The Citadel Protocol

Thomas P Braun
Avarok Cybersecurity

January 7, 2023

Abstract

The Citadel Protocol is a novel post-quantum peer-to-peer communications protocol that uses some of the latest advancements in cryptography in order to provide developers with a software development kit (SDK) to construct higher-level applications safely and efficiently. Built in the rust programming language using the asynchronous Tokio runtime, the Citadel protocol is cross-platform and mobile compatible with the emerging internet of things (IoT) era in mind. The Citadel protocol, similar to Nginx, can be conditionally compiled to support multi-threaded or single-threaded runtimes for both low and high throughput traffic depending on the needs of the developer. Additionally, sessions can be configured with options such as arbitrarily-deep multi-layered encryption, one of many post-quantum key encapsulation mechanisms, Perfect Forward Secrecy (PFS) and Best Effort Modes (BEM), and either Xchacha20-Poly1305 or AES-256-GCM-SIV for symmetric encryption. Overall, the Citadel protocol offers a highly configurable protocol using the latest advancements and paradigms in software technology.

Contents

I. Introduction

In 1995, Peter Shor discovered vulnerabilities in elliptical curve cryptography due to its inherent use of integer prime factorization. As Shor theoretically proved [7], using sufficiently many qubits, a quantum computer could find the prime factors of the large integers used in elliptical curve cryptography using what we now know as Shor's algorithm in feasible time. This would be disastrous since the networks then (and today) still widely use elliptical curve cryptography.

Imagine an adversary discovering Shor's algorithm in 1995. Like anybody, they could access a public network and store encrypted packets into a long-term database. While in 1995, the adversary would not be able to decrypt the captured information, all the adversary would have to do is wait until a powerful enough quantum computer was available and thereafter *retroactively decrypt* the packets. This implies that irreversible damage is inevitable once quantum computers mature. While nothing can be done about data already captured, the most secure policy would be to secure further transmissions using quantum-secure communication to prevent further damage.

The Citadel Protocol provides a platform for developers to secure communications in a world of emerging quantum computers. In order to achieve this, post-quantum key encapsulation mechanisms (KEMs) are used to securely establish symmetric keys between Alice and Bob. Work by theoretical cryptographers around the world has already yielded an array of possible KEMs. The National Institute of Standards and Technology (NIST) started a post-quantum competition in 2016 [3] to update both public key cryptography and signature schemes for future standardization, allowing these cryptographers to submit, receive critique, and update their algorithms throughout the competition. As of round 3, Classic MeEliece, NTRU, Crystals-Kyber, and Saber KEMs remain in the competition. The finalists are expected to be revealed in 2022, where the head cryptographer Dustin Moody expects to select "several" KEMs [6].

II. Cryptographic Primitives

This section will define the used cryptographic primitives in the Citadel protocol. Thereafter, the next section will explicitly define the operations used in the protocol for clarity.

i. Post-Quantum KEMs

At the start of each session, the Citadel protocol allows the selection of the following KEMs:

KEM Family	Variant	Type
Saber	Lightsaber Saber Firesaber	Learning With Errors
NTRU	Ntruhps2048509 Ntruhps2048677 Ntruhps4096821 Ntruhrss701	Lattice-Based
Crytsals-Kyber	Kyber51290s Kyber76890s Kyber102490s	Learning With Errors

Three of the four round 3 KEMs are selectable in the Citadel protocol. We omit the use of Classic McEliece and its variants due to the disproportionately large key sizes implicated in the transmission process. When the finalists are announced by the NIST in 2022, the list will be updated to reflect the result of the competitions to remain synchronized with NIST standards.

ii. Hashing Algorithms

The Citadel protocol uses key derivation functions to create further keys from the symmetric keys established from post-quantum KEMs. Specifically, the safe hashing algorithm III (SHA3) is used for deriving keys. SHA3 was selected by the NIST in 2015 from the Keccak family of hash functions [4]. Both SHA3-256 and SHA3-512 are used depending on the subroutine.

Passwords are also used in the Citadel Protocol. Argon2 is used over the common pbkdf2 (used in Signal) or scrypt (used in Wickr) password-based hash functions since it is generally considered stronger and recommended by the Internet Engineering Task Force (IETF) [2]. Argon2id is a special version of Argon2 that runs both argon2i (provides resistance to side-channel attacks) and argon2d (provides resistance to GPU attacks) over the input password in order to balance protection against a variety of attack vectors. Citadel uses Argon2id, and utilizes the following parameters:

Parameter	Type
password	bytes
salt	bytes
parallelism	integer
tag length	integer
memory size	integer (KB)
iterations	integer
key	bytes
associated data	bytes

The Citadel protocol has a built-in autotuner that automatically determines the optimal parameters for the argon2id password hashing function given a target time (typically between 0.5 and 1.0 seconds, but can be shorter or longer, depending on the application requirements determined by the developer). For determining the optimal execution time, reading the original paper on argon2 by Biryukov et. al. is recommended [1].

iii. Encryption Algorithms

Insofar, the Citadel protocol supports the use of two encryption algorithms that provide authenticated encryption with associated data (AEAD) at 256-bit levels: AES-256-GCM-SIV and Xchacha20-Poly1305. The SIV variant of AES-256-GCM provides nonce misuse resistance for additional security. The X variant of Chacha20-Poly1305 also provides nonce misuse resistance for additional security. By letting the developer choose between the two AEADs, performance can be fine-tuned depending on the target machines. On machines without dedicated AES instructions, chacha generally runs faster (best for mobile/IoT devices).

III. Definitions

i. Citadel-Specific Definitions

Before describing how the aforementioned primitives are used together in order to construct the Citadel protocol, there are several Citadel-specific abstractions that must be defined in order to understand the system as a whole. Each will be discussed in detail in the following sections of this paper.

Entropy Store. *A set of cryptographically-secure random bytes that is symmetric between two endpoints used to derive nonces. Always transferred in encrypted form. Endowed with version n*

Security Level. *An unsigned 8-bit integer, n, that determines the number of symmetric keys, $n+1$, established per key exchange. Allows messages to use multi-layered encryption up to $n+1$ times. Default $n=0$*

Chain Key. *A recursively-generated key that is dependent upon the symmetric key established at the end of each KEM negotiation as well as the previous chain key*

Meta Chain Key. *Generated by taking the hash of the concatenation of every chain key in a symmetric key matrix*

Version. *The version endowed unto a symmetric row or matrix*

Symmetric Row. *A 4-tuple endowed with version n containing a key for Alice, a key for Bob, a chain key, and an entropy store. All elements in the row are symmetric between endpoints*

Symmetric Key Matrix. *a matrix endowed with version n and security level S containing $S+1$ symmetric rows*

ii. Notation

The following notations will be used to describe data and transformations thereof in Citadel's cryptographic protocol

Notation	Definition	Notes
$\varphi_{256}(x)$	SHA3-256(x)	One-way mapping of an input, x, to 256 bits
$\varphi_{512}(x)$	SHA3-512(x)	One-way mapping of an input, x, to 512 bits
$\rho(x)$	Argon2id(x, C)	Password hashing algorithm with parameters C
$A\|\|B$	concat(A, B)	Concatenation of two byte sequences A and B
$(A, B)_n$	split(X, n)	Split a byte sequence X at position n to obtain a tuple (A, B)
$A \oplus B$	$\forall a \ni A, \forall b \ni B, a \oplus b$	Xors each element in byte sequence A with each corresponding element in byte sequence B
$[E]_n$	Entropy store for symmetric row n	See definition
S	Security Level (unsigned 8-bit integer)	A value on the range [0, 256)
$count(S)$	$S + 1$	Determines the number of shared symmetric keys that should be generated by a key encapsulation mechanism given security level S
$[K]_n$	A symmetric key matrix with $count(S)$ rows by 4 columns. Version n	A matrix where each row holds a symmetric key for Alice, Bob, a chain key, and an entropy store $[E]_n$, respectively. Access notation is $[K]_n[row][column]$
λ	1 if Alice, 2 if Bob	Used to access a key in column λ of $[K]_n$ denoted by $[K]_n[row][\lambda]$
$\theta(P, K)$	AEAD algorithm	Encrypts the plaintext P using symmetric key K
$\omega(C, K)$	AEAD algorithm	Decrypts the ciphertext C using symmetric key K

iii. Definition of Subroutines

Throughout the protocol, multiple subroutines will be used in order to encrypt, decrypt, and derive keys to establish symmetric key matrices. The next page is dedicated to the subroutines of interest for the protocol.

```
$\mathbf{DeriveKeysInitial}(\mathbf{K_0})$
    $T_0 := \varphi_{512}(K_0)$
    $(K_{alice0}, K_{bob0})_{32} = split(T_0, 32)$
    $K_{chain0} := \varphi_{256}(K_{alice0} \oplus K_{kbob0})$
    $\textbf{return } [K_{alice0}, K_{bob0}, K_{chain0}]$
```

```
$\mathbf{DeriveMetaChainKey_n}([\mathbf{K}]_{(\mathbf{n-1})}, \mathbf{S_{max}})$
    $C_t := [K]_{n-1}[0][3] \ || \ .. \ || \ [K]_{n-1}[S_{max}][3]$
    $K_{meta(n)} = \varphi_{256}(C_t)$
    $\textbf{return } K_{meta(n)}$
```

```
$\mathbf{DeriveKeysChain_n}(\mathbf{K_{alice(n-1)}}, \mathbf{K_{bob(n-1)}}, \mathbf{K_{meta(n-1)}}, \mathbf{K_n})$
    $T_n := \varphi_{512}(K_{meta(n-1)} || K_n)$
    $(T_{alice(n)}, T_{bob(n)})_{32} = split(T_n, 32)$
    $K_{alice(n)} := \varphi_{256}(K_{alice(n-1)} \oplus T_{alice(n)})$
    $K_{bob(n)} := \varphi_{256}(K_{bob(n-1)} \oplus T_{bob(n)})$
    $K_{chain(n)} := \varphi_{256}(K_{alice(n)} \oplus K_{bob(n)})$
    $\textbf{return } [K_{alice(n)}, K_{bob(n)}, K_{chain(n)}]$
```

```
$\textbf{Encrypt}([K]_n, j, \lambda, S, P)$
    $\textbf{if } (j = S) \textbf{ then return } \theta(P, K_n[S][\lambda])$
    $\textbf{return Encrypt}([K]_n, \lambda, j+1, \theta(P, K_n[j][\lambda]))$
```

```
$\textbf{Decrypt}([K]_n, \lambda, S, P)$
    $\textbf{if } (S = 0) \textbf{ then return } \omega(P, K_n[0][\lambda])$
    $\textbf{return Decrypt}([K]_n, \lambda, S-1, \omega(P, K_n[S][\lambda]))$
```

IV. Protocol

The protocol is split into multiple phases: The client to server registration phase, the client to server connection phase, the peer to peer registration phase, the peer to peer connection phase, and the message-passing phase (where the latter is used by both client to server and peer to peer connections after a connection is established).

i. Client to Server Registration Phase

Client, M, selects the maximum static session security level S_{max} and initial post-quantum key encapsulation mechanism Ω_{static}. M selects username U and password P. P is immediately saved to memory as $\rho(P)$. Client M then begins execution of the *key exchange subphase*, which will be referred to throughout the document.

i.1 Key Exchange Subphase

Next, M obtains $count(S_{max})$ static public keys P_{static} using Ω_{static} exactly $count(S_{max})$ times. This will constitute the matrix:

$$\vec{A_b} = \begin{bmatrix} P_{static0} \\ \vdots \\ P_{static(S_{max})} \end{bmatrix}$$

$\vec{A_b}$ is then sent to a trusted central server $\hat{S}$ via the underlying protocol [1]. Upon receiving $\vec{A_b}$, $\hat{S}$ uses Ω_{static} to generate $count(S_{max})$ symmetric Keys K_{static} to obtain the column $[K_{static}]$. Next, for each row in K_{static}, $\hat{S}$ executes the subroutine **DeriveKeysInitial**(K_{static}) to obtain the matrix $([K_{alice}], [K_{bob}], [K_{chain}])$. Thereafter, $\hat{S}$ generates $count(S_{max})$ entropy banks $[E_{static}]$ to obtain the column $[[E_{static}]]$. To obtain the *symmetric key matrix*, $\hat{S}$ must then append the column $[[E_{static}]]$ to $([K_{alice}], [K_{bob}], [K_{chain}])$ to obtain the static symmetric key matrix:

$$[K]_{static} = \begin{bmatrix} K_{alice0} & K_{bob0} & K_{chain0} & [E]_0 \\ \vdots & \vdots & \vdots & \vdots \\ K_{alice(S_{max})} & K_{bob(S_{max})} & K_{chain(S_{max})} & [E]_{S_{max}} \end{bmatrix}$$

The goal is now to establish $[K]_{static}$ on the endpoint for M. To accomplish this, $\hat{S}$ uses Ω_{static} to obtain $count(S_{max})$ ciphertexts C to form the column $[C]$. Next, for each entropy bank $[E]$ in $[K_{static}][row][4]$, $\hat{S}$ executes the subroutine **Encrypt**$([K]_{static}, 0, 1, S_{max}, [E])$ to obtain the encrypted column $[[\hat{E}]]$. $\hat{S}$ then concatenates $[C]$ to $[[\hat{E}]]$ to obtain the matrix:

$$\vec{B_a} = \begin{bmatrix} C_0 & [\hat{E}]_{static0} \\ \vdots & \vdots \\ C_{S_{max}} & [\hat{E}]_{static(S_{max})} \end{bmatrix}$$

$\hat{S}$ then sends $\vec{B_a}$ to M over the underlying protocol. Thereafter, for each C in $\vec{B_a}[row][1]$, M uses Ω_{static} to obtain $count(S_{max})$ symmetric keys K_{static} to obtain the column $[K_{static}]$. Next, for each row in K_{static}, M executes the subroutine **DeriveKeysInitial**(K_{static}) to obtain the matrix $([K_{alice}], [K_{bob}], [K_{chain}])$. To obtain the symmetric key matrix, M must first decrypt each $[\hat{E}]$ in $\vec{B_a}[row][2]$ using the subroutine **Decrypt**$([K]_{static}, 1, S_{max}, [\hat{E}])$to obtain the column $[[E]]$. Finally, M concatenates $[[E]]$ to the matrix $([K_{alice}], [K_{bob}], [K_{chain}])$ to form the symmetric key matrix $[K]_{static}$.

That concludes the key exchange subphase. The same process with few alterations are used in the future phases to renegotiate keys to ensure perfect forward secrecy of messages.

i.2 Credential Registration

The next step is to securely establish the authenticity of the user using the device. To accomplish this, M sends **Encrypt**$([K]_{static}, 0, 0, S_{max}, U||\rho(P))$ to $\hat{S}$. Upon reception, $\hat{S}$ executes **Decrypt**$([K]_{static}, 1, S_{max}, U||\rho(P))$ to obtain

[1]The underlying protocol can be either TCP or TLS

U and $\rho(P)$. $\hat{S}$ ensures U is unused, and if so, then $\hat{S}$ stores U and $\rho(\rho(P))$ to the backend [2].

$\hat{S}$ concludes the registration process by sending an encrypted welcome message back to M. If M successfully decrypts the message, then the registration phase is concluded.

ii. Client to Server Connection Phase

M selects a maximum session security level S_{sess} and computes $\rho(P_{in})$ from a recent password input P_{in}. M also accepts username U_{in}. M verifies that $S_{sess} \leq S_{max}$ to ensure all chains in $[K]_{static}$ can recursively be input into the generator for the next symmetric key matrix $[K]_0$. Next, M performs the key exchange subphase with $\hat{S}$ used in the connection phase, but with several alterations. Instead of calculating $[K]_{static}$, the symmetric key matrix $[K]_0$ will be generated using the **DeriveKeysChain** subroutine. Additionally, instead of using the maximum row count of $count(S_{max})$ for the matrices, the maximum row count $count(S_{sess})$ is used.

After establishing the symmetric key matrix $[K]_0$ between M and $\hat{S}$, M then sends **Encrypt**$([K]_0, 0, 0, S_{sess}, U_{in}||\rho(P_{in}))$ to $\hat{S}$. Upon reception, $\hat{S}$ then executes **Decrypt**$([K]_0, 0, S_{max}, U_{in}||\rho(P))$ to obtain U and $\rho(P)$. $\hat{S}$ then verifies that $U_{in} = U$ and $\rho(\rho(P_{in})) = \rho(\rho(P))$ from the values stored in the backend. If successful, then M is logged-in to $\hat{S}$. $\hat{S}$ concludes the connection phase by sending an encrypted welcome message back to M.

iii. Peer to Peer Registration Phase

In the peer to peer registration phase, the trusted central server acts as an observer of the registration between peers to assist in Network Address Translation (NAT) Traversal and peer discovery for the connection phase. Importantly, all packets sent between peer to server to the adjacent peer are encrypted using the symmetric key matrix generated in the client to server connection phase, thus transforming the ciphertext payload as the packet pivots at the trusted central server.

Before continuing, one clarification must be made. Each peer connected to the central server is endowed with a unique client identification number (CID) derived during the client to server registration phase. This is used for identification during peer discovery. The Citadel Protocol returns a set of CIDs on-demand to registered clients, which the client can then choose from in order to post registration requests. The IP address is not needed to forge these connections since they are only known by $\hat{S}$ (the IP addresses of each peer are not known until *after* registration and during the peer to peer connection phase where NAT traversal is attempted).

We will assume that two peers, p_0 and p_1, both registered to a trusted central server $\hat{S}$, wish to communicate with each other. Before the two peers connect, either one of them must post a registration request containing the unique CID of the other peer. In this hypothetical case, p_0 posts a registration request to $\hat{S}$ containing the CID of the p_1. If p_1 is online, the registration request is immediately routed to p_1. If p_1 is offline, the message is stored to the backend where it will be delivered once p_1 goes back online.

Once p_1 receives the registration request, p_1 can choose to accept or deny it. p_1 then sends the response back to $\hat{S}$, where the request is forwarded back to p_0. If p_1 accepted the request, either p_0 or p_1 can then post a *connection* request to the other in order to engage in encrypted message passing using a unique symmetric key matrix independent to the symmetric key matrix used between client and server.

iv. Peer to Peer Connection Phase

Once two peers p_0 and p_1 have consented to registration, connections between the two peers may ensue. The role of the trusted central server $\hat{S}$ is to act as a node behind the NAT of p_0 and p_1 to help forge a connection between the two peers. $\hat{S}$ also acts as an observer to verify that neither registration between the peers

[2]The backend is either the local filesystem or a remote database. In the later case, TLS is used by default

has been revoked.

Using the key exchange subphase with the **DeriveKeysInitial** subroutine with $\hat{S}$ for packet routing, both nodes establish an initial symmetric key matrix $[K]_0$. Thereafter, $\hat{S}$ passes the socket address of the peers so that they may attempt TCP hole punching concurrently to program execution. Once the TCP hole punching subroutine finishes, the connection will automatically be upgraded from using TURN-like routing using $\hat{S}$ to a direct bidirectional peer to peer connection between p_0 and p_1.

If TCP hole-punching fails, the packets between p_0 and p_1 will use $\hat{S}$ for routing packets. Since the symmetric key matrix $[K]_0$ is only known by p_0 and p_1, $\hat{S}$ has no knowledge of $[K]_0$ as desired.

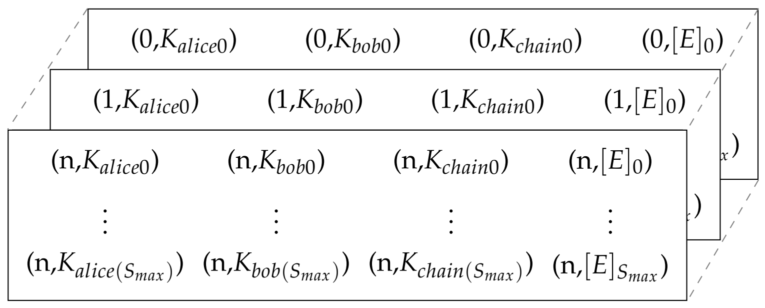
v. Message Passing Phase

Messages on the Citadel network can be passed between two peers or a client and the trusted central server (and vice-versa) after the connection phase is complete. The frequency of re-keying is determined by the *secrecy mode* set by the node that initiated the connection. When re-keying occurs, the message plaintext is concatenated with the information required to initiate a re-key to both deliver and update the symmetric key matrix on both endpoints, and thereafter, encrypted using the most recent symmetric key matrix version.

Perfect Forward Secrecy Mode (PFS). *Each packet that is sent outbound from either node is guaranteed to be encrypted using a unique symmetric key matrix to prevent key re-use. Packets sent out before re-keying is complete are enqueued for delivery. Enqueued packets are not sent outbound until being asynchronously triggered by the completion of a re-key*

Best Effort Mode (BEM). *Messages are sent outbound independent to whether or not a re-key is in process. If no re-key is in process, a new re-key is triggered*

A challenge in BEM is that despite the use of TCP or TLS, order delivery is not guaranteed. The Citadel Protocol processes packets concurrently, allowing multiple asynchronous packet processing subroutines to work cooperatively. Since the underlying asynchronous Tokio runtime does not guarantee the order of poll completion, the order in which packets are processed is not guaranteed. As such, a series of synchronization primitives are used in the protocol to ensure no conflicts exist even under highly stressed loads. The exact nature of those primitives is outside of the scope of this paper, but exist within the source code.



$(0,K_{alice0})$ $(0,K_{bob0})$ $(0,K_{chain0})$ $(0,[E]_0)$
$(1,K_{alice0})$ $(1,K_{bob0})$ $(1,K_{chain0})$ $(1,[E]_0)$
(n,K_{alice0}) (n,K_{bob0}) (n,K_{chain0}) $(n,[E]_0)$
$\vdots$ $\vdots$ $\vdots$ $\vdots$
$(n,K_{alice(S_{max})})$ $(n,K_{bob(S_{max})})$ $(n,K_{chain(S_{max})})$ $(n,[E]_{S_{max}})$

Figure 1: *3D Symmetric Key Matrix, version n*

As re-keying occurs, the version of the symmetric key matrix continues to increment: $[K]_0 \Rightarrow [K]_1 \Rightarrow [K]_2 \Rightarrow ... \Rightarrow [K]_n$. Since in BEM mode, packets of differing versions may arrive, a history of these symmetric key matrices are kept in memory in a bounded queue (figure 1) that drops off the oldest element after the capacity is reached to help keep memory usage to a minimum.

V. Additional Technical Specifications

i. Hybrid Protocol

Until the KEM finalists of the NIST's post-quantum competition are released, cryptographers recommend the use of hybrid cryptography. This means that the Citadel Protocol should use an underlying TLS protocol to encrypt all traffic. Before booting the Citadel protocol on a trusted central server, a server admin may specify a PKCS-12 formatted identity, optional TLS domain, and optional password to ensure all clients connecting to the central server use TLS to encrypt traffic from the Citadel protocol.

ii. Miscellaneous Security Features

The Citadel protocol is impervious to replay attacks by appending an unsigned 64-bit integer, *pid*, at the end of each input plaintext byte sequence. The concatenation of the two is input into the AEAD algorithm. Since the processing of packets is unordered, a short history of previous packets is stored to ensure no two packets received have the same *pid*.

The Citadel Protocol also has passive background re-keying determined by the maximum session security level. Background re-keying occurs in both PFS/BEM configurations. When a session is idled, this helps bolster the security of future transmissions. In cases where a hacker captures packets implicated in the key exchange phase in order to begin discovering the symmetric key matrices, passive background re-keying helps confine the temporal window where a symmetric key matrix is valid; by the time the symmetric key matrix is (hypothetically) discovered, a new symmetric key matrix is possibly available, forcing the hacker to chase a moving target.

Side-channel attacks via an external application accessing the memory displaced by the Citadel protocol are mitigated by using *mlock* on Linux/Mac and *VirtualLock* on Windows. Specifically, any time plaintext or passwords are in memory, a call to the underlying operating system is made to protect access to the memory. Furthermore, the memory of such information is zeroed on drop.

iii. Bimodal Execution

Each node on the Citadel network may operate as either a client, a server, or *both*. This implies that a client that makes outgoing connections to other servers (and peers thereon) can simultaneously act as a trusted central server for other peers to connect to. The interaction between all this traffic is propagated to a unified *NetKernel* asynchronous interface, allowing the SDK developer to create complex network topologies using the Citadel protocol.

iv. Selectable Backends

By default, the Citadel protocol uses the local filesystem to store required information for data persistence. In cases where only one server is needed to handle traffic of an application, this is recommended. However, when multiple servers are needed to load-balance traffic, the Citadel protocol also has built-in support to use SQL databases[3] for unified data persistence across multiple nodes. When SQL is used, TLS is required.

In mobile contexts, using sqlite for the backend is also recommended since firebase messages may be delivered asynchronously in the background, triggering sporadic wakeups; by synchronizing data to a database, the protocol doesn't have to reload the files each time the protocol boots.

v. Google Firebase Compatibility

When building mobile applications, a particular challenge is the how the operating system (e.g., Android and IOS) shuts down the application when in the background. Without execution, ordinary connections between peers would be rendered ineffective. To remedy the issue, the Citadel Protocol has compatibility with Google Firebase to automatically generate custom authentication tokens at the trusted central server that way clients can use Firebase's Realtime Database for packet transmission.

vi. Multi/Single Threaded Event Loops

At compile-time, a developer can choose between using multi-threaded or single-threaded event loops for the Citadel protocol. In general, a single-threaded asynchronous event

[3]MySQL, Sqlite, and PostgreSQL are currently supported

loop will have lower latency and lower maximum throughput, whereas a multi-threaded asynchronous event loop will have higher latency and higher throughput. In general, it is recommended that when building applications for clients on the edge, using a single-threaded event loop is optimal. For central servers expecting heavy traffic, a multi-threaded event loop is recommended.

vii. Rust

The Citadel protocol was built using the Rust programming language. The use of *safe* rust is mathematically-guaranteed [5] to be free of memory leaks, dangling pointers, and memory races. The Citadel protocol uses **no unsafe code** other than mandatory unsafe when interacting with foreign function interface boundaries used to call mlock and VirtualLock.

The use of nightly Rust is currently required to take advantage of the latest paradigms available in the Rust programming language. In the future, stable Rust will be used.

VI. Discussion

The extensive use of recursion in generating symmetric key matrices implies that the more re-keys performed in a session, the harder it becomes for a hacker to break through the encryption of the latest message. In other words, assuming the use of PFS mode, breaking the encryption of message m encrypted with the symmetric key matrix $[K]_n$ will require that the hacker discovers the symmetric key matrices for $[K]_n, [K]_{n-1}, [K]_{n-2}, ...[K]_0$. Of course, a hacker could always attempt generating random 256-bit symmetrical keys until they discover the plaintext, but they too would have to correctly uncover the correct entropy store which generates the correct nonce input into the AEAD algorithm. The chances that both occur are essentially nil in human time spans.

Since the difficulty in breaking the encryption increases for each message sent, a user could optionally call upon the Citadel protocol to begin re-keying multiple times before sending messages. This is mathematically unnecessary, but demonstrates how *some* users may decide to manually tune the security by taking advantage of the inherent design of the Citadel protocol.

i. Applications

The Citadel protocol, supporting the use of up to 256 layers of multi-layered encryption and PFS/BEM configurations means developers can easily balance security and performance. In cases where high security is needed, selecting PFS mode at the start of a session is recommended (e.g., messaging apps). When speed is a requirement, BEM can be selected instead (e.g., web servers and VPNs using a TUN/TAP interface). Since BEM implies that multiple packets/messages may use the same symmetric key for encryption, increasing the maximum security level at the start of each session allows the developer to effortlessly apply multiple layers of encryption for each individual message to force a hacker to discover multiple keys instead of just one. In cases where speed is not an issue, and, a paranoid degree of security is required, the developer can use PFS coupled with 256 layers of encryption.

References

[1] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. pages 292–302, 2016.

[2] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson. The memory-hard Argon2 password hash and proof-of-work function. Internet-Draft draft-irtf-cfrg-argon2-13, Internet Engineering Task Force, Mar. 2021. Work in Progress.

[3] I. T. L. Computer Security Division. Public-key post-quantum cryptographic algorithms: Nominations, Dec 2016.

[4] M. J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, July 2015.

[5] R. Jung. Understanding and evolving the rust programming language. 2020.

[6] D. Moody. Let's get ready to rumble: The nist pqc competition, April 2018.

[7] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, Oct 1997.