Extensible Post Quantum Cryptography Based Authentication

Homer A. Riva-Cambrin, Rahul Singh, Sanju Lama, and Garnette R. Sutherland

Project neuroArm, Dept. Of Clinical Sciences,

Cumming School of Medicine, University of Calgary

(Dated: June 10, 2025)

Abstract

Cryptography underpins the security of modern digital infrastructure, from cloud services to health data. However, many widely deployed systems will become vulnerable after the advent of scalable quantum computing. Although quantum-safe cryptographic primitives have been developed, such as lattice-based digital signature algorithms (DSAs) and key encapsulation mechanisms (KEMs), their unique structural and performance characteristics make them unsuitable for existing protocols. In this work, we introduce a quantum-safe single-shot protocol for machine-to-machine authentication and authorization that is specifically designed to leverage the strengths of lattice-based DSAs and KEMs. Operating entirely over insecure channels, this protocol enables the forward-secure establishment of tokens in constrained environments. By demonstrating how new quantum-safe cryptographic primitives can be incorporated into secure systems, this study lays the groundwork for scalable, resilient, and future-proof identity infrastructures in a quantum-enabled world.

I. INTRODUCTION

When clients access services that contain restricted information, authorization is essential to limit access to intended users. The security of these measures is of special importance for privacy-focused disciplines such as medicine, where patient or health information must be closely guarded [1]. Traditionally, this was achieved by sending client credentials (i.e., a username and password) with each request. However, this required the service to store passwords and inherited the well-known weaknesses of password-based authorization like frequent user error [2].

A more modern approach is token-based authorization such as OAuth 2.0 [2, 3]. In OAuth 2.0, a client authenticates with an authorization server to receive an access token, which is used to authorize subsequent requests. Since OAuth 2.0 is not suited for authentication [2], protocols like OpenID Connect (OIDC) [4] extend OAuth 2.0 to provide authentication. The OIDC protocol uses a specific type of token called a JSON Web Token (JWT) [5], which rely on digital signatures [6]. In a digital signature scheme, a signer holds a private key and makes their public key available to all. Messages signed by the private key can be verified by the corresponding public key, confirming that the private key holder did indeed sign the message. In addition, these signatures also enable integrity verification: if the message is modified, the signature is no longer valid. However, since these tokens carry the signature

with each request, they tend to be large, increasing network bandwidth usage.

The emergence of quantum computing presents a new challenge to current cryptographic protocols [7]. Although it opens the door to new paradigms of quantum-based authentication [8, 9], it also threatens conventional asymmetric cryptography methods [7, 10] such as elliptic curve cryptography [11] and RSA, which can be broken with Shor's algorithm [12]. These algorithms underpin much of modern secure communication, such as web traffic and financial systems.

To prepare for this shift, the National Institute of Standards and Technology (NIST) initiated a standardization effort for quantum-safe digital signatures [13]. On August 13, 2024, NIST standardized two digital signature algorithms, ML-DSA [14] and SLH-DSA [15] that are believed to be secure against large-scale quantum computers.

One of the main challenges in adopting these new signatures is their considerable size [16]. ML-DSA produces 2,420-byte signatures and SLH-DSA has 7,860-byte signatures, which are orders of magnitude larger than 64-byte Elliptic Curve Digital Signing Algorithm (ECDSA) signatures used today. Embedding such large signatures in tokens would increase transmission costs and latency, and could also make systems susceptible to denial-of-service attacks [10, 16].

Clearly, new authentication and authorization protocols must be designed with these constraints in mind. One such example is KEMTLS [17] which replaces the TLS handshake with quantum-safe primitives. Inspired by these developments, and the need for health data security in the operating room (IoT-based surgical systems), we here propose a novel protocol that solves a specific goal: efficient and secure machine-to-machine authorization and authentication in the presence of quantum-capable adversaries. Our protocol uses quantum-safe key encapsulation mechanisms (KEMs) and digital signature algorithms (DSAs) to establish tokens over insecure channels, eliminating the need to include large digital signatures in the tokens themselves. It also supports key rotation. To verify its security properties, we formally analyze the protocol using TAMARIN [18], a symbolic verification tool for cryptographic protocols.

II. CRYPTOGRAPHY FOR HEALTH DATA

Suppose there is a medical device that transmits data to a cloud server. We want only that authorized machine communicating with the cloud. A stronger guarantee is the certainty that the messages sent over the network really *do* originate from the machine, and have not been modified or forged by an attacker.

A. Integrity

Suppose the machine has a message m to send to the server which comprises of medical data. This medical data is essential to the treatment of the patient, and thus we must validate that m is received by the server exactly as it was sent by the machine, and not modified maliciously by an attacker.

One cryptographic primitive that could be applied here is a hash function. A hash function h maps strings of arbitrary length x to strings of fixed length $h(x)$. A desirable property of hash functions is *pre-image resistance*. That is, if we have a hash value $h(x)$, it is computationally infeasible to find x. We also want *collision resistance*, that is, if we have two unique messages x_1, x_2 such that $x_1 \neq x_2$ then $h(x_1) \neq h(x_2)$. [7]

To detect modifications to the message m, the machine sends a hash along with the message, making the final message $(m, h(m))$. When the server receives the message, the server computes $h(m)'$ and checks $h(m)' = h(m)$. Since the hash function has collision resistance, if even a single bit was changed, the hash is different and it becomes evident that the message has been modified. However, the obvious attack here is that the message can be modified, the hash recomputed, and in doing so the server is tricked into believing no modification took place.

A digital signature algorithm provides better guarantees. The machine generates a key pair consisting of a private key k_{priv} and a public key k_{pub}. The public key is transmitted publicly so that the attacker and the server can see it. Now, the machine sends the message $(m, \{m\}_{k_{\text{priv}}})$, which is the message m followed by the signature of m under k_{priv}.

The server uses the public key to trivially verify that m was *indeed* signed by the machine. If m had been modified by an attacker, the signature would no longer be valid, and if the attacker were to modify the signature, then it would no longer be verifiable using the public

FIG. 1. **A key encapsulation mechanism (KEM).** The client starts with the decapsulation key `DK` and the encapsulation key `EK`, sends the `EK` to the server, which the server uses to encapsulate the shared secret `SEC` which can be decpasulated by the client.

key. Thus, we have shown that the message was sent by the machine *and* that it has not been modified.

B. Shared Secrets

Suppose the machine needs to establish a secret string in order to encrypt messages. Considering the initial channel is insecure, a shared secret will need to be established without the attacker learning said secret. For this, a key encapsulation mechanism (KEM) is employed. (Fig. 1)

The machine begins by generating a KEM private key, called the decapsulation key dk, and then a public key, called the encapsulation key pk which is shared with the server. The server then encapsulates the shared secret S using the encapsulation key to produce the cipher text. The machine receives the cipher text, and uses the decapsulation key to retrieve the secret S. All of this occurs without the attacker obtaining knowledge of S, and thus a shared secret is successfully established between the two parties.

III. PROTOCOL

A. Overview and Preconditions

Here we present a brief overview of the protocol (Fig. 2) using the same medical device and server from the previous section. Our goal is to use quantum-safe digital signatures and KEMs to establish a token to authorize future requests. Before authentication can proceed, the medical device must be registered with the server by an administrator.

The administrator generates an ID for the medical device, along with a private and public

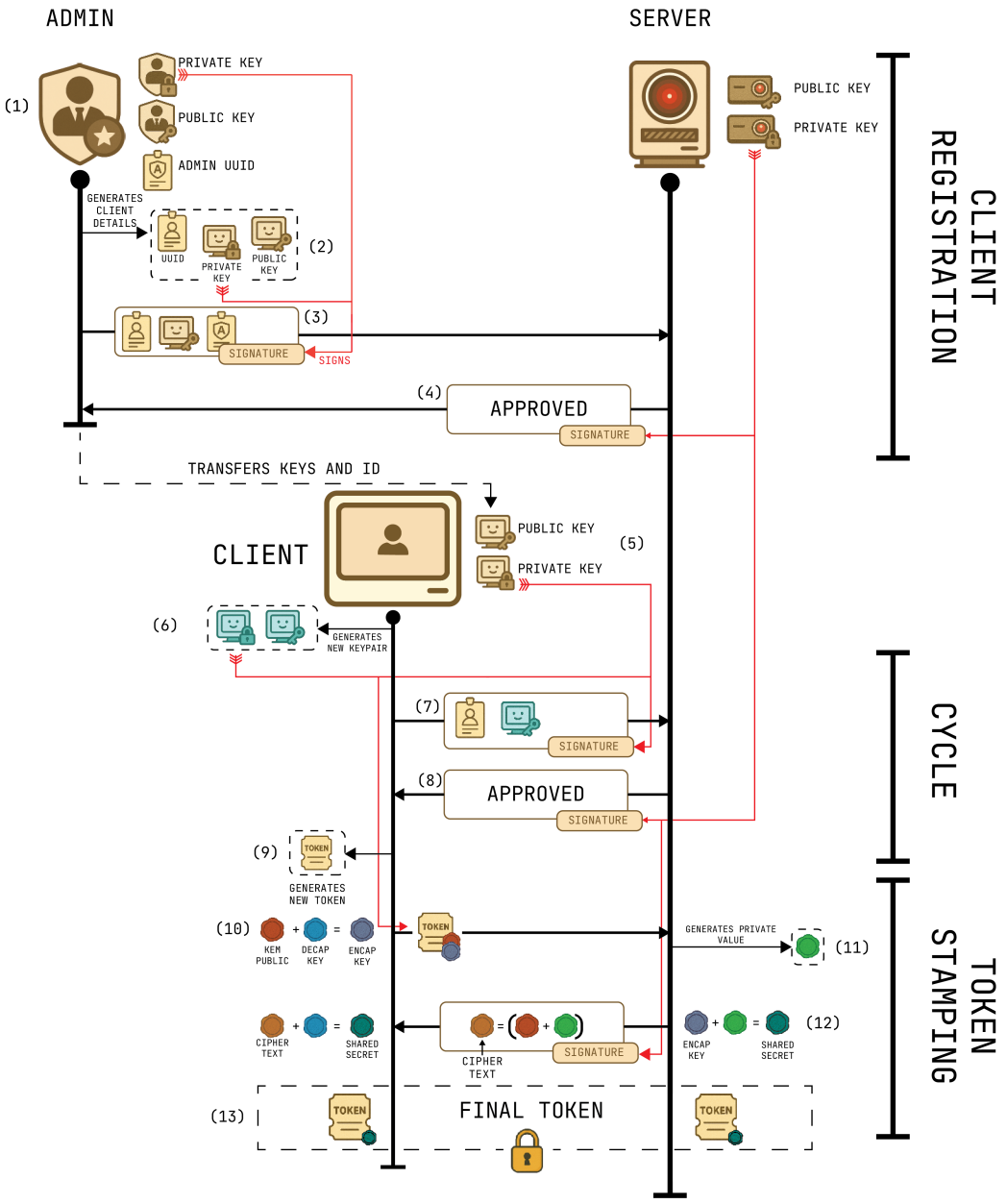

FIG. 2. **A simplified overview of the protocol.** The protocol begins with the administrator (1) generating client credentials (3) and sending them to the server for approval (3). The server approves the credentials (4), and the administrator installs them securely on the client machine (5). To initiate a key cycle, the client generates a new key pair (6) signed with both old and new keys, and sends it to the server (7). The server verifies and approves the key cycle (8). For token stamping, the client generates a preview token (9), attaches a KEM public key and encapsulation key, and transmits this to the server (10). The server uses a private value (11) to generate a ciphertext which is sent to the server (12). Both parties derive the shared secret and compute the final token (13). Cycles and token stamping may take place an arbitrary number of times and in any order.

key. The administrator forms a message containing the ID and the public key and signs it with both the client key and the administrator key to show that the private key does exist, and that the administrator did authorize this request.

The server will examine and validate this request, then form an approval message and sign it with the server private key. This is received and validated by the administrator. The administrator securely installs the client UUID and private key on the medical device, who will direct all interaction with the server from there on. As the private key serves as the proof-of-identity for the medical device, it is paramount that it is transported securely [19].

The medical device may choose to change the key arbitrarily, or when required by the server. This process is called key cycling. The medical device generates a new key pair, and forms a message comprising of the device ID and new public key. This message is signed twice. The first signature is by the new private key, to prove that this key is owned and exists, and the second signature is by the old private key, to authorize this request. This message, along with the signatures, is sent to the server, who trivially validates it, installs the new key, and sends an approval message signed with the server private key.

We will now obtain a token. A preview token is proposed by the medical device which states certain properties that the device would like the token to have. This could be permission scopes or a certain version of the protocol. The token also contains a timestamp and a payload which is a random string of 32 bytes. We also compute a KEM encapsulation & decapsulation key. The token and encapsulation key are signed and then sent to the server.

The server validates the signature and the token details, that the timestamp is valid, and that the permission scopes requested are reasonable. The server then uses the encapsulation key to generate a shared secret $\mathfrak{S}$. The token payload is replaced by $\mathfrak{S}$, and then the token is hashed and stored in the server database. The server sends the KEM ciphertext back along with the hash of the final token, signed by the server private key.

The client uses the KEM ciphertext & decapsulation key to derive $\mathfrak{S}$, derives the same token, and verifies it by checking the hash. Thus, a secret token is successfully established over an insecure channel.

B. Protocol Time

To prevent replay attacks, the protocol uses a special counter-based timestamp known as the *protocol time*. This time stamp system is simple and avoids the need for synchronized clocks or external time sources. The principal goal is to ensure that no two tokens signed under the same key can ever be identical.

Each client maintains its own protocol time with the server, which begins at zero upon successful registration. Every token request, regardless of error or success, increments this value by one. If the client initiates a key cycle, the protocol time is reset to zero. If the client sends an incorrect time, then the server should report back the correct protocol time.

To prevent timestamp reuse via counter wrapping, the protocol enforces cycling once the maximum value of an unsigned 64- bit integer is reached. This forces the counter back to zero, which is sound since the key has changed and thus old tokens may not be replayed. This design ensures strict monotonicity and protects against replay-based token reuse.

C. Tokens

FIG. 3. **The layout of the token byte structure in big endian.** Each rectangle specifies the contents of the field as well as the amount of bytes the field takes up. Bytes are ordered from top-down, i.e., protocol comes before device type.

Tokens have a defined and well-known structure and thus are called *transparent* (Fig 3). The protocol and device field allow specifying the signing algorithms to use. The UUID field contains the client ID who signed this message. The 16-byte permission field is for extensibility and is not specified. It could be used for permission scopes similar to claims on a JWT[5]. The timestamp is the protocol time encoded as a 64-bit unsigned integer. The payload is a 32-byte string. In a preview token, this will be random. In a finalized token, this is the shared secret established by the KEM.

D. Procedure

The following sections will describe the protocol (Fig. 2) in detail. The network is assumed to be insecure, so an attacker can read and modify any message. However, the protocol could be layered over a secure channel such as TLS. Each section will also briefly discuss the security goals of the various parts of the protocol [20].

1. Special Notation

For the purposes of simplicity and concise presentation, some special notation is used for signatures and key pairs. For signatures we write $\{m\}_k$ which is the signature of m under k. For private key k, the public key is written $\overline{k}$. In a tuple, we may write $(m, \{\leftarrow\}_k)$ which is the message m and then the signature of m under k. If we write $(m, \{\leftarrow\}_k, \{\leftarrow\}_{k_2})$ then that is m, the signature of m under k, and the signature of the previous signature under k_2.

2. Registration

The goal of registration is that only authorized administrators may register new clients to the server, and that clients who are being registered actually possess the key pair they claim to possess. The exact semantics of administrators is out of scope for this protocol. Registration is initiated by an administrator who possesses the administrator private key k_α and administrator ID I_α. The administrator proposes a new client with ID I_c and public key $\overline{k_c}$. The administrator produces the message $M_1 = \langle \texttt{REGISTER}, I, \overline{k_c}, I_\alpha \rangle$. The administrator uses the client private key and the administrator private key to produce the final message $(M_1, \{\leftarrow\}_{k_c}, \{\leftarrow\}_{k_\alpha})$, which is sent to the server.

Let the server key pair be $(k_S, \overline{k_S})$. Upon receiving this message, the server verifies the validity of both signatures and that the public key and identifier are unique. The server responds $(\langle \texttt{REGSUCCESS}, h(I) \rangle, \{\leftarrow\}_{k_S})$ to the administrator. The administrator verifies the message was signed by the server, and verifies the identifier hash. This prevents the impersonation of the server, and ensures the registration response may never be replayed since identifiers must be unique.

Through secure means, the administrator installs the client identifier I and private key k_c on the client. All subsequent communications take place between the client and the server.

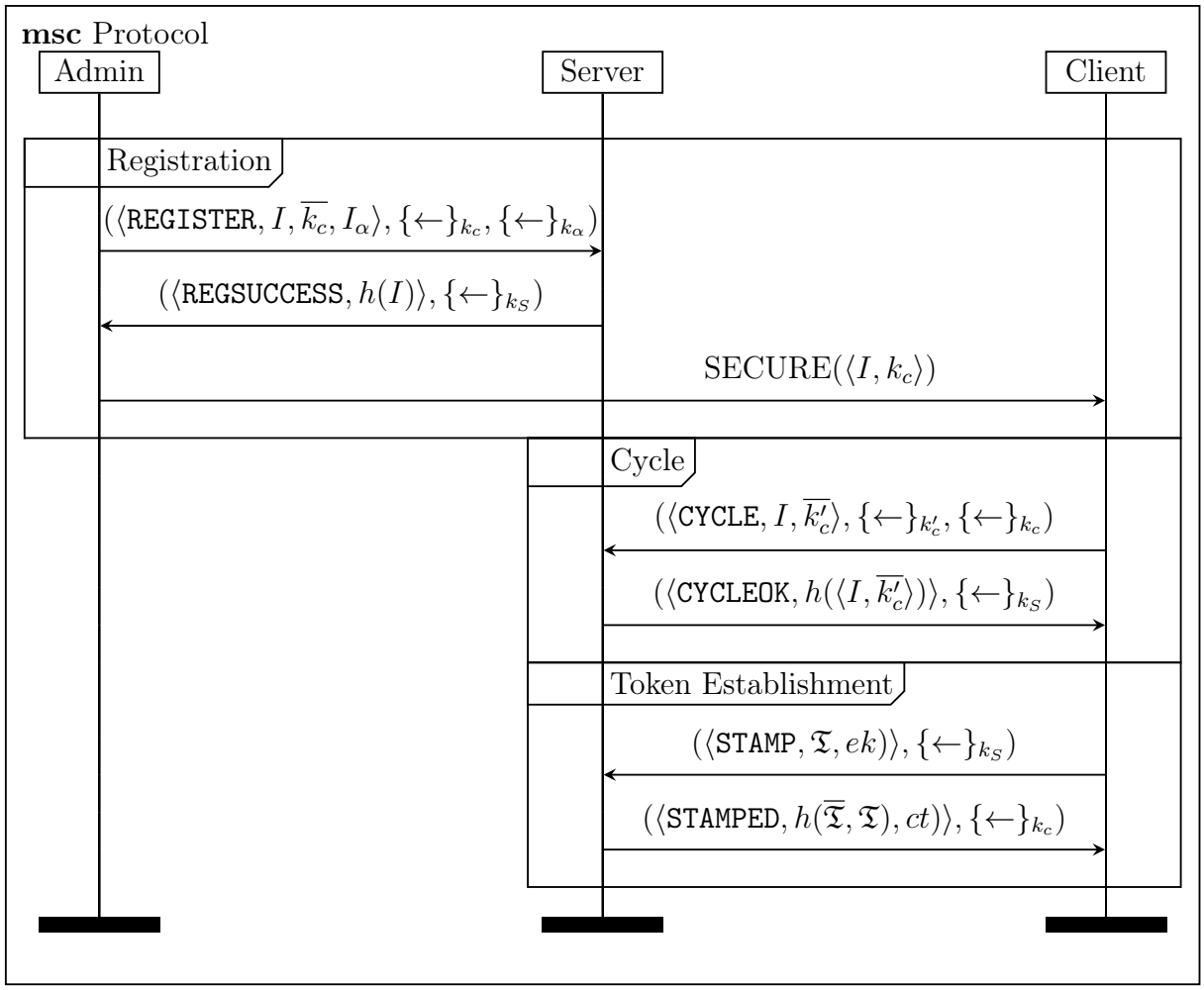

FIG. 4. **Protocol.** The exact protocol specification showing the messages exchanged between the administrator and server and then subsequently between the client and server.

3. Key Cycling

Key cycling is only necessary when the key expires, as determined by server policy. If a client attempts to get a token with an expired key, the server will direct the client to initiate a cycle. Alternatively, a client may choose to initiate a cycle arbitrarily often.

The goal of key cycling is that only the owner of the private key can update the public key on the server, and that the public key proposed was created by the client and forms part of a legitimate key pair.

To begin the cycling process, the client generates a new key pair $(k'_c, \overline{k'_c})$. The client then sends $(\langle \texttt{CYCLE}, I, \overline{k'_c} \rangle, \{\leftarrow\}_{k'_c}, \{\leftarrow\}_{k_c})$ to the server. Here we note that the cycling process is practically identical to the registration process, with the difference being that the client

self-authorizes, removing the need to include the administrator ID.

The server verifies the signatures, verifies the public key is unique, and computes the verification hash $\mathfrak{v} = h(\langle I, \overline{k'_c} \rangle)$. The server then responds with $(\langle \texttt{CYCLEOK}, \mathfrak{v} \rangle, \{\leftarrow\}_{k_S})$. The client verifies the authenticity of the response with the server public key, and verifies the verification hash.

4. Token Stamping

The final operation supported is token establishment, or *stamping*, the central operation of the protocol. Tokens are stamped in such a way to prevent replay attacks. Additionally, since a token verifies against a client, a token is valid if and only if the client itself is valid. Each token stamped corresponds uniquely to a single non-repeating instance of the protocol, fulfilling the property of injective agreement. Tokens are forward-secure due to their reliance on quantum-safe KEMs which are independent from the signing keys. Furthermore, the server stores only hashes of tokens, meaning that an attacker cannot extract valid tokens even in a database breach.

Token establishment process begins with the client generating a preview token. The token has the following structure:

$$\mathfrak{T} = \langle p_\alpha, p_\beta, I, p_\psi, p_\Delta, p_\zeta \rangle \tag{1}$$

where p_α is the protocol, p_β the device, I the client UUID, p_ψ the permission field, p_Δ the timestamp, and p_ζ the payload field. The payload field may be initialized to a random string of bytes.

The client then generates a KEM encapsulation key pk and decapsulation key sk. The client sends $(\langle \texttt{STAMP}, \mathfrak{T}, pk \rangle, \{\leftarrow\}_{k_c})$ to the server.

The server receives this, verifies the signature, and verifies that the token's timestamp aligns with the server protocol time for that client. Additionally, the server checks and sees that this token is not yet in the database.

The server may also verify the permission scopes, however this is out of scope for the protocol and is left up to the implementer.

The server computes the KEM shared secret $\mathfrak{S}$ and produces the ciphertext ct. The server computes the final token $\overline{\mathfrak{T}_S}$ as,

$$\overline{\mathfrak{T}}_S = \langle p_\alpha, p_\beta, I, p_\psi, p_\Delta, \mathfrak{S} \rangle \tag{2}$$

The server then computes the approval hash $\mathfrak{a}_h = h(\langle\overline{\mathfrak{T}_S}, \mathfrak{T}\rangle)$ and sends $(\langle \texttt{STAMPED}, \mathfrak{a}_h, ct\rangle, \{\leftarrow\}_{k_S})$ to the client.

The client will verify the signature using the server public key. The client then uses the ciphertext ct and the decapsulation key dk to derive the shared secret $\mathfrak{S}$. The client computes the client token:

$$\overline{\mathfrak{T}}_C = \langle p_\alpha, p_\beta, I, p_\psi, p_\Delta, \mathfrak{S}\rangle \tag{3}$$

The client now verifies that $h(\langle\overline{\mathfrak{T}_C}, \mathfrak{T}\rangle) = \mathfrak{a}_h = h(\langle\overline{\mathfrak{T}_S}, \mathfrak{T}\rangle)$, asserting that the token was correctly established and that $\overline{\mathfrak{T}_S} = \overline{\mathfrak{T}_C}$.

IV. ANALYSIS

We formally analyzed the protocol using the Tamarin prover [18], a state-of-the-art tool for symbolic verification of cryptographic protocols. Tamarin supports rigorous security proofs under the Dolev-Yao adversary model [21] in which the adversary has complete control over the network. The adversary can delete messages, modify them, and has full knowledge of everything sent across the network.

Tamarin includes built-in support for standard cryptographic primitives, such as hashing functions and digital signatures. For instance, hashing is modeled as `h/1`, which is a unary function with no equations. That is `h(x1)=h(x2)` if and only if `x1=x2`. Digital signatures are modeled by the following equations:

$$\texttt{verify}(\texttt{sign}(m, sk), m, \texttt{pk}(sk)) = \texttt{true} \tag{4}$$

which states that a valid signature under a private key sk is only verifiable using its corresponding public key.

Since key encapsulation mechanisms are not natively supported, we incorporate a custom KEM model based on the Tamarin analysis of KEMTLS [17]. We introduce three new functions: `kempk/2`, `kemencaps/3`, and `kemdecaps/3` governed by the following equation:

$$\texttt{kemdecaps}(g, \texttt{kemencaps}(g, ss, \texttt{kempk}(g, sk)), sk) = ss \tag{5}$$

where g represents shared parameters. This captures the core property of KEMs: only the two parties can derive the shared secret. To support protocol-level guarantees such as identity uniqueness and signature verification, we specified formal restrictions, which prune invalid

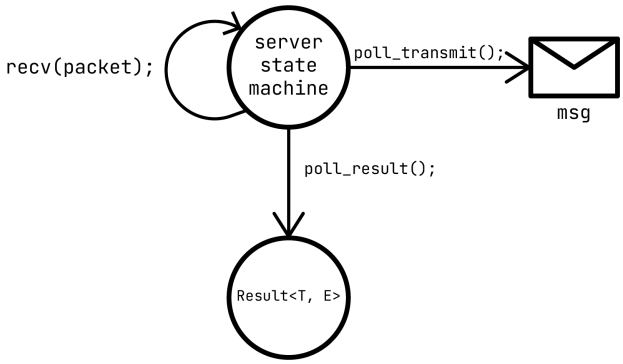

FIG. 5. **Server state machine in the `quath` implementation.** The `recv` function processes an input and advances the internal state. The `poll_transmit` emits a message if external work is required (i.e., I/O or database interaction), and `poll_result` returns either a result or an error.

execution traces from the search space. In our case, these are traces where the signature does not verify, or non-unique identifiers are being used.

For the server keys, a minimal public key infrastructure (PKI) was define, allowing us to model the distribution and compromise of the long-term server key pair. Adversarial capabilities were explicitly modeled via rules such as `Reveal_ltk`, `Reveal_client`, and `Reveal_token`, which simulate the leakage of private keys and tokens. Additionally, token expiration was modeled through dedicated rules, enabling the verification of time-bounded access control policies.

Security properties are formalized as lemmas and verified through automated interactive proofs. To demonstrate the protocol's executability in the absence of an attacker, we included several `exists-trace` lemmas confirming that honest protocol executions are possible.

A current limitation of the model is that each client is restricted to a single key cycle, and tokens can only be verified once per run. While sufficient to validate core security properties, future work should model an unbounded amount of cycles and multiple rounds of token verification to provide full analysis.

The complete Tamarin model and proof scripts are available at:
https://github.com/OrbSurgicalAI/quath.

V. IMPLEMENTATION

A core design goal of the protocol was to ensure ease of implementation across a variety of environments, including both synchronous and asynchronous runtimes. To achieve this, we adopted a SANS I/O design pattern [22] where the protocol logic operates purely on bytes and never performs I/O directly. This decouples the protocol from the execution context, allowing it to be embedded in diverse systems ranging from web servers to embedded devices.

Each component of the protocol (registration, key cycling, and token stamping) is modeled as an explicit state machine (Fig. 5). The server state machines are advanced with the `recv` function, which consumes an input and updates internal state. When external work is required such as I/O or database interaction, the state machine emits a request via `poll_transmit`. The driver then invokes `poll_result` which can either be `Pending` if the state machine is awaiting further work, or `Ready` if the state machine has terminated with a result. Example driver code is included in Fig. 6.

To ensure high performance and type-safety, two qualities which we believe to be critical in safe protocol design, the implementation was written in the Rust programming language. Rust offers strong compile-time guarantees that eliminate entire classes of runtime errors and provides robust memory safety without requiring a garbage collector. These properties make it well-suited for implementing cryptographic protocols where correctness and safety are paramount.

To support flexibility across cryptographic primitives, the implementation leverages Rust's generic trait system to abstract over DSAs, KEMs, and hash functions. This design allows the protocol logic to remain agnostic to specific cryptographic choices, enabling support for a variety of post-quantum algorithms, including ML-DSA [14], SLH-DSA [15] and ML-KEM [23].

This modular and generic architecture makes the protocol highly portable and easy to deploy. For instance, an organization such as a bank can readily adopt the system by integrating the state machines into their existing infrastructure. They would simply need to connect the storage and networking handlers, and the protocol will operate seamlessly within the environment.

```rust
let mut machine = StateMachine::new(); // Instantiate a State Machine.
let mut input_queue = VecDeque::new(); // Service queue.
loop {
    if let Some(item) = input_queue.pop_back() {
        machine.recv(item); // If we have a new input.
    }
    if let Some(to_handle) = machine.poll_transmit() {
        input_queue.push_front(handle_fn(to_handle)); // Service the request.
    }
    if let Poll::Ready(result) = machine.poll_result() {
        return result; // The result is ready.
    }
}
```

FIG. 6. **An example of driving the state machine forward in Rust.** In this example the state machine is driven forward by some rust code.

VI. PARAMETERS

TABLE I. Recommended protocol parameters based on NIST security levels. Selections follow NIST guidelines.

Level	DSA	KEM	Hash Function
1	ML-DSA-44	ML-KEM-512	SHA3-256
3	ML-DSA-65	ML-KEM-768	SHA3-384
5	ML-DSA-87	ML-KEM-1024	SHA3-512

The parameters recommended based on the desired security category are shown in Table I. The level is the security category based on the weakest link in the parameter set. Digital signature algorithms specified in the table are described in FIPS204 [14], the key encapsulation mechanisms are described in FIPS203 [23], and the hash functions are described in FIPS202 [24].

VII. DISCUSSION

This work presents a practical and formally verified approach for integrating state-of-the-art post-quantum cryptographic primitives into a robust authentication and authorization protocol. The relevance of such systems is increasingly urgent as large-scale quantum computers threaten the long-term security of currently deployed cryptosystems. Critical data, including financial records, medical information, and state secrets, may be vulnerable to attack if post-quantum security is not adopted.

A key feature of our protocol is the compact and extensible token format. Each token occupies only 74 bytes, a stark contrast to the often kilobyte-scale tokens used in comparable systems. This efficiency yields substantial bandwidth savings in high-throughput environments and is particularly advantageous for resource-constrained deployments such as embedded systems.

Beyond efficiency, the protocol offers strong flexibility and extensibility. Tokens can be customized to include permission scopes and application-specific metadata, making the protocol well-suited for secure machine-to-machine interactions in any environment. The protocol need not be limited to machines alone; as enterprise security evolves, this protocol could readily extend to secure user authentication scenarios as well.

A central motivation behind this work is trust, which is a foundational requirement in cryptographic systems [25]. Trust is especially important when we are dealing with sensitive data such as healthcare data or financial records [1]. However, proving the security of protocols remains a profound challenge, which in the past was conjectured to be an insurmountable task [26, 27] due to the inherent complexities in modeling real-world adversarial behaviors.

Nonetheless, formal verification remains one of the most effective tools for providing provable security guarantees for protocols [20, 26].

The need for formal analysis is underscored by the nature of the data guarded by the protocols we use daily. For instance, TLS secures daily web traffic. The triple handshake vulnerability in TLS was uncovered and healed through formal analysis [27, 28]. Historically, protocols like Otway-Rees [29], which appear to be secure on first glance, have been shown to have major flaws with trivial formal analysis. These finings highlight the inadequacy of intuition-driven protocol design and verification, and the necessity of machine-checked verification.

To that end, we employed the Tamarin Prover [18] to formally verify our protocol's key phases: registration, key cycling, and token issuance. Tamarin's symbolic model, employing the Dolev-Yao adversary [21], allows reasoning over all possible executions under a network controlled by a powerful adversary. Indeed, Tamarin has been used for proofs of prolific protocols such as TLS1.3 [27] and Apple iMessage [30].

We prove properties such as injective agreement, session key ownership, and resilience to token expiration attacks under a limited version of the model. Importantly, the model also models scenarios such as database leakage and full-server compromise. These formally proven properties provide robust assurance of the protocol's correctness and resilience, even under a powerful adversary.

A. Limitations

One limitation of the protocol is that the client must have prior knowledge of the server's public key. This requirement can be mitigated by executing the protocol over an authenticated secure channel, such as one established through KEMTLS, thus eliminating the necessity to pre-distribute the server's public key. Additionally, features such as token revocation could be explicitly modeled in future versions of the protocol.

Another limitation is the scope of the security analysis presented. While important properties such as key cycling and token verification have been demonstrated for a single protocol cycle, a more comprehensive analysis over an arbitrary number of key cycles and multiple rounds of token verification would provide significantly stronger assurances.

Future work should focus on formulating rigorous inductive proofs to thoroughly verify these properties over repeated interactions.

B. Additional Information

Contact Information: Homer A. Riva-Cambrin, homer.rivacambrin@ucalgary.ca; Sanju Lama, slama@ucalgary.ca; Garnette R. Sutherland, garnette@ucalgary.ca; Rahul Singh, rahul.singh3@ucalgary.ca;

Competing Interests: The security protocol as presented in the work, and its widespread application, stems from the need for health data protection relative to our innovations in

the Operating Room technologies. All authors have affiliation to a University of Calgary (Project neuroArm Medical Robotics research facility) spin-off called OrbSurgical Ltd.

Funding: Canadian Institutes for Health Research Commercialization Grant Application #390405 .

Data Sharing: All code and data can be located at the following link: https://github.com/OrbSurgicalAI/quath

---

[1] A. S. Nadhan and I. Jeena Jacob, Enhancing healthcare security in the digital era: Safeguarding medical images with lightweight cryptographic techniques in iot healthcare applications, Biomedical Signal Processing and Control **88**, 105511 (2024).

[2] D. Hardt, The OAuth 2.0 Authorization Framework, RFC 6749 (2012).

[3] D. Hardt, A. Parecki, and T. Lodderstedt, *The OAuth 2.1 Authorization Framework*, Internet-Draft draft-ietf-oauth-v2-1-11 (Internet Engineering Task Force, 2024) work in Progress.

[4] N. Sakimura, J. Bradley, M. B. Jones, B. de Medeiros, and C. Mortimore, OpenID Connect Core 1.0 (2014).

[5] M. B. Jones, J. Bradley, and N. Sakimura, JSON Web Token (JWT), RFC 7519 (2015).

[6] P. Saha, A comprehensive study on digital signature for internet security, Accent. Trans. Inf. Secur **1**, 1 (2016).

[7] D. J. Bernstein and T. Lange, Post-quantum cryptography, Nature **549**, 188 (2017).

[8] N.-H. Lim, J.-W. Choi, M.-S. Kang, H.-J. Yang, and S.-W. Han, Quantum authentication method based on key-controlled maximally mixed quantum state encryption, EPJ Quantum Technology **10**, 35 (2023).

[9] G. Chen, Y. Wang, L. Jian, Y. Zhou, and S. Liu, Quantum identity authentication based on the extension of quantum rotation, EPJ Quantum Technology **10**, 11 (2023).

[10] Y. Baseri, V. Chouhan, and A. Hafid, Navigating quantum security risks in networked environments: A comprehensive study of quantum-safe network protocols, Computers and Security **142**, 103883 (2024).

[11] J. R. Wohlwend, Elliptic curve cryptography: Pre and post quantum (2016), undergraduate Research Paper, Massachusetts Institute of Technology.

[12] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a

quantum computer, SIAM Journal on Computing **26**, 1484–1509 (1997).

[13] N. I. of Standards and Technology, Post-quantum cryptography (2025), accessed: 2025-05-18.

[14] National Institute of Standards and Technology, Module-lattice-based digital signature standard, Federal Information Processing Standards Publication (FIPS) NIST FIPS 204 (2024).

[15] National Institute of Standards and Technology, *Stateless Hash-Based Digital Signature Standard*, Federal Information Processing Standards Publication FIPS 205 (National Institute of Standards and Technology, Washington, D.C., 2024).

[16] D. Adrian, Post-quantum cryptography is too damn big (2024), accessed: 2025-05-18.

[17] P. Schwabe, D. Stebila, and T. Wiggers, Post-quantum TLS without handshake signatures, Cryptology ePrint Archive, Paper 2020/534 (2020).

[18] D. Basin, C. Cremers, J. Dreier, and R. Sasse, Tamarin: Verification of large-scale, real-world, cryptographic protocols, IEEE Security & Privacy **20**, 24 (2022).

[19] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, *OAuth 2.0 Security Best Current Practice*, Internet-Draft draft-ietf-oauth-security-topics-29 (Internet Engineering Task Force, 2024) work in Progress.

[20] L. Paulson, Proving security protocols correct, in *Proceedings - Symposium on Logic in Computer Science* (1999) pp. 370 – 381.

[21] D. Dolev and A. Yao, On the security of public key protocols, IEEE Transactions on Information Theory **29**, 198 (1983).

[22] C. Benfield, Building protocol libraries the right way, `https://www.youtube.com/watch?v=Evchk7aNKdQ` (2016), presentation at PyCon UK 2016.

[23] National Institute of Standards and Technology, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, Tech. Rep. FIPS 203 (U.S. Department of Commerce, 2024).

[24] M. J. Dworkin, *SHA-3 Standard*, Tech. Rep. FIPS PUB 202 (National Institute of Standards and Technology, 2015).

[25] E. Balsa, H. Nissenbaum, and S. Park, Cryptography, trust and privacy: It's complicated, in *Proceedings of the 2022 Symposium on Computer Science and Law*, CSLAW '22 (Association for Computing Machinery, New York, NY, USA, 2022) p. 167–179.

[26] H. Comon and V. Shmatikov, Is it possible to decide whether a cryptographic protocol is secure or not?, Journal of Telecommunications and Information Technology **4** (2009).

[27] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, A comprehensive symbolic

analysis of tls 1.3, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17 (Association for Computing Machinery, New York, NY, USA, 2017) p. 1773–1788.

[28] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, Triple handshakes and cookie cutters: Breaking and fixing authentication over tls, in *2014 IEEE Symposium on Security and Privacy* (2014) pp. 98–113.

[29] D. Otway and O. Rees, Efficient and timely mutual authentication, SIGOPS Oper. Syst. Rev. **21**, 8–10 (1987).

[30] F. Linker, R. Sasse, and D. Basin, A formal analysis of apple's iMessage PQ3 protocol, Cryptology ePrint Archive, Paper 2024/1395 (2024).