The Ficcus Handshake

ccertain *

October 28, 2026

1 Sketch

Two ficcus nodes talk over a byte stream: a TCP connection, or a WebSocket for nodes running in a browser. Before any message is exchanged the stream goes through a handshake that identifies both peers, makes the peer that opened the connection pay a proof of work, exchanges public keys if needed, and agrees on a pair of symmetric keys, one per direction. Keys are agreed with ML-KEM [2] and bound to both peers' long term ML-DSA [3] identities by signatures. Once two peers share keys they keep them: later connections skip the key exchange and derive fresh keys from the stored ones by hashing.

After the handshake every message travels in a frame encrypted with ChaCha20 [4] under the sender's key, carrying a BLAKE3 [5] hash of its content. Inside a frame, a message is a type prefix, a request id and a postcard encoded body.

The implementation is `ficcus/src/network/connection.rs` (handshake, frames), `ficcus/src/network/symmetric.rs` (key state) and `ficcus/src/messages/` (messages).

Notation. H is BLAKE3 with 32 byte output, $H_k(x)$ its first k bytes, $\|$ concatenation. $\mathrm{be}_{32}(n)$ is n as 4 bytes big endian. All integers written during the handshake are big endian. The peer that opened the connection is the *opener*, the other the *acceptor*.

2 Identities

A node has a long term ML-DSA signing key of type ML-DSA-44, ML-DSA-65 or ML-DSA-87, numbered 0, 1 and 2; nodes make ML-DSA-65 keys. Its identity is

$$\mathrm{id} = H_{24}(\texttt{"ficcus_signing_key_"} \,\|\, \text{type name} \,\|\, \texttt{":"} \,\|\, \text{public key}),$$

with type name `MLDSA44`, `MLDSA65` or `MLDSA87`. An identity determines its public key: a public key received for an identity is accepted only if it hashes to it. Nodes keep the public keys they have seen in a registry, and identities are compared as byte strings, which orders them.

3 Transport

Nodes accept TCP connections on port 6769. Browsers can't open TCP connections, so nodes also accept WebSocket connections on their HTTPS server at the path `/ficcus/ws`; the stream is carried in binary WebSocket messages and is otherwise the same. Opening a TCP connection

*ccertain@tutanota.com

may take 10 seconds. The handshake below has to finish within 30 seconds of the stream opening, or the connection is closed.

4 Handshake

Both peers run the same steps. In each step a peer writes its part, then reads the other's, so the two directions proceed at the same time; where a step is asymmetric the table says who writes.

4.1 Identification

from	bytes	content
both	8	prefix, eight bytes of value 99
both	24	own identity

A peer closes the connection if the prefix doesn't match. The opener usually knows the identity it dialed, from the address or a previous connection, and closes the connection if the acceptor's identity differs. Nothing is proven yet: the identity is a claim until the key exchange binds it to a signature, or a stored key confirms it.

4.2 Proof of Work

The acceptor sends a 32 byte seed and a 16 byte target, and the opener answers with an 8 byte nonce that solves it, as specified in [1], Section 3, with the puzzle of [1], Section 2. The difficulty depends on how often the opener's address connected lately. The opener refuses challenges above 28 bits, and the acceptor closes the connection if the nonce doesn't solve the challenge. This step comes before every expensive step that follows: public key parsing, ML-KEM, signature verification and database writes.

4.3 Public Keys

from	bytes	content
both	1	1 if it needs the other's public key, 0 otherwise
asked	1	signing key type, 0 to 2
asked	4	public key length ℓ
asked	ℓ	public key

A peer asks for the public key if its registry doesn't have one for the claimed identity. A peer that is asked sends its key type and public key. A received key is rejected if ℓ exceeds 100 KiB or the key doesn't hash to the claimed identity; otherwise it goes into the registry.

4.4 Key Reuse

Order the two identities as $a < b$. A key state for the pair (Section 5) may be stored from a previous connection.

from	bytes	content
both	32	the hash of its stored key state for the pair, or 32 zero bytes

If a peer has a stored state and the other sent the same hash, both reuse it: each reseeds the state (Section 5), stores the reseeded state in place of the old one, and the handshake is complete. Reseeding, and with it the handshake, fails unless the stored state holds both peers' signatures from the key exchange that made it. Otherwise the peers run the key exchange.

4.5 Key Exchange

The peer with the smaller identity is the *initiator*.

from	bytes	content
initiator	1	KEM type, 1 for ML-KEM-768
initiator	4	encapsulation key length e
initiator	e	ML-KEM-768 encapsulation key ek, from a new key pair
other	4	ciphertext length c
other	c	ciphertext ct encapsulating a shared secret ss to ek

The other peer rejects KEM types other than ML-KEM-768 (0 and 2 are reserved for ML-KEM-512 and ML-KEM-1024) and lengths above 4096 bytes; the initiator rejects ciphertexts above 4096 bytes, and decapsulates ss. The key pair is used once. Both compute the key state of Section 5 from ss and kem_id $= H(ek)$, and then authenticate it:

from	bytes	content
both	4	signature length g
both	g	ML-DSA signature over the state hash h

Each peer signs h with its identity key and verifies the other's signature against the other's public key; signatures above 8192 bytes are rejected. The state hash covers both identities, both keys and kem_id, so each signature binds its signer to this exchange's encapsulation key and to the keys derived from it. Both peers store the state with both signatures, and the handshake is complete.

5 Key State

For identities $a < b$ and a shared secret ss of at least 32 bytes, the key state is

$$k_a = H(a \,\|\, ss), \qquad k_b = H(b \,\|\, ss),$$

with kem_id and, once known, the signatures of a and b. Key k_a encrypts what a sends, k_b what b sends. The state hash is

$$h = H(a \,\|\, k_a \,\|\, b \,\|\, k_b \,\|\, \text{kem_id}).$$

Reseeding replaces each key by its hash, $k_a \leftarrow H(k_a)$ and $k_b \leftarrow H(k_b)$, and restarts both keystreams. Every connection that reuses a stored state reseeds it first, so successive connections use different keys (but see Remark 2), and the stored keys move one step along the hash chain per connection. The signatures stay those of the exchange that started the chain.

Keystreams. Each direction is a ChaCha20 [4] keystream under its key with an all zero nonce, starting at position 0 when the state is created or reseeded. Every byte a peer sends in a frame is XORed with the next byte of its keystream, and every byte it receives with the next byte of the other's. Stream positions are not stored. A position is a 32 bit byte count, so a direction can carry 4 GiB before the connection fails.

6 Frames

After the handshake each message is sent as a frame. For a message m of length $|m| \leq 10$ MiB, the sender picks 8 random bytes r, and with $L = \text{be}_{32}(8 + |m|)$ sends:

bytes	content, each XORed with the sender's keystream in this order
16	$H_{16}(L \,\|\, r \,\|\, m)$
4	L
8	r
$\|m\|$	m

The receiver decrypts the hash and the length, rejects lengths below 8 or above 10 MiB plus 8, reads and decrypts the rest, closes the connection if the hash doesn't match, and drops r.

The hash isn't keyed, so what keeps a frame from being changed is r: it is random and only travels encrypted, so someone who knows m still can't compute the hash of a replacement message, $H_{16}(L \,\|\, r \,\|\, m')$, needed to fix up the encrypted hash. Changing a frame comes down to guessing r, with probability 2^{-64} per try, and a wrong guess closes the connection. Once a frame has started arriving the rest has to arrive within 30 seconds; a frame has to be written within 30 seconds.

7 Messages

A message is

bytes	content
8	type prefix, H_8 of the type's name and field types, e.g. `RequestNamespace(NamespaceId,Option<[u8;32]>)`
1	kind: 0 no response expected, 1 request, 2 response
varies	for kinds 1 and 2, a request id, a postcard `u64`
varies	the fields, each postcard encoded, in order

A request carries a random id and its response the same id; a request that gets no response within 30 seconds fails. A peer can send a burst of 5000 messages and 500 a second after that; a peer that sends faster is disconnected ([1], Section 4).

8 Connection Lifetime

A completed handshake makes the connection the peer's connection, used for every message to it. If a peer already has an open connection, a new one opened by the same side replaces it. When the two were opened by different sides, e.g. both peers dialed each other, both keep the one opened by the peer with the smaller identity, so they keep the same one. A connection closes when either peer closes the stream, on a frame that fails to decrypt, verify or arrive in time, or when the peer overspends its message budget.

9 Friends

Two identities can also run the handshake with no connection between them, over namespaces, to talk privately on the public network. The implementation is `ficcus/src/friends/`.

Namespaces. For identities x and y, x writes to its namespace with subject

$$\text{friend}(y) = H(\texttt{"ficcus friend"} \,\|\, y)$$

and y to its own with subject friend(x), so each namespace carries what one sends to the other. Namespaces replicate to every node that follows them, so each side reads the other's as it arrives. Each write below is followed by a publish.

Attempts. A handshake is an *attempt*, named by 8 random bytes t, written in paths as Crockford base32. The side that wants to talk, the opener, writes the file `/attempt/`t, removing any earlier one, and runs the handshake as the opener. Its first bytes, the identification, show the other side's user a request. If the user accepts, the other side runs the handshake as the acceptor, and from then on answers the other's new attempts without asking. If both open an attempt, both run the one of the smaller identity, and the other side closes its own.

Stream. Each side's half of the handshake's byte stream goes into files of its namespace, `/handshake/`t`/`n, for $n = 0, 1, \ldots$ written as 6 decimal digits, and each side reads the other's half from the same files of the other's namespace, in order. A file holds what was written in a burst, usually a step of the handshake. A side that gives up writes `/handshake/`t`/closed`, and a closed attempt is neither answered nor waited on. The opener waits for the acceptor's first file for as long as it takes the user to accept; after that each file has to arrive within 5 minutes, in place of the 30 second handshake timeout. The proof of work is that of an address with no recent connections, 8 bits. Friends keep no key state: they send 32 zero bytes in the Key Reuse step, so every attempt runs the key exchange, and the state isn't stored.

Messages. From the key state of a completed attempt each side derives a key per direction, $c_x = H(\texttt{"ficcus friend channel"} \,\|\, t \,\|\, k_x)$ for what x sends, with k_x the key of Section 5, and stores them. The i-th message x sends under t, from $i = 0$, is the file `/messages/`t`/`i of x's namespace, i written as 10 decimal digits. It is a frame (Section 6) of $\mathrm{be}_{64}(\mathrm{ms}) \,\|\, \mathrm{text}$, with ms the unix time in milliseconds it was sent and the text in UTF-8 of at most 16 KiB, but where a connection's frames continue one keystream, a message has its own: ChaCha20 under c_x with the 12 byte nonce $0^4 \,\|\, \mathrm{be}_{64}(i)$, from position 0. Messages persist, and a stream position kept in memory would be used again after a restart. The reader discards a message whose hash doesn't match.

10 Remarks

These are properties of the handshake as specified, for the review of the transport cryptography.

Remark 1. *Key reuse is decided by comparing state hashes sent in the clear by both peers at once. A peer that claims an identity without holding the state can send back the hash it receives, and the node reseeds and stores the state as for a genuine reuse. The impostor can't derive the reseeded keys, so it can neither read the node's frames nor send one that verifies, and the connection fails at the first frame the impostor sends. Until then it has replaced the genuine peer's connection, and the genuine peer's stored state is now a step behind, so their next connection runs a fresh key exchange. Reuse authenticates by possession of the state only through the frames that follow.*

Remark 2. *Two connections between the same pair that reuse the stored state at the same time both reseed the same stored state and so derive the same keys, encrypting with the same keystream. One of the two connections is closed once both complete, but frames may have been sent on both before.*

Remark 3. *Stored states hold the current keys, so whoever reads a node's database can read the frames of the current and later connections with that peer, though not those of earlier connections, since the hash chain only runs forward.*

Remark 4. *A handshake between friends (Section 9) is public, and so are the messages that follow: anyone can read both identities, the encapsulation key, the ciphertext and both signatures, and see who talks to whom, when, and how long the messages are. Only the shared secret and what the messages say stay private. A message's integrity rests on the namespace's signatures, which only its owner can make, not on the frame's hash, which isn't keyed: anyone holding c_x can make a frame that verifies.*

11 Constants

constant	value
identification prefix	eight bytes of value 99
TCP port, WebSocket path	6769, `/ficcus/ws`
TCP connect timeout	10 s
handshake timeout	30 s
largest public key	100 KiB
largest KEM key or ciphertext	4096 bytes
largest signature	8192 bytes
largest message	10 MiB
frame read and write timeouts	30 s
request timeout	30 s
proof of work	see [1]

References

[1] ccertain, Rate Limiting in Ficcus, `pow/spec.pdf` in the ficcus repository.

[2] National Institute of Standards and Technology, Module-Lattice-Based Key-Encapsulation Mechanism Standard, FIPS 203, 2024, `https://doi.org/10.6028/NIST.FIPS.203`

[3] National Institute of Standards and Technology, Module-Lattice-Based Digital Signature Standard, FIPS 204, 2024, `https://doi.org/10.6028/NIST.FIPS.204`

[4] Yoav Nir and Adam Langley, ChaCha20 and Poly1305 for IETF Protocols, RFC 8439, 2018, `https://www.rfc-editor.org/rfc/rfc8439`

[5] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves and Zooko Wilcox-O'Hearn, BLAKE3: one function, fast everywhere, 2020, `https://github.com/BLAKE3-team/BLAKE3-specs`