJCA Proceedings on International Conference on Advances in Emerging Technology*, ICAET 2016(2):31–37. Full text available.

Kushwaha, P., Sonkar, H., Altaf, F., and Maity, S. (2020). A brief survey of challenge–response authentication mechanisms. In *ICT Analysis and Applications*, pages 573–581. Springer Singapore.

ling Chan, C., Fontugne, R., Cho, K., and Goto, S. (2018). Monitoring TLS adoption using backbone and edge traffic. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE.

Marqas, R., Almufti, S., and Rebar, R. (2020). Comparing symmetric and asymmetric cryptography in message encryption and decryption by using aes and rsa algorithms. *Xi'an Jianzhu Keji Daxue Xuebao/Journal of Xi'an University of Architecture Technology*, 12:3110–3116.

Mockapetris, P. (1987). Domain names - concepts and facilities. Technical report.

Moody, D., Alagic, G., Apon, D. C., Cooper, D. A., Dang, Q. H., Kelsey, J. M., Liu, Y.-K., Miller, C. A., Peralta, R. C., Perlner, R. A., Robinson, A. Y., Smith-Tone, D. C., and Alperin-Sheriff, J. (2020). Status report on the second round of the NIST post-quantum cryptography standardization process. Technical report.

Moore, N. C. and Halderman, J. A. (2021). How let's encrypt doubled the internet's percentage of secure websites in four years. Wiki do abnTeX2.

Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K., and Steenkiste, P. (2014). The cost of the "s"in HTTPS. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM.

Paquin, C., Stebila, D., and Tamvada, G. (2020). Benchmarking post-quantum cryptography in tls. In *PQCrypto*, pages 72–91.

Petitcolas, F. (2011). *Kerckhoffs Principle*.

Pornin, T. (2013). Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA). Technical report.

Sahu, A. and Ghosh, S. (2017). Review paper on secure hash algorithm with its variants.

Shannon, C. E. (1949). Communication theory of secrecy systems∗. $BellSystemTechnicalJournal, 28(4) : 656 - -715.$

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509.

Sobti, R. and Ganesan, G. (2012). Cryptographic hash functions: A review. *International Journal of Computer Science Issues, ISSN (Online): 1694-0814*, Vol 9:461 – 479.

Stallings, W. (2013). *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, USA, 6th edition.

Triantafyllou, A., Sarigiannidis, P., and Lagkas, T. (2018). Network protocols, schemes, and mechanisms for internet of things (iot): Features, open challenges, and trends. *Wireless Communications and Mobile Computing*, 2018:1–24.

Turau, V. (2003). Httpexplorer: Exploring the hypertext transfer protocol. volume 35, pages 198–201.

van Tilborg, H. C. A. and Jajodia, S., editors (2011). *Encyclopedia of Cryptography and Security*. Springer US.

Wardlaw, W. P. (2000). The RSA public key cryptosystem. In *Coding Theory and Cryptography*, pages 101–123. Springer Berlin Heidelberg.

Yalagandula, P. (2000). Dhcp: Dynamic host configuration protocol.