Automated Issuance of Post-Quantum Certificates: a New Challenge

Alexandre Augusto Giron[1][0000−0001−7668−7505], Frederico Schardong[2,3], Lucas Pandolfo Perin[4], Ricardo Custódio[2], Victor Valle[2], and Victor Mateu[4]

[1] Federal University of Technology - Parana (UTFPR), Toledo-PR, Brazil
[2] Federal University of Santa Catarina (UFSC), Florianópolis-SC, Brazil
[3] Instituto Federal do Rio Grande do Sul (IFRS) Rolante – RS – Brazil
[4] Technology Innovation Institute (TII), Abu Dhabi, UAE
alexandregiron@utfpr.edu.br

Abstract. The Automatic Certificate Management Environment protocol (ACME) has significantly contributed to the widespread use of digital certificates in safeguarding the authenticity and privacy of Internet data. These certificates are required for implementing the Transport Layer Security (TLS) protocol. However, it is well known that the cryptographic algorithms employed in these certificates will become insecure with the emergence of quantum computers. This study assesses the challenges in transitioning ACME to the post-quantum landscape using Post-Quantum Cryptography (PQC). To evaluate the cost of ACME's PQC migration, we create a simulation environment for issuing PQC-only and hybrid digital certificates. Our experiments reveal performance drawbacks associated with the switch to PQC or hybrid solutions. However, considering the high volume of certificates issued daily by organizations like Let's Encrypt, the performance of ACME is of utmost importance. To address this concern, we propose a novel challenge method for ACME. Compared to the widely used HTTP-01 method, our findings indicate an average PQC certificate issuance time that is 4.22 times faster, along with a potential reduction of up to 35% in communication size.

Keywords: Post-Quantum Cryptography · ACME Protocol · Certificate Management

1 Introduction

Encrypted data channels play a crucial role in ensuring data privacy on the Internet. One of the most widely used protocols for implementing these channels is the Transport Layer Security (TLS) [21]. However, the rapid and reliable issuance of digital certificates at minimal cost and the management of associated cryptographic keys throughout their lifecycle presents a bottleneck in the large-scale adoption of TLS. The widespread deployment of the protocol became possible only with the emergence of the Let's Encrypt project. Let's Encrypt's Certificate Authority (CA) has issued over 1 billion digital certificates and continues to experience substantial growth [6]. The success of Let's Encrypt can

be attributed to the automation of all necessary steps for issuing and renewing digital certificates. The automation of certificate issuance is facilitated by the Automatic Certificate Management Environment (ACME) protocol [1].

TLS and ACME protocols rely on classical cryptography to guarantee their security properties. However, the existence of Shor's quantum algorithm [27] gives an expiry date to the current protocols dated at the time a Cryptographically Relevant Quantum Computer (CRQC) [13] exists. This computer could compromise digital certificates and Key Exchange (KEX) mechanisms based on classical Public Key Cryptography (PKC). Consequently, attackers could collect transmitted data today with the anticipation of decrypting it using a CRQC in the future, a scenario known as "store-now-decrypt-later" attacks [3]. Such attacks would impact the security of existing protocols and applications dependent on TLS before a CRQC exists.

It is necessary to replace vulnerable algorithms to mitigate the quantum threats to classical cryptography. The cryptographic algorithms that can execute on a classical computer and offer security against attackers with access to a CRQC are called Post-Quantum Cryptography (PQC) [2]. The security of these cryptographic schemes relies on mathematical problems with no known efficient solutions for both quantum and classical computation. There is currently a significant global effort to evaluate and standardize post-quantum schemes. Regarding the adoption of these schemes, two primary strategies have emerged. The first strategy involves directly replacing classical algorithms with post-quantum ones. The second strategy, "hybrid mode" [3], utilizes both classical and post-quantum algorithms. Proponents of hybrid methods argue that post-quantum algorithms are relatively new and have not undergone the same level of scrutiny as classical algorithms. Their reasoning states that by including a classical algorithm alongside a post-quantum one, the security properties of the cryptographic protocol can still be guaranteed in case of a flaw or crypto-analytical attack on the post-quantum algorithm.

The transition from classical to PQC presents several challenges. One of the most relevant ones is the significantly increased size of cryptographic objects, such as public keys and signatures, and their impact on the protocol performance. For example, certain post-quantum algorithms like Classic McEliece are impractical for regular TLS handshakes due to the size of their public keys. To address this issue, researchers have conducted numerous benchmarks of PQC in network protocols like TLS [28,18], and others have proposed protocol changes to better accommodate PQC [24,25]. Such changes and evaluations are crucial to understand the performance implications imposed by PQC in advance. Therefore, adapting and evaluating protocol changes must be undertaken prior to the arrival of quantum computers to ensure a smooth transition to PQC.

Although several PQC-based TLS proposals and experiments have been proposed, we could not find any proposal for PQC in the context of ACME. Therefore, the impacts of using post-quantum schemes in such a scenario still need to be explored. In this paper, we address this gap by providing the following contributions:

- We integrate PQC schemes, namely Dilithium, Falcon, and Sphincs+, along with hybrid modes, into ACME implementations and the required libraries. Our modified implementations are publicly available.
- We evaluate ACME using geographically-distant peers, where the server is close to the Let's Encrypt CA location. Such a distance allows us to compare and estimate the impact of PQC on certificate issuance in a more realistic scenario.
- To expedite the certificate issuance process, we propose an alternative ACME challenge which can be used for issuing both classical and PQC certificates.
- We analyze the time and communication costs associated with our proposed challenge, demonstrating that it reduces both issuance time and byte cost for post-quantum cryptographic objects.

The remainder of this paper is organized as follows. Section 2 presents the necessary background concepts for understanding this work. Section 3 discusses quantum threats in ACME, the details of PQC integration, and the evaluation methodology. Section 4 presents our proposed ACME challenge design, its evaluation, and a discussion of the obtained results. Finally, Section 5 provides concluding remarks and outlines potential future work.

2 Background

First, we present the main characteristics of TLS and ACME. After that, we describe PQC concepts and the standardization process conducted by NIST. Finally, we conclude this section by showing related works about PQC adoption in network protocols.

2.1 TLS version 1.3

Formerly known as Secure Sockets Layer (SSL), the TLS protocol, in its current version (1.3), is described in RFC 8446 [21]. TLS provides a communication channel with confidentiality and authentication assurances between two peers: a client (e.g., a browser) and a server (e.g., a web server). TLS requires the server to provide authentication credentials when establishing a connection, while client authentication is optional.

The TLS 1.3 specification divides the protocol into three parts: (1) a Handshake protocol; (2) a Record protocol; and (3) an Alert protocol. The first part covers how the two communicating peers establish a session, aided by an Authenticated Key Exchange (AKE) and cryptographic computations ordered in a Key Schedule [21]. The second part covers how peers use their session data (and keys) to exchange application data securely, typically utilizing Authenticated Encryption with Associated Data (AEAD) algorithms. The last part covers how the peers should handle alert messages and protocol exceptions.

The mechanics of a complete TLS 1.3 handshake are as follows. First, a TLS client initiates the handshake by sending a `ClientHello` message. The message

can include several pieces of information, such as supported algorithms, cipher suites, and an extension message called `keyshare`. The `keyshare` is an ephemeral Elliptic Curve Diffie–Hellman (ECDH) public key used to create shared secrets for deriving symmetric keys. Upon receiving the `ClientHello`, the server responds with a set of messages: `ServerHello`, `Certificate`, `CertificateVerify`, `EncryptedExtensions`, and `Finished`. The server hello includes information about algorithm selection, the corresponding ECDH `keyshare`, and additional extensions (if available). The server provides a set of certificates, a digital signature, and an HMAC [15] to authenticate over the handshake transcript data (`Certificate`, `CertificateVerify`, and `Finished` messages, respectively). Except for the `ServerHello`, all messages are encrypted using keys derived from the `keyshare` pair. The `EncryptedExtensions` message, sent immediately after the `ServerHello`, is also encrypted.

The client receives the server's response and processes it. It verifies the handshake signature, validates the certificates, and the `Finished` message. Additionally, the client checks if the server's reply includes the optional `CertificateRequest` message. If it does, the client will authenticate using a certificate and a handshake signature with its private key. Otherwise, it sends the mandatory `Finished` message and any desired application data to the server, concluding the handshake and initiating secure communication. TLS is commonly used in upper-layer network protocols like HTTPS and network applications like OpenVPN. In this work, we focus on using TLS by the ACME protocol.

2.2 ACMEv2 Characteristics

The Automated Certificate Management Environment (ACME) protocol is defined in RFC 8555 [1]. ACME offers services for verifying identity over the Internet and managing certificates. The primary objective of the protocol is to minimize the need for human intervention in configuring web servers and handling certificates. ACME enables an ACME server (controlled by an Issuer CA) to issue a Domain-Validated (DV) digital certificate to the ACME client. The issuance and domain validation processes are fully automated. Currently in its version 2, ACME plays a crucial role in Let's Encrypt, one of the largest CAs on the Internet. Moreover, many certification authorities and PKI vendors, such as ZeroSSL [30], are adopting the ACME protocol in their products because it simplifies and enhances the quality of service provided to their customers.

ACME relies on two communication channels: (1) the ACME Channel, protected by TLS; and (2) the Validation channel, which depends on the validation method. An ACME client uses TLS to request the issuance of one or more DV certificates from an ACME server. ACME servers store ACME client accounts associated with a public-key pair that clients use to authenticate themselves to the server. However, the server only issues a certificate after the client proves control over the desired identifier to be certified, i.e., the domain name. To accomplish this, the client must solve an ACME challenge. RFC 8555 [1] specifies the HTTP and DNS challenge types, and RFC 8737 [26] describes the TLS-ALPN challenge. Generally, a challenge is considered fulfilled if (a) the client

proves control of the private key associated with the ACME account and (b) the client proves control of the domain name in question.

ACME protocol messages are based on the JSON Web Signature (JWS) standard [9] and transmitted through HTTPS/TLS requests. Typically, ACME HTTPS requests are signed using the account's private key, while the public key is usually not included in the JWS body. However, when creating a new account or revoking a certificate, the "jwk" field (i.e., the public key) is included in the request. Other requests identify keys using a "Key ID" ("kid") field in the request [9]. This way, the server can determine which key to verify subsequent requests.

Figure 1 illustrates the necessary ACME messages for issuing an X.509 certificate. The issuance process is divided into three steps: (1) account creation; (2) challenge; and (3) issuance. Communication between the ACME client and server occurs through HTTPS requests, requiring the ACME client to trust the ACME server. This trust is established by the ACME client's confidence in the server's certificate chain, which includes intermediate and root CAs. Typically, root CAs are pre-installed in the client's certificate repository.

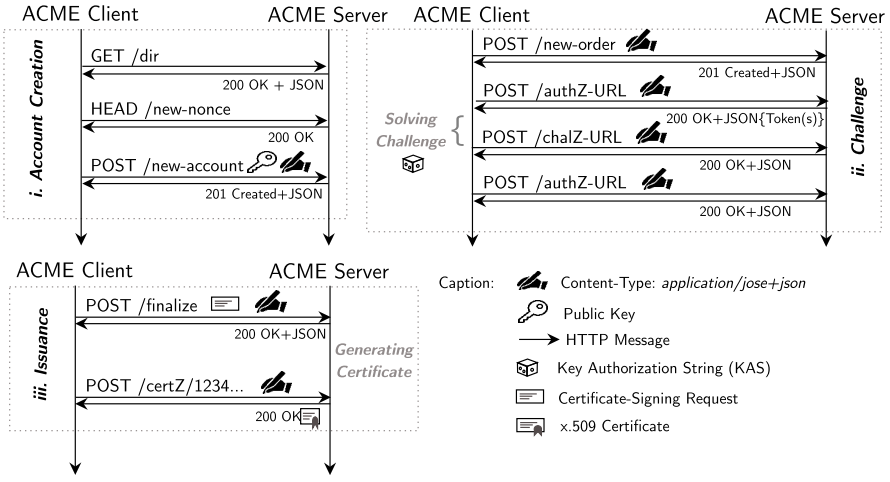

Fig. 1. ACME Issuance Overview

The client initiates an account creation request with the ACME server in the first step. The client's account can optionally include contact information and is associated with a key pair generated by the client. To initiate the creation process, the client requests server resources by sending a `GET /dir` message. The server responds with an HTTP code (typically 200 for success) and a JSON payload. The JSON payload contains the URLs for the desired resources and the Terms of Service. If it is the client's first connection, a new nonce is required. The

client obtains the nonce by sending a `HEAD /new-nonce` message. This nonce is used to protect against possible replay attacks. The registration is concluded with a `POST /new-account` request. At this point, it is important to note that the ACME server does not have any means to confirm the claimed identity other than the newly registered authentication key, referred to as the "account key". Subsequent HTTP requests from the client must be signed with the account key.

The second step aims to prove the client's identity through an Identifier Validation Challenge [1]. The ACME protocol specification focuses on domain name identifiers. There are different types of challenges available, such as HTTP-01, DNS-01, and TLS-ALPN-01, with HTTP-01 being the most commonly used [5]. In general, to complete the challenge, the client must demonstrate possession of the account key and control over the identifier. In the case of HTTP-01, the client must serve a file over HTTP containing the Key Authorization String (KAS). A KAS is formed by concatenating a 128-bit random token (previously generated by the server), a dot separator ('.'), and the base64-encoded key fingerprint. The ACME server retrieves and checks the file over HTTP to validate the challenge. Refer to Appendix A for additional details on HTTP-01.

Figure 1 provides an abstract representation of the challenge-solving step. First, the client requests a new certificate by sending a `POST /new-order` message. The server's response includes information about the available challenges, their respective URLs, and the KAS for each challenge. Each challenge requires a unique KAS generated on demand, meaning authorization requests can fail, and the client may need to retry them. Additionally, each challenge has a state (e.g., pending, valid, deactivated), allowing the server to expect multiple requests using the same KAS until the certificate is issued. Therefore, the client must check the status of the desired KAS by sending a `POST /authZ/...` request and then proceed with the relevant challenge.

After completing the challenge, the client sends a `POST /chal/...` message to inform the server that the challenge has been completed, and it waits for the server to validate the challenge. The client can check the challenge's status by sending `POST /authZ/...` requests. Once the challenge is deemed valid by the server, it is considered completed. The server stores the authorization and marks it as valid for a specific period (not controlled by RFC 8555 [1]).

The issuance step, as depicted in Figure 1, is the final part of the process. The client sends a `POST /finalize` message, which includes a PKCS#10 Certificate-Signing Request (CSR) [17], to the server. It is important to note that the account key pair used for the CSR generation differs from the one used for the account registration. Specifically, the CSR must not contain a public key for any known account. The server validates the CSR and generates the certificate. Finally, the client can download the issued certificate using a `POST /certZ/...` message, often referred to as "POST-AS-GET" [1]. Once the client has obtained the certificate, the ACME client's request flow is complete. ACME client implementations like Certbot [8] typically store and automatically configure the certificate(s) in the web server repository. It enables the seamless activation of an HTTPS-secured web server with just a few command-line instructions. Ad-

ditionally, Certbot configures automatic certificate renewal, thereby simplifying certificate management operations. It is worth mentioning that RFC 8555 does not distinguish between certificate issuance and renewal, meaning the renewal process starts with a new request to `/new-order`.

2.3 Post-Quantum Cryptography

Post-Quantum Cryptography (PQC) or Quantum-Safe Cryptography is an area of research that focuses on developing cryptographic algorithms that are resistant to attacks from quantum computers. Traditional public-key schemes based on problems such as the Discrete Logarithm Problem (DLP), Elliptic Curve Discrete Logarithm Problem (ECDLP), and Integer Factorization Problem (IFP), are considered to be vulnerable to attacks by quantum computers, specifically Shor's algorithm [27].

The threat of quantum computers to current cryptographic systems raises concerns about the confidentiality and authentication of data transmitted over the internet. While the impact on confidentiality is more immediate, as an adversary can gather encrypted data today and decrypt it in the future with the help of a quantum computer, the impact on authentication is less urgent since quantum adversaries cannot retroactively impersonate past communications [3].

In this context, efforts are underway to standardize post-quantum algorithms. One notable initiative is led by the National Institute of Standards and Technology (NIST) [14]. NIST has been running a standardization process for PQC algorithms, including key exchange, public-key encryption, and digital signatures. The initial choice of standards includes Kyber for key exchange and public-key encryption, as well as Dilithium, Falcon, and Sphincs+ for digital signatures. These algorithms have gone through multiple rounds of evaluation, and the process is currently in the fourth round, with additional schemes under scrutiny [16].

Regarding the impact on the ACME protocol and TLS, the transition to post-quantum cryptography will involve replacing current signature algorithms with post-quantum digital signature schemes. However, the transition process is expected to take significant time, as it requires coordination among various entities such as certificate authorities (CAs), client and server implementations, and browsers. Therefore, it is crucial to experiment, evaluate, and plan for a smooth transition to post-quantum cryptography in ACME and TLS [10]. Table 1 provides an overview of the post-quantum signature schemes expected to be standardized by NIST, along with their sizes and corresponding security levels.

There is limited work specifically focusing on the issuance of post-quantum certificates. Two main methods have been proposed for implementing hybrid post-quantum certificates within the X.509 standard format. One method involves concatenating cryptographic objects, such as public keys and signatures, while the other adds PQC algorithm information as X.509 extensions. The second method uses non-critical extensions and minimizes the risk of compatibility issues with legacy implementations that do not support post-quantum algorithms. Security analyses have been conducted to evaluate the effectiveness of these combining methods [4]. The impact of post-quantum certificates on PKI

operations and TLS connections has been discussed in the literature, highlighting concerns about performance, particularly when dealing with the certificate chain. However, there are often no objections to using the hybrid mode, which combines both classical and post-quantum algorithms, regarding performance penalties [20,11].

Table 1. Currently digital signature schemes to be standardized by the NIST PQC process.

Algorithm Parameter Set Name	NIST Security Level	Public key size + Signature size (bytes)
Dilithium2	1	3732
Dilithium3	3	5245
Dilithium5	5	7187
Falcon-512	1	1587
Falcon-1024	5	3123
SPHINCS+-SHAKE256-128s-simple	1	7888
SPHINCS+-SHAKE256-128f-simple	1	17120
SPHINCS+-SHAKE256-192f-simple	3	35712
SPHINCS+-SHAKE256-192s-simple	3	16272
SPHINCS+-SHAKE256-256f-simple	5	49920
SPHINCS+-SHAKE256-256s-simple	5	29856

3 Quantum Threat and PQC Adoption

We begin by examining the threats to ACME security in the presence of a quantum computer in Section 3.1. Subsequently, we delve into implementation and design specifics in Section 3.2. Finally, we explore the implications of evaluating ACME with PQC in Section 3.3.

3.1 Quantum Threats in ACME

The ACME protocol relies on PKC to ensure its cryptographic properties. Consequently, once a CRQC exists, the protocol would become insecure. While the threat exists, the transition to PQC may not be as urgent for ACME compared to other cases, given that most interactions are certificate-related. However, RFC 8555 [1] specifies that a secure channel, often implemented using TLS, must be used for client requests to the server. Therefore, a quantum-safe ACME implementation depends on a quantum-safe TLS. To prevent "store-now-decrypt-later" attacks, a quantum-safe Key Exchange (KEX) algorithm must be used before a CRQC arrives. It is worth noting that the challenge validation channel in ACME does not necessarily require TLS.

One of the benefits that ACME provides to clients is the ability to reuse valid authorizations. After completing a challenge, a client can reuse the authorization to issue a new certificate more efficiently. This feature allows clients to issue certificates at their convenience, not necessarily immediately after challenge validation. However, it introduces a potential vulnerability in the form of a store-now-decrypt-later attack. An attacker could collect TLS-encrypted ACME messages and, in the future, exploit a hypothetical quantum attack on the TLS layer to gain access to the ACME information containing challenge authorization details. Since RFC 8555 [1] leaves the deactivation of authorizations up to implementations, many challenge authorizations could remain valid for an extended period. As a result, an attacker could exploit old valid authorizations to issue unauthorized certificates. Figure 2 illustrates the attack. Therefore, the authorization reuse feature needs careful redesign considering the existence of future CRQCs. More details about authorizations and their validity times are discussed in Section 4.3.

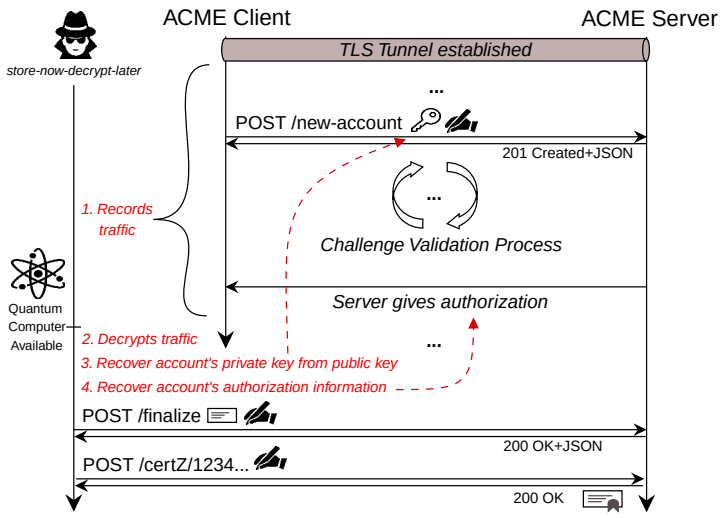

Fig. 2. Unauthorized issuance of a certificate with the help of a quantum computer.

Both attack scenarios, targeting classical certificates and the classical communication channel, can be mitigated by using PQC.

3.2 Integrating PQC algorithms

We selected PQC implementations from the Open Quantum-Safe project `liboqs` [29]. Since our project is developed using the Go language, we used the liboqs-go binding [19]. We integrated them into Pebble's ACME server and

LEGO ("Let's Encrypt Client and ACME Library in Go"). Pebble is suitable for testing ACME client implementations. For reproducibility, our ACME implementations and test scripts are publicly available[5]. We used the selected standard candidates from the NIST PQC standardization process for integration:

- **Kyber**: for Key Exchange in TLS, using security levels 1,3 and 5.
- **Dilithium** and **Falcon**: we use the same algorithm and security level parameters in all required cryptographic objects. Namely: ACME client account keys and CSR; ACME server digital certificate (TLS level); issued certificates; and the certificate chains of issued certificates (Root CA certificate and Intermediate CA certificate). For simplicity, we did not change Pebble's certificate chain size for TLS. We only alter Pebble's TLS chain to use PQC algorithms without adding a new Intermediate CA certificate.
- **Sphincs+**: due to its increased signature sizes, we restrict Sphincs+ only for the Root CA certificate. We omit Root CA certificates in TLS handshakes, so Sphincs+ increased sizes are not transmitted in the handshake. Sphincs+ selected parameters are: `SHAKE` for the hash function, "s" for compact signatures and improved verification timings, and "simple" for performance.
- **Hybrid modes**: using NIST P-curves, namely `P256`, combined by concatenating with Kyber, Dilithium, and Falcon cryptographic objects. For simplicity, we opted to concatenate cryptographic objects into certificates (public keys and signatures). Hybrids are recommended because the confidence in PQC security is not well established yet [3], but also because RFC 8555 states "MUST/SHOULD implement" for some classical algorithms [1], thus keeping our integration close to the specification. We refer to the hybrid mode using the 'H' letter (e.g., "Dilithium H.").

3.3 Impacts of PQC in ACME

To better understand the consequences of using PQC in ACME, we run several experiments using two geographically distant Google N2 Virtual Machines (VMs) with identical configurations (8 GB memory, two vCPUs). The ACME client VM was hosted in Osasco, São Paulo, Brazil, while the ACME server location was based on one of Let's Encrypt's data centers in Salt Lake City, Utah, USA. The average round-trip time (RTT) for this geographically distant network was measured to be 157 ms. The number of successful requests was computed by employing 1024 threads to POST requests to the `/finalize` endpoint for six minutes. Each thread simulated a different client sending CSRs, thereby increasing the server's load during certificate issuance. We set `ulimit -n 1048576` to enhance the server's load test configuration.

Figure 3 illustrates the impacts of PQC observed during a load test experiment. For automation purposes, the default option is to generate a CSR during protocol execution, which we refer to as the "CSR-on-the-fly" test. This approach includes key generation and signing computational times, resulting in delayed

[5] https://github.com/AAGiron/acme-newchallenge

clients and fewer successful requests handled by the server. Alternatively, using a pre-computed CSR can reduce the PQC impact at the cost of some of ACME's automation properties.

From the CA's perspective, the results demonstrated a noticeable impact when deploying PQC in the standard ACME configuration. The reduced number of successfully handled requests implies fewer certificates generated and issued by the ACME server. Furthermore, larger PQC objects can congest the network earlier than the baseline configuration (see Section 4.3).

It is worth noting that our experiments did not provide an exact measurement of the number of certificates issued per second due to the nature of the protocol. However, our load test is representative as it involves handling multiple signed requests, CSR generation by client threads, and verification by the server.

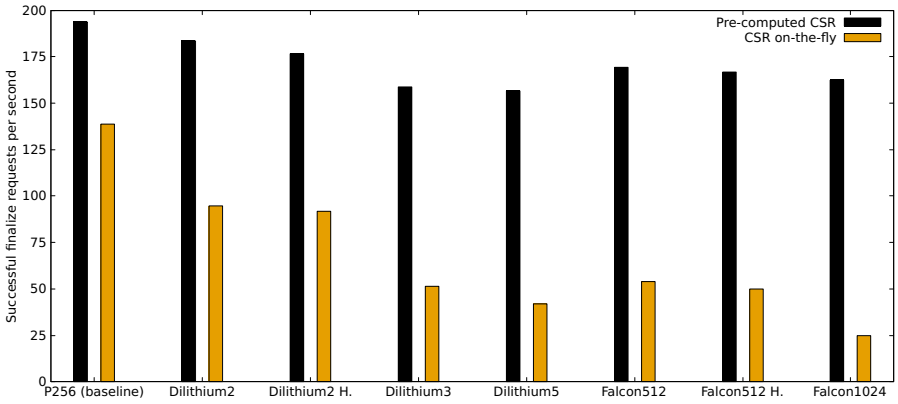

Fig. 3. Load test experiment with and without CSR cryptographic operations.

4 Proposed ACME Challenge

In order to speed up the issuance of digital certificates, we propose an alternate ACME challenge. In this section, we present our proposed ACME challenge (Section 4.1). After that, we evaluate and compare our proposed challenge against standard ACME certificate issuance and renewal. Lastly, we discuss the experimental findings in Section 4.3.

4.1 Design details

We can consider two general scenarios when an ACME client C will ask an ACME server S for a new certificate. In the first scenario, C already has a classical certificate, so C can ask: (a) for a renewal, using the same ACME account; (b) for a new classical certificate (new account); or (c) a new PQC (or

hybrid) certificate. In the second scenario, C does not have a classical certificate: in this case, C can only ask for a new PQC (or hybrid) certificate.

In the first scenario, we assume that C already has a previously issued certificate. Having a certificate means that these ACME peers have a relationship that could be used to optimize the certificate issuance process. In the second scenario, there is no previous relationship available. Therefore, for the second scenario, C must comply with all ACME requirements, i.e., fulfill the account creation, challenge validation, and issuance steps.

As described in Section 2.2, the issuance flow has several digitally-signed requests between peers. Using PQC signatures in such requests would increase protocol communication costs and impact the overall interaction between those peers. Also, one could take advantage of the scenario in which the server already has a certificate. To speed up the issuance process, we propose a new ACME challenge, depicted in Figure 4.

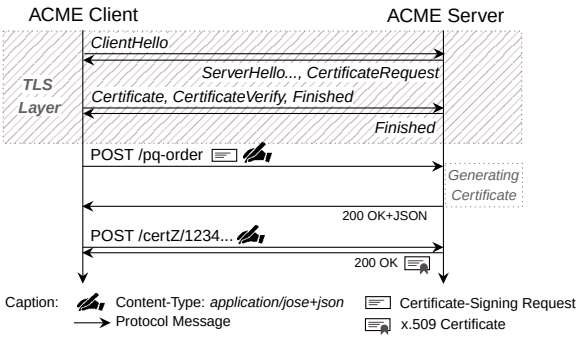

Fig. 4. Proposed ACME Challenge

Note that our proposal is valid for the scenario in which ACME clients already have a certificate. We provide an alternative to the original `/new-order` ACME server endpoint, called `/pq-order`. This new endpoint (at the server) expects a CSR in an HTTP POST message, as the usual `/finalize` endpoint. The main difference is that it requires a mutually authenticated TLS handshake. Mutual authentication means that the ACME client authenticates directly in the TLS layer, proving that it possesses the private key of that certificate. If the client successfully authenticates to the server, the server can issue the new (PQC or hybrid) certificate, replying with the URL where the certificate can be downloaded.

The fact that the ACME client already possesses a certificate plays a crucial role in this approach. For example, let $C_{classic-cert}$ be the certificate the client is willing to use in the TLS authentication layer, and $C_{pqc-cert}$ the certificate the client requests. If $C_{classic-cert}$ was issued by the same ACME server where $C_{pqc-cert}$ will be requested, then the peer trust relationship is already estab-

lished. The ACME server will trust $C_{classic-cert}$ (in the pre-quantum scenario), so additional configuration or protocol messages are unnecessary. In this example, the ACME client can ask for a PQC certificate with this new challenge in a single request. Comparatively, we remove (at least) 4 signed requests from the ACME flow and replace the challenge with TLS client authentication using the $C_{classic-cert}$.

Since the `/pq-order` is an endpoint of the ACME server, clients perform POST requests with their account information accompanied by a CSR. In this case, the CSR can be created using a PQC algorithm (hybrid or not), allowing the issuance of a post-quantum certificate. Note, however, that the signature present in the request also uses a post-quantum algorithm. Appendix B gives an example of a POST message. Additionally, our proposed challenge applies to clients willing to issue a classical certificate, if desired.

Regarding the request validation, the server uses the algorithm name, nonce, and key ID (`kid`) information to search for the required account information. Note that the `kid` field can be replaced by the public key in a field called `jwk` for verifying the message. The optional `certhash` value is a way of binding the request to the particular certificate used in the TLS mutual authentication. In this way, the server can check if the hash is on his list of issued certificates and if it belongs to the corresponding account in the request. Alternatively, the ACME server can obtain the client certificate from the TLS layer and compare domain names and hashes. Golang provides access directly through the standard library [12]. The ACME server processes the CSR as usual. If the validation is successful, the server can issue the certificate.

Security Considerations. RFC 8555 describes a threat model against active and passive attackers considering two communication channels: the ACME channel, using TLS for security, and the Validation channel, which is dependent on the ACME challenge (e.g., HTTP). Since the validation channel is bound to the signatures transferred in the ACME channel, abusing only the validation channel should not be enough to impersonate a legitimate client (i.e., obtain a valid authorization).

Regarding the ACME channel, the only thing we changed is that it uses PQC algorithms. On the other hand, our proposed challenge replaces the available validation channels from the original ACME challenges with a mutually authenticated TLS connection channel. Note that our proposed challenge requires a valid mutual authentication TLS session and a valid signature in the request. Therefore, our challenge keeps the binding between the validation and ACME channels, thus not deviating from the RFC's threat model. The main requirement is a mandatory client authentication policy since client authentication is optional in TLS. An additional consideration is to avoid TLS Post-Handshake Authentication [21] because the ACME server can issue the certificate only after the mutually authenticated connection is established.

Our proposed challenge assumes that who owns a certified (and valid) key pair for a particular domain owns the identifier in question, i.e., the domain.

This might not be directly applicable in some cases, such as hosting providers. For example, when the domain ownership is transferred, the original owner could use the certificate to obtain a new one through our proposed challenge. Although this is a problem, it could be mitigated by simply revoking the certificate before transferring a domain. If revoked, the certificate can not be used to authenticate in our proposed challenge. Therefore, the server will not issue a new certificate in this case. This requirement implies keeping the certificate's validity period within the granted domain ownership validity period.

4.2 Issuance and Renewal Timings

We use the same experiment methodology as described in Section 3.3. In this case, the issuance time was measured at the client and encompassed all ACME steps (depicted in Figure 1) until the client obtained its certificate. The renewal time was considered as a new issuance process by requesting the `/new-order` endpoint without creating a new account. Consequently, this metric measured the time from the `/new-order` POST request until the client received the certificate. Both renewal and issuance times were computed from 500 protocol executions (resulting in 500 certificates per algorithm instance) to obtain the average and standard deviation statistics.

Figure 5 shows the issuance and renewal times for ACME with baseline (classical) and PQC compared to our proposed challenge. The bars correspond to average timings, and the graph includes standard deviation information (above the bars). All standard deviations obtained from our proposed challenge executions are below 10 ms, whereas in standard ACME, it reaches 1.4 seconds. All bars are below the baseline standard deviation (using NIST's P256), which suggests no PQC transition impact in the timings perceived by the ACME client.

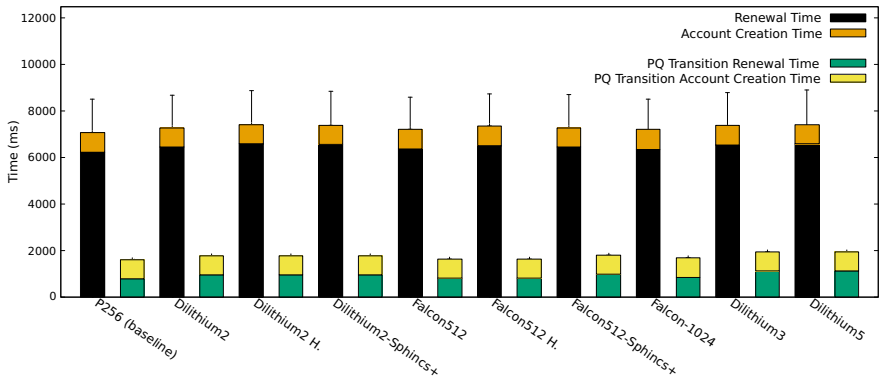

Fig. 5. Issuance and renewal average timings for different PQC algorithm instantiations. Note: Issuance time is the sum of Account Creation and Renewal time.

From the ACME client's perspective, Figure 5 shows that the average impact in PQC is not significant. Here the network time dominates, and ACME's query-response nature increases the variations (as shown by the standard deviations). Sphincs+, `Dilithium3`, `Dilithium5`, and `Falcon-1024` also do not greatly influence the timings, so these configurations are also viable. On average, it took near to 7.5 seconds to issue a classical, PQC, or hybrid certificate.

On the other hand, our results suggest that the issuance and renewal time can be significantly reduced using our proposed challenge. The issuance times are, on average, 4.22x faster compared to the commonly-used HTTP challenge. Renewals are also much faster: without the account creation time, our renewals are near or below 1s (on average), regardless of the algorithm selected for the new certificate (P256 or PQC). We highlight that our proposed challenge can be used generically, both for renewing classical certificates or issuing PQC certificates.

4.3 Discussion

In the context of PQC, we expected a significant slowdown in issuance and renewal times due to the increased sizes of PQC instances. For example, using `Dilithium2` imposes a payload size of 64.21 KiB on the network. However, this byte amount is divided among several request messages in ACME (as depicted in Figure 1). Assuming at least seven signed requests in ACME, each carrying less than 10 KB (except for certificate download), the data can be transported within a single round trip without requiring additional RTTs, assuming a standard TCP/IP network stack. Furthermore, we are not transmitting the Sphincs+ certificate, which saves bytes and keeps the size below network limits, such as the TCP window size. While literature shows scenarios where PQC imposes additional RTTs in other designs [28], issuance times depend not only on RTTs but also on the variable number of requests and waiting times. Our results indicate that the average issuance time for PQC was close to the baseline.

We were able to modify ACME and achieved better performance under reasonable assumptions, such as the client already having a classical certificate. By transitivity, having control of the certificate that certifies a domain demonstrates control over that domain, even if the certificate's private key is not stored on the server. In this scenario, our proposed modification reduced the byte costs of ACME. Table 2 illustrates the impacts of PQC on ACME and the sizes of our proposed challenge. Since not all ACME requests are used in our challenge, it reduces network RTTs. Compared to the original ACME message flow, our challenge saves 35.39% and 32.14% for `Dilithium2` and `Falcon-512` instantiations, respectively.

It is important to note that our challenge differs from the TLS-ALPN-01 challenge. Defined in RFC 8737 [26], the ACME client generates a self-signed X.509 certificate with the challenge information, such as the KAS, and starts the TLS server under its control. The ACME server performs a handshake with this new TLS server to check the required information. However, the TLS-ALPN-01 challenge focuses on cases where the web service providing content is separate from the TLS server, such as reverse proxies. Additionally, the DNS-1 challenge

Table 2. Comparison of sizes of ACME client requests, sampled from a pcapng capture file. Note: "Total (ACME)" excludes repeated requests (like POST /authZ); however, in practice, more bytes are transmitted (see Section 2.2). Certificate sizes include server and intermediate CA certificate.

Request	Request is in our challenge?	P256 (baseline) size (bytes)	Dilithium2 size (bytes)	Falcon-512 size (bytes)
GET /dir	✓	223	223	223
HEAD /new-nonce	✓	207	207	207
POST /new-account	✓	718	6097	3010
POST /new-order	✗	594	3747	1399
POST /pq-order	✓	1194	10558	4336
POST /authZ	✗	624	3776	1423
POST /challZ	✗	627	3779	1425
POST /finalize	✗	1088	10648	4417
POST-AS-GET /certZ	✓	649	3713	1361
Total (ACME*)	-	**4730**	**32190**	**13465**
Total (Our Challenge)	-	**2991**	**20798**	**9137**
Certificate size	-	1913	17838	7157

can take up to one hour or up to five minutes under specific conditions [22], making it unsuitable for direct comparison against our challenge. Since the HTTP challenge is the most commonly used, our experiments focused on this scenario.

While our proposed challenge provides faster issuance times, it is not meant to replace other existing ACME challenges. There may be scenarios where our challenge is not suitable. One example is when the client does not have a classical certificate. Another example relates to the validity period of certificates and the reuse of valid authorizations, as allowed in RFC 8555 [1].

RFC 8555 [1] does not impose a limit on the expiration time of authorizations, leaving the validity period of a valid authorization to the implementation. For instance, Let's Encrypt's current policy allows reuse for up to 30 days. Therefore, if an HTTP challenge has been fulfilled, the ACME client has 30 days to issue or renew certificates, improving performance by skipping the challenge step. However, this 30-day policy is subject to change [7] and may vary or be denied in other implementations. On the other hand, our challenge's validity is limited to the certificate's validity period (currently 90 days in Let's Encrypt's policy). In the context of PQC transition, we highly recommend deactivating authorizations of accounts created with classical cryptography. Deactivation is necessary because ACME servers cannot guarantee that the TLS connection established by ACME clients is quantum-safe. Non-PQC TLS usage by clients and valid authorizations facilitate quantum attacks, as discussed in Section 3.1.

Nevertheless, our proposal improves performance for issuing certificates (including account creation time) and renewals (assuming the client has an account with the server). In scenarios where our challenge's assumptions hold, ACME clients can utilize our approach for renewing classical certificates faster, or during the PQC transition phase and subsequently renew their PQC certificates. For security reasons, Issuer CA policies can impose usage limits on clients renew-

ing with our challenge. These limits can reduce the impact of a certificate's key compromise, forcing the client to prove ownership using a different challenge.

Our proposed challenge can be further optimized if additional modifications are made at the TLS layer. Specifically, mutual authentication in TLS involves transferring certificates over the network, increasing the size of TLS messages. RFC 7924 [23] specifies certificate caching mechanisms (client or server), which could be employed in ACME's TLS channel to reduce the TLS payload size.

5 Final Remarks and Future Work

This work provided a comprehensive evaluation of ACME's performance when secured with PQC algorithms, considering the perspectives of ACME clients (e.g., web servers) and servers (e.g., Issuer CAs). The comparison against classical cryptography highlighted different impacts on these entities.

Regarding challenges required for the certificate issuance process, our proposed design showed favorable results. We achieved smaller communication sizes and decreased network bandwidth by replacing the HTTP challenge and eliminating associated signed requests. To encourage practical adoption, we have made our design and prototype implementation available to the community. We have also provided an RFC-like description of our challenge as a guide for future implementations.

There are interesting opportunities for further research and evaluation of ACME. For instance, investigating ACME's performance in different computing environments, such as the Internet of Things (IoT), would be valuable. Additionally, exploring how ACME performs when issuing certificates for KEMTLS, a key encapsulation mechanism-based TLS, could provide valuable insights. It is worth noting that issuing KEM-based certificates in ACME poses challenges due to the typical usage of CSRs with signature methods. Nonetheless, ACME remains a significant security-enabling protocol that has already benefited various applications and is likely to continue doing so in the future.

Acknowledgements This work was supported by the Federal University of Technology - Parana (UTFPR) and the Technology Innovation Institute (TII).

Appendix

A ACME's HTTP-01 Challenge

Figure 6 focus on the HTTP challenge message flow, which is more commonly used, probably due to its simplicity. We omit account creation messages, order and download requests. First, the client obtains the necessary information for the challenge (e.g., KAS) with the steps presented in Figure 1. Basically, the client places the KAS file in (one or more) HTTP servers that it controls. Therefore, the KAS binds the HTTP server to the ACME client's account. Then, the

client notifies the server with a POST to `/challZ` endpoint. The validation steps include checking the response (e.g., if the domain name matches the previous order information) and, most importantly: (i) if the KAS inside the downloaded file matches; and (ii) if the digital signatures (in the requests) can be verified using the corresponding account's public key. Otherwise, the challenge fails.

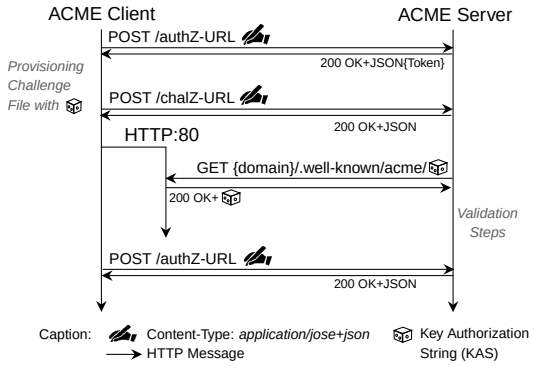

Fig. 6. HTTP challenge flow

In practice, the HTTP challenge (and the other types) can consume more POST requests to `/authZ` endpoint than shown in Figure 6. The ACME client will repeat such a POST request until the status of the order is "valid" (or "invalid" in the case of an error). This can increase network traffic when considering multiple clients at the same time. Moreover, Although the most common option, the HTTP-01 challenge is not the best option for issuing multiple certificates for multiple servers and if firewalls are blocking HTTP port (80).

B POST Request Example

Figure 7 shows an example of a POST request to `/pq-order` (in our proposed ACME challenge). We followed the notation of the `/new-order` endpoint [1]. The main differences are: we removed the order's validity period, when focusing on the PQC transition, due to the uncertainty of when quantum computers will arrive; and we included an (optional) `certhash` field in the protected header. Both protected and payload fields have integrity guarantees (e.g., by signing). In this example, `Dilithium2` is the PQC algorithm used for signing. The CSR included in the payload in this case use a post-quantum signature algorithm. However, we note that one could also use classical algorithms in the POST and CSR, aiming at renewing classical certificates.

The POST message uses the account's private key to sign the protected and payload JSON fields. This complies to JSON Web Signature (JWS) [9] requirements. After validating the POST message, the server issues the certificate

```
POST /acme/pq-order HTTP/1.1
 Host: example.com
 Content-Type: application/jose+json
 {
  "protected": base64url({
   "alg": "Dilithium2",
   "kid": "https://example.com/acme/acct/evOfKhNU60wg",
   "nonce": "5XJ1L3lEkMG7tR6pA00clA",
   "url": "https://example.com/acme/pq-order",
   "certhash": "89f308210c7c7820b...947c3188dedba6e3"
  }),
  "payload": base64url({
   "csr": "MIIBPTCBxAIBADBFMQ...KdZeGsysoCo4H9P",
    "identifiers": [
     { "type": "dns", "value": "www.teste.org" },
     { "type": "dns", "value": "teste.org" }
    ],
  }),
  "signature": "H6ZXtGjTZyUnPeKn...wEA4TklBdh3e454g"
 }
```

Fig. 7. POST request example

and returns to the client the URL for the certificate's location (similarly as in standard ACME). In this way, the ACME client can ask for a classical or PQC certificate with our proposed challenge in a single request.

References

1. Barnes, R., Hoffman-Andrews, J., McCarney, D., Kasten, J.: Automatic certificate management environment (acme). RFC 8555, RFC Editor (March 2019)
2. Bernstein, D.J., Lange, T.: Post-quantum cryptography. Nature **549**(7671), 188–194 (2017)
3. Bindel, N., Braun, J., Gladiator, L., Stöckert, T., Wirth, J.: X. 509-compliant hybrid certificates for the post-quantum transition. Journal of Open Source Software **4**(40), 1606 (2019)
4. Bindel, N., Herath, U., McKague, M., Stebila, D.: Transitioning to a quantum-resistant public key infrastructure. In: Lange, T., Takagi, T. (eds.) Post-Quantum Cryptography. pp. 384–405. Springer International Publishing, Cham (2017)
5. Encrypt, L.: Challenge types (2020), https://letsencrypt.org/docs/challenge-types/
6. Encrypt, L.: Let's encrypt stats (2022), https://letsencrypt.org/pt-br/stats/
7. Forum, L.E.C.: The lifecycle of a valid authorization (2019), https://community.letsencrypt.org/t/the-lifecycle-of-a-valid-authorization/101387/2
8. Foundation, E.F.: Certbot - get your site on https. (2022), https://certbot.eff.org/
9. Jones, M., Bradley, J., Sakimura, N.: Json web signature (jws). RFC 7515, RFC Editor (May 2015), http://www.rfc-editor.org/rfc/rfc7515.txt, http://www.rfc-editor.org/rfc/rfc7515.txt
10. Joseph, D., Misoczki, R., Manzano, M., Tricot, J., Pinuaga, F.D., Lacombe, O., Leichenauer, S., Hidary, J., Venables, P., Hansen, R.: Transitioning organizations to post-quantum cryptography. Nature **605**(7909), 237–243 (May 2022). https://doi.org/10.1038/s41586-022-04623-2, https://doi.org/10.1038/s41586-022-04623-2
11. Kampanakis, P., Panburana, P., Daw, E., Geest, D.V.: The viability of post-quantum x.509 certificates. Cryptology ePrint Archive, Paper 2018/063 (2018), https://eprint.iacr.org/2018/063, https://eprint.iacr.org/2018/063

12. LLC, G.: Go standard library (2023), https://pkg.go.dev/std
13. Mosca, M., Piani, M.: Quantum threat timeline report 2020. Available at: https://globalriskinstitute.org/publications/quantum-threat-timeline-report-2020/. Accessed on 20.07.2021. (2020)
14. NIST: Post-quantum cryptography (2016), https://csrc.nist.gov/Projects/Post-Quantum-Cryptography. Acessed: 2021-06-26
15. NIST: HMAC - Glossary. https://csrc.nist.gov/glossary/term/hmac. Acessed: 2022-11-01 (2022)
16. NIST: Pqc standardization process: Announcing four candidates to be standardized, plus fourth round candidates (2022), https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4. Acessed: 2022-11-02
17. Nystrom, M., Kaliski, B.: Pkcs #10: Certification request syntax specification version 1.7. RFC 2986, RFC Editor (November 2000)
18. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in tls. In: Ding, J., Tillich, J.P. (eds.) Post-Quantum Cryptography. pp. 72–91. Springer International Publishing, Cham (2020)
19. Project, O.Q.S.: liboqs-go: Go bindings for liboqs. Available at: https://github.com/open-quantum-safe/liboqs-go (2022), [Online; accessed 25-Jan-2022]
20. Raavi, M., Chandramouli, P., Wuthier, S., Zhou, X., Chang, S.Y.: Performance characterization of post-quantum digital certificates. In: 2021 International Conference on Computer Communications and Networks (ICCCN). pp. 1–9. IEEE, Athens, Greece (2021). https://doi.org/10.1109/ICCCN52240.2021.9522179
21. Rescorla, E.: The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor (August 2018)
22. Rudra, A.: How long does it take for dns to update? dns propagation (2022), https://powerdmarc.com/how-long-does-it-take-for-dns-to-update/
23. Santesson, S., Tschofenig, H.: Transport layer security (tls) cached information extension. RFC 7924, RFC Editor (July 2016)
24. Schwabe, P., Stebila, D., Wiggers, T.: Post-Quantum TLS Without Handshake Signatures, p. 1461–1480. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3372297.3423350
25. Schwabe, P., Stebila, D., Wiggers, T.: More efficient post-quantum kemtls with predistributed public keys. In: Bertino, E., Shulman, H., Waidner, M. (eds.) Computer Security – ESORICS 2021. pp. 3–22. Springer International Publishing, Cham (2021)
26. Shoemaker, R.: Automated certificate management environment (acme) tls application-layer protocol negotiation (alpn) challenge extension. RFC 8737, RFC Editor (February 2020)
27. Shor, P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: Proceedings 35th annual symposium on foundations of computer science. pp. 124–134. IEEE, IEEE, Santa Fe, NM, USA (1994)
28. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Assessing the overhead of post-quantum cryptography in tls 1.3 and ssh. In: Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies. pp. 149–156. Association for Computing Machinery, New York, NY, USA (2020)
29. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptography – SAC 2016. pp. 14–37. Springer International Publishing, Cham (2017)
30. ZeroSSL: Free ssl certificates and ssl tools (2022), https://zerossl.com/