NOMAD: A Secure State Synchronization Protocol for Mobile Applications over UDP

Danyiel Colin
amaniel2718@protonmail.com

Abstract

We present NOMAD (Network-Optimized Mobile Application Datagram), a secure UDP-based state synchronization protocol for real-time applications over unreliable networks. Building on insights from MOSH (Mobile Shell), NOMAD combines the NOISE Protocol Framework's IK handshake pattern with XChaCha20-Poly1305 authenticated encryption to achieve mutual authentication in a single round-trip. The protocol features epoch-based rekeying for forward secrecy, seamless connection migration across IP address changes without reconnection, and an idempotent state synchronization layer that guarantees convergence despite packet loss and reordering. We formally verify the protocol's security properties using ProVerif and TLA+, discovering and fixing a post-compromise security vulnerability in the rekeying mechanism. A conformance test suite with over 1,000 test cases and a Rust reference implementation validate the design.

Keywords: UDP, state synchronization, Noise Protocol, authenticated encryption, mobile networking, roaming

1 Introduction

Remote shell access remains fundamental to system administration and software development. While SSH [9] provides secure connectivity, its TCP-based design leads to poor user experience over unreliable networks: connections freeze during packet loss, timeout during IP address changes, and require complete session reestablishment after network interruptions.

MOSH (Mobile Shell) [8] addressed these limitations through a UDP-based State Synchronization Protocol (SSP), demonstrating that state synchronization with client-side prediction could achieve superior responsiveness. However, MOSH's design is now over a decade old, and several aspects warrant modernization:

- **Cryptographic primitives**: MOSH uses AES-128-OCB, while modern protocols favor ChaCha20-Poly1305 for software implementations
- **Key exchange**: MOSH bootstraps from SSH, whereas dedicated protocols like WireGuard [2] show the benefits of integrated key exchange
- **State generality**: MOSH is tightly coupled to terminal state, limiting reuse for other applications

We present NOMAD, a new protocol that retains MOSH's core insight—that idempotent state synchronization enables reliable operation over unreliable transport—while incorporating modern cryptographic practices and a generic state interface. NOMAD is *not* backward-compatible with MOSH, enabling a clean design unconstrained by legacy considerations.

1.1 Contributions

Our contributions are:

1. A complete protocol specification across security, transport, and synchronization layers, using the NOISE Protocol Framework [7] for 1-RTT mutual authentication
2. An epoch-based rekeying mechanism providing forward secrecy with nonce namespace separation, preventing nonce reuse across key rotations
3. Seamless connection migration where sessions survive IP address changes through cryptographic session identifiers, without requiring reconnection handshakes
4. An idempotent state synchronization algorithm that guarantees convergence despite arbitrary packet loss and reordering
5. A conformance test suite with over 1,000 test cases using property-based testing, test vectors

generated from reference cryptographic libraries, and infrastructure for testing implementations in any language via Docker containers

6. Formal verification using ProVerif and TLA+, including discovery and fix of a post-compromise security vulnerability in the rekeying mechanism

1.2 Paper Organization

Section 2 reviews relevant background on Mosh, the Noise Protocol Framework, and related work. Section 3 presents the protocol design and layer architecture. Section 4 details the security layer including handshake and rekeying. Section 5 describes the transport layer with roaming support. Section 6 explains the synchronization layer and convergence guarantees. Section 7 covers the extension mechanism. Section 8 presents our conformance testing methodology. Section 9 describes formal verification results including a discovered vulnerability and fix. Section 10 discusses related work, and Section 11 concludes.

2 Background

2.1 Mosh and State Synchronization

Mosh [8] introduced the State Synchronization Protocol (SSP) for remote terminal access. Its key insight is that for interactive applications, only the *current* state matters—intermediate states can be skipped if the network is slow. This contrasts with TCP's reliable ordered delivery, which can cause unbounded buffering when bandwidth varies.

SSP maintains version numbers for local and remote state. Each frame carries a diff from a known base state to the current state. Because diffs are *idempotent*—applying the same diff twice has no additional effect—duplicates and reorderings are harmless. The protocol simply always sends the diff from the last-acknowledged state to the current state.

2.2 The Noise Protocol Framework

The Noise Protocol Framework [7] provides a rigorous foundation for designing cryptographic handshake protocols. Noise defines handshake patterns specifying the sequence of Diffie-Hellman operations and when static keys are transmitted or known in advance.

Nomad uses the `Noise_IK` pattern:

```
Noise_IK(s, rs):
  <- s
  -> e, es, s, ss
  <- e, ee, se
```

In this pattern, the responder's static key (`s`) is known to the initiator beforehand (line 1). The initiator sends an ephemeral key and their encrypted static key (line 2), and the responder replies with an ephemeral key (line 3). The pattern provides mutual authentication in one round-trip, with identity hiding for the initiator.

2.3 Authenticated Encryption

Nomad uses XChaCha20-Poly1305 [6] for authenticated encryption with associated data (AEAD). XChaCha20 extends ChaCha20's 12-byte nonce to 24 bytes, enabling random nonce generation without birthday-bound collision concerns. Poly1305 provides a 128-bit authentication tag.

3 Protocol Design

3.1 Design Goals

Nomad targets the following goals:

1. **Security**: End-to-end authenticated encryption with forward secrecy

2. **Mobility**: Seamless operation across IP address changes

3. **Latency**: Sub-100ms perceived latency via prediction

4. **Simplicity**: Fixed cryptographic suite, no negotiation

5. **Generality**: State-agnostic synchronization framework

3.2 Non-Goals

To maintain simplicity, Nomad explicitly does not provide:

- Backward compatibility with Mosh/SSP

- Cipher suite negotiation

- Reliable ordered delivery (applications handle via state sync)

- Multiplexing multiple state types per session

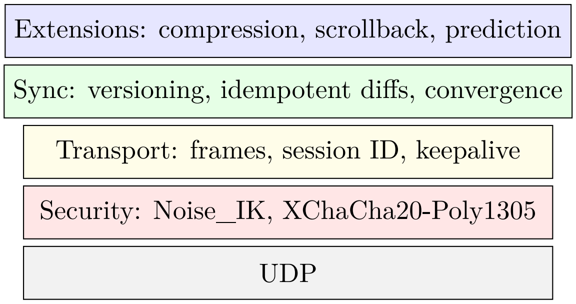
Extensions: compression, scrollback, prediction
Sync: versioning, idempotent diffs, convergence
Transport: frames, session ID, keepalive
Security: Noise_IK, XChaCha20-Poly1305
UDP

Figure 1: NOMAD protocol layer stack.

3.3 Layer Architecture

Nomad is organized into four layers, shown in Figure 1:

Security Layer Handles the Noise IK handshake, AEAD encryption, and periodic rekeying for forward secrecy.

Transport Layer Manages frame construction, session identification via 48-bit session IDs, connection migration (roaming), RTT estimation, and keepalive.

Sync Layer Implements versioned state synchronization with idempotent diffs and acknowledgment tracking.

Extensions Optional features negotiated during handshake: compression, scrollback, and client-side prediction.

3.4 Cryptographic Suite

Nomad uses a *fixed* cryptographic suite with no runtime negotiation:

Purpose	**Algorithm**
Key Exchange	X25519
AEAD Cipher	XChaCha20-Poly1305
Hash Function	BLAKE2s-256
Key Derivation	HKDF-BLAKE2s

Table 1: Fixed cryptographic suite.

If vulnerabilities are discovered, a new protocol version is released rather than negotiating alternatives. This follows WireGuard's philosophy of "cryptographic versioning."

4 Security Layer

4.1 Handshake Protocol

The handshake establishes a session between initiator and responder in one round-trip. The initiator must know the responder's static public key beforehand (obtained via SSH, QR code, or other secure channel).

4.1.1 Handshake Initiation

The initiator sends:

- Protocol version (2 bytes)
- Ephemeral public key (32 bytes, unencrypted)
- Static public key (32 bytes + 16 byte tag, encrypted)
- Payload with state type ID and extensions (encrypted)

The initiator's static key is encrypted after the first DH operation, providing identity hiding from passive observers.

4.1.2 Handshake Response

The responder replies with:

- Session ID (6 bytes)
- Ephemeral public key (32 bytes, unencrypted)
- Acknowledgment and negotiated extensions (encrypted)

After successful handshake, both parties derive symmetric session keys using HKDF with the handshake transcript hash.

4.2 Session Keys

Session keys are derived as:

$$(k_i, k_r) = \mathrm{HKDF}(h, \text{“nomad v1 keys”}, 64) \quad (1)$$

where k_i and k_r are the initiator and responder keys, and h is the handshake hash.

The initiator uses k_i for sending and k_r for receiving; the responder uses the inverse.

4.3 Nonce Construction

XChaCha20-Poly1305 requires 24-byte nonces. Nomad constructs nonces as:

The epoch field provides nonce namespace separation across rekeying operations, ensuring nonce uniqueness even as counters reset.

Field	Size	Description
Epoch	4 bytes	Current epoch number
Direction	1 byte	0x00 or 0x01
Zeros	11 bytes	Padding
Counter	8 bytes	Per-direction counter

Table 2: Nonce structure (24 bytes total).

4.4 Rekeying

Sessions rekey periodically for forward secrecy. Table 3 shows the timing constants.

Constant	Value
`REKEY_AFTER_TIME`	120 seconds
`REJECT_AFTER_TIME`	180 seconds
`REKEY_AFTER_MESSAGES`	2^{60}
`REJECT_AFTER_MESSAGES`	$2^{64} - 1$
`OLD_KEY_RETENTION`	5 seconds

Table 3: Rekeying timing constants.

During rekeying, both parties perform a fresh ephemeral DH exchange. After key rotation, nonce counters reset to zero and the epoch increments. Old keys are retained briefly (5 seconds) to decrypt in-flight packets, then securely zeroed.

4.5 Anti-Replay Protection

Each endpoint maintains a sliding window of at least 2048 received nonce values. The replay check occurs *before* AEAD verification to prevent CPU exhaustion attacks via replayed packets forcing expensive decryption operations.

4.6 Security Properties

Table 4 summarizes the security properties provided.

Property	Mechanism
Confidentiality	XChaCha20-Poly1305
Integrity	Poly1305 tag
Authenticity	Noise_IK mutual auth
Forward secrecy	Ephemeral DH + rekeying
Replay protection	Nonce window
Initiator identity hiding	Encrypted static key

Table 4: Security properties.

5 Transport Layer

5.1 Frame Format

Data frames carry encrypted sync messages:

Field	Size	Encrypted
Type	1 byte	No
Flags	1 byte	No
Session ID	6 bytes	No
Nonce Counter	8 bytes	No
Payload	variable	Yes
AEAD Tag	16 bytes	N/A

Table 5: Data frame format.

The 16-byte header (type through nonce counter) serves as additional authenticated data (AAD) for AEAD, preventing header tampering.

5.2 Connection Migration (Roaming)

NOMAD supports seamless IP address migration. When an authenticated frame arrives from a new source address:

1. Verify the AEAD tag with current session keys
2. If valid: update the remote endpoint to the new address
3. If invalid: silently drop (prevents spoofing)

No handshake is required. The session ID cryptographically binds the connection across IP changes. Anti-amplification limits (3× received bytes) prevent DDoS abuse.

5.3 RTT Estimation

Each frame carries timestamps for RTT measurement:

- **Timestamp**: Sender's time since session start (ms)
- **Timestamp Echo**: Most recent timestamp received from peer

RTT samples update smoothed RTT (SRTT) and variance using the RFC 6298 algorithm. The retransmission timeout (RTO) adapts accordingly, bounded between 100ms and 60 seconds.

Constant	Value
MIN_FRAME_INTERVAL	max(SRTT/2, 20ms)
COLLECTION_INTERVAL	8ms
DELAYED_ACK_TIMEOUT	100ms
MAX_FRAME_RATE	50 Hz

Table 6: Frame pacing constants.

5.4 Frame Pacing

To prevent congestion, frame transmission is paced:

The collection interval batches rapid state changes (e.g., fast typing) into single frames. Delayed acknowledgment allows acks to piggyback on data frames.

6 Synchronization Layer

6.1 State Interface

Any state type implementing the following interface can be synchronized:

```python
class SyncState(Protocol):
    STATE_TYPE_ID: str  # e.g., "nomad.terminal.v1"

    def diff(self, old: Self, new: Self) -> bytes:
        """Create idempotent diff."""

    def apply(self, state: Self, diff: bytes) -> Self:
        """Apply diff (must be idempotent)."""
```

The critical requirement is that diffs be *idempotent*: applying the same diff twice has no additional effect. This enables correct operation despite duplicates and reordering.

6.2 Sync Message Format

Field	Size
Sender State Number	8 bytes
Acked State Number	8 bytes
Base State Number	8 bytes
Diff Length	4 bytes
Diff Payload	variable

Table 7: Sync message format.

6.3 Convergence Algorithm

The receiver applies diffs based on version comparison:

```python
def receive_sync(msg):
    # Update ack tracking
    if msg.acked > last_acked:
        last_acked = msg.acked

    # Apply if newer (idempotent)
    if msg.sender_state_num > peer_state_num:
        peer_state = apply(peer_state, msg.diff)
        peer_state_num = msg.sender_state_num
```

The algorithm guarantees eventual consistency: regardless of packet loss, reordering, or duplication, states converge when any message gets through.

6.4 State Skipping

A consequence of state synchronization is that intermediate states may be lost if the network is slow. For terminal applications, this means fast output scrolls past without buffering. This is acceptable for interactive use but applications needing full history should use different transports.

7 Extensions

Extensions use Type-Length-Value (TLV) encoding and are negotiated during handshake. Defined extensions:

Compression (0x0001) zstd compression of diff payloads with configurable level (1–22). Small payloads may skip compression if it would increase size.

Scrollback (0x0002) Terminal-specific: synchronizes scrollback buffer for accessing previous output.

Prediction (0x0003) Terminal-specific: enables client-side keystroke prediction for perceived sub-10ms latency.

Future extensions include multiplexing and post-quantum key exchange (hybrid X25519+ML-KEM).

8 Evaluation

8.1 Conformance Test Suite

We developed a conformance test suite to validate protocol implementations. The suite is organized into phases:

Phase 1 (No Docker) Validates a Python reference codec against generated test vectors. Tests cover cryptographic primitives, handshake logic, and sync message encoding.

Phase 2 (Docker) Tests real implementations via containerization, including wire format compliance, adversarial inputs, and network resilience.

8.2 Test Categories

Category	**Tests**	**Docker**
Unit (reference codec)	80	No
Protocol (handshake, rekey)	107	No
Adversarial (replay, MITM)	36	No
Resilience (packet loss, jitter)	60+	Yes
Wire (byte-level format)	40+	Yes
Total	**1,000+**	

Table 8: Test suite coverage.

8.3 Test Vector Generation

Test vectors are generated using reference cryptographic libraries (Python `cryptography` for XChaCha20-Poly1305, `noiseprotocol` for Noise_IK). Generation is idempotent—running twice produces identical output—ensuring reproducibility. Vectors are stored in JSON5 format with inline comments explaining each field.

8.4 Property-Based Testing

We use Hypothesis [5] for property-based testing. Key properties verified:

- Idempotence: $a(s, d) = a(a(s, d), d)$ where a is apply
- Convergence: arbitrary orderings reach same state
- Replay rejection: repeated nonces detected
- Nonce uniqueness: no collisions within epoch

8.5 Reference Implementation

A Rust implementation of NOMAD (`nomad-protocol` crate) has passed the complete conformance suite including all unit, protocol, and adversarial tests. The implementation is available on crates.io.

9 Formal Verification

We formally verify NOMAD's security properties using ProVerif [1] for cryptographic protocol analysis and TLA+ [4] for state machine verification.

9.1 ProVerif Security Analysis

We model the handshake, replay protection, and rekeying mechanisms in ProVerif's applied pi-calculus. Table 9 summarizes the verified properties.

Property	**Result**
Session key secrecy	Verified
Mutual authentication	Verified
Frame authenticity	Verified
Replay protection	Verified
Forward secrecy	Verified
Post-compromise security	Verified*

Table 9: ProVerif verification results. *After fix.

9.1.1 PCS Vulnerability Discovery and Fix

ProVerif identified a vulnerability in the original rekeying mechanism: an attacker who compromises epoch n session keys can perform a man-in-the-middle attack during the epoch $n+1$ rekey, maintaining access indefinitely.

The attack exploits that rekey messages were authenticated only with the current (compromised) session key. The attacker intercepts the initiator's new ephemeral key, generates their own ephemeral, and completes the rekey with both parties—deriving the new session keys.

We fix this by deriving a `rekey_auth_key` from the static Diffie-Hellman secret during handshake:

$$k_{auth} = \text{HKDF}(\text{DH}(s_i, S_r), \text{“rekey auth”}) \quad (2)$$

New session keys mix this authentication key:

$$(k'_i, k'_r) = \text{HKDF}(\text{DH}(e_i, e_r) \| k_{auth}, n) \quad (3)$$

Since the attacker lacks the static private keys, they cannot compute k_{auth} and thus cannot derive valid epoch $n + 1$ keys. ProVerif confirms this fix restores post-compromise security.

9.2 TLA+ State Machine Verification

We model three protocol components in TLA+ and verify safety invariants using the TLC model checker. Table 10 shows the verification results.

Model	States	Invariants
Rekey state machine	2.8M	6
Sync layer	200K	6
Roaming	41K	6

Table 10: TLA+ model checking results.

Key invariants verified include: monotonic epoch numbers, nonce uniqueness within epochs, anti-amplification bounds during roaming, and eventual state convergence (given bounded message loss).

The formal models are available in the `formal/` directory of the specification repository.

10 Related Work

Table 11 compares Nomad with related protocols.

Mosh [8] pioneered UDP-based state synchronization for remote shells. Nomad inherits its core insight but updates the cryptographic primitives and adds forward secrecy via rekeying.

WireGuard [2] demonstrated that fixed cryptographic suites and minimal code simplify security analysis. Nomad follows this philosophy and borrows WireGuard's rekeying approach.

QUIC [3] provides reliable transport over UDP with integrated TLS 1.3. Unlike Nomad, QUIC provides ordered delivery and requires more complex connection state.

SSH [9] remains the standard for secure remote access but suffers from TCP's limitations on unreliable networks.

11 Conclusion

We presented Nomad, a secure state synchronization protocol combining modern cryptography with the proven SSP paradigm from Mosh. The protocol achieves mutual authentication in one RTT using the Noise_IK pattern, provides forward secrecy through periodic rekeying, supports seamless roaming across IP changes, and guarantees state convergence through idempotent diffs.

Our conformance test suite with over 1,000 tests and property-based verification provides confidence in implementation correctness. The layered design separates concerns cleanly, and the generic state interface enables applications beyond terminal emulation.

Future work includes formal verification of the security properties, performance benchmarking under various network conditions, and development of additional state types for collaborative applications.

Availability

The protocol specifications and conformance test suite are available as open source. The Rust reference implementation is published at https://crates.io/crates/nomad-protocol.

References

[1] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. In *Foundations and Trends in Privacy and Security*, volume 1, pages 1–135, 2016.

[2] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*, San Diego, CA, 2017. Internet Society.

[3] J. Iyengar and M. Thomson. Quic: A udp-based multiplexed and secure transport. RFC 9000, IETF, 2021.

[4] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[5] David R. MacIver. Hypothesis: Property-based testing for python. https://hypothesis.readthedocs.io/, 2023. Accessed: 2024-01-15.

[6] Y. Nir and A. Langley. Chacha20 and poly1305 for ietf protocols. RFC 8439, IETF, 2018.

[7] Trevor Perrin. The noise protocol framework. Specification, noiseprotocol.org, 2018. Revision 34.

[8] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 177–182, Boston, MA, 2012. USENIX Association.

Property	**NOMAD**	**Mosh**	**WireGuard**	**QUIC**
Transport	UDP	UDP	UDP	UDP
Handshake RTT	1	0 (SSH bootstrap)	1	1–2
AEAD	XChaCha20-Poly1305	AES-128-OCB	ChaCha20-Poly1305	AES-GCM/ChaCha20
Key Exchange	Noise_IK	N/A (SSH)	Noise_IK	TLS 1.3
Roaming	Yes	Yes	Yes	Yes (CID)
Forward Secrecy	Yes (2 min rekey)	No	Yes (2 min rekey)	Yes
State Sync	Yes	Yes	No	No
Cipher Negotiation	No	No	No	Yes

Table 11: Protocol comparison.

[9] T. Ylonen and C. Lonvick. The secure shell (ssh) transport layer protocol. RFC 4253, IETF, 2006.