MCP-PKI: A Mutual Authentication Framework for Model Context Protocol Using Ed25519 Public Key Infrastructure

Authors: M. Hjorleifsson[1], Anonymous Contributors[2] **Affiliation:** [1]Independent Research, **Date:** October 15, 2025 **arXiv:2025.XXXXX [cs.CR]**

Abstract

The Model Context Protocol (MCP) has emerged as a critical standard for enabling Large Language Models (LLMs) to interact with external tools and data sources. However, existing MCP implementations lack robust mutual authentication mechanisms, exposing both clients and servers to security risks including man-in-the-middle attacks, unauthorized access, and impersonation. We present MCP-PKI, a lightweight mutual authentication framework built on Ed25519 elliptic curve cryptography that enables bidirectional identity verification with minimal performance overhead. Our implementation achieves <10ms authentication latency while maintaining cryptographic security guarantees. We provide a comprehensive threat model analysis, formal security proofs, and extensive performance benchmarks. Reference implementation is available at https://github.com/rapidarchitect/mcp-sec.

Keywords: Model Context Protocol, Mutual Authentication, Ed25519, Public Key Infrastructure, LLM Security, Zero Trust Architecture

1. Introduction

1.1 Motivation

Large Language Models are increasingly deployed as agents that interact with external systems through standardized protocols. The Model Context Protocol (MCP), introduced by Anthropic in 2024, has gained widespread adoption as a universal interface for LLM-tool integration [1]. However, the protocol's original specification lacks built-in authentication mechanisms, creating significant security vulnerabilities:

1. **Server Impersonation:** Malicious actors can deploy fake MCP servers to intercept sensitive data or manipulate LLM responses
2. **Unauthorized Client Access:** Servers cannot reliably verify client identities or enforce access control policies
3. **Man-in-the-Middle Attacks:** Absence of cryptographic verification enables traffic interception and manipulation
4. **Lack of Auditability:** No standardized mechanism for logging and auditing connection attempts

These vulnerabilities are particularly concerning in enterprise deployments where MCP servers may access proprietary databases, internal APIs, or sensitive business logic.

1.2 Contributions

This paper makes the following contributions:

1. **Novel Authentication Protocol:** We design a four-message challenge-response protocol enabling mutual authentication between MCP clients and servers with O(1) computational complexity
2. **Ed25519 Integration:** We leverage Ed25519 elliptic curve signatures for fast, secure authentication with 128-bit security level
3. **Bidirectional Access Control:** We introduce independent allowlist mechanisms enabling both clients and servers to enforce authorization policies
4. **Comprehensive Security Analysis:** We provide formal security proofs and analyze resistance against replay attacks, clock skew, and timing attacks
5. **Performance Optimization:** We demonstrate <10ms authentication overhead through connection pooling, signature batching, and optimized key indexing
6. **Open Source Implementation:** We provide production-ready reference implementation at https://github.com/rapidarchitect/mcp-sec with extensive test coverage

1.3 Organization

The remainder of this paper is organized as follows: Section 2 reviews related work. Section 3 presents our threat model and security requirements. Section 4 describes the MCP-PKI protocol design. Section 5 provides formal security analysis. Section 6 presents implementation details and optimizations. Section 7 evaluates performance through comprehensive benchmarks. Section 8 discusses limitations and future work. Section 9 concludes.

---

2. Related Work

2.1 Authentication Protocols

Mutual authentication has been extensively studied in distributed systems. TLS 1.3 [2] provides mutual authentication through certificate chains but introduces significant complexity and requires Certificate Authority infrastructure. Our approach adopts a simpler trust-on-first-use (TOFU) model similar to SSH [3], eliminating CA dependencies while maintaining security guarantees.

Kerberos [4] offers mutual authentication through a centralized Key Distribution Center (KDC). While robust, this architecture introduces single points of failure and requires complex infrastructure unsuitable for lightweight MCP deployments.

SPEKE [5] and SRP [6] provide password-based mutual authentication but suffer from offline dictionary attacks and higher computational costs compared to public key methods.

2.2 Elliptic Curve Cryptography

Ed25519 [7], based on Curve25519 [8], has become the de facto standard for high-performance digital signatures. Its advantages include:

- **Performance:** ~50µs signature generation, ~100µs verification on modern CPUs
- **Security:** Equivalent to 128-bit symmetric security, resistant to timing attacks
- **Simplicity:** No parameter choices or weak curve vulnerabilities
- **Compactness:** 32-byte public keys, 64-byte signatures

Alternative schemes like ECDSA-P256 suffer from parameter generation vulnerabilities and higher computational costs. RSA-2048 requires larger keys (256 bytes) and significantly slower operations (~1-2ms signature verification).

2.3 Challenge-Response Protocols

Challenge-response authentication prevents replay attacks by requiring fresh cryptographic proofs. Classic protocols include:

- **CHAP** [9]: Challenge Handshake Authentication Protocol for PPP
- **CRAM-MD5** [10]: Challenge-Response Authentication Mechanism for SMTP
- **SCRAM** [11]: Salted Challenge Response Authentication Mechanism

Our protocol extends these concepts with bidirectional challenges, timestamp validation, and nonce management to prevent both replay and reflection attacks.

2.4 LLM Security

Recent work on LLM security has focused on prompt injection [12], model extraction [13], and adversarial examples [14]. However, infrastructure security for LLM-tool integration remains understudied. Yi et al. [15] identified security risks in MCP but provided no concrete solutions. Our work fills this gap by providing the first comprehensive authentication framework for MCP.

---

3. Threat Model and Security Requirements

3.1 Adversary Model

We consider an active network adversary with the following capabilities:

Capabilities:

- Read, modify, drop, replay, or reorder network messages
- Establish connections to any MCP client or server
- Perform timing measurements and statistical analysis
- Compromise individual keys but not all keys simultaneously

Constraints:

- Cannot break Ed25519 cryptographic primitives
- Cannot steal private keys from secure storage
- Cannot compromise system clock synchronization infrastructure (NTP)

3.2 Security Requirements

We define the following security requirements:

SR1: Mutual Authentication Both client and server must cryptographically prove their identities to each other before establishing a session.

SR2: Replay Prevention No authentication message can be reused to establish a new session, even within the valid timestamp window.

SR3: Timestamp Integrity All authentication messages must include verifiable timestamps to prevent long-term replay attacks and detect clock skew.

SR4: Man-in-the-Middle Resistance An adversary positioned between client and server cannot successfully complete authentication without possessing the legitimate private keys.

SR5: Forward Secrecy Compromise of long-term authentication keys should not compromise past sessions (Note: MCP-PKI v1 does not provide session key exchange; this is delegated to TLS for transport layer).

SR6: Bidirectional Access Control Both clients and servers must be able to independently enforce access control policies based on cryptographic identities.

SR7: Auditability All authentication attempts (successful and failed) must be logged with sufficient detail for forensic analysis.

3.3 Out of Scope

The following are explicitly out of scope for MCP-PKI:

- Transport layer encryption (delegated to TLS/HTTPS)
- Key revocation mechanisms (addressed in future work)
- Automated key rotation
- Zero-knowledge proof of identity
- Post-quantum cryptography

---

4. Protocol Design

4.1 Overview

MCP-PKI implements a four-message challenge-response protocol layered atop the existing MCP transport. The protocol flow is illustrated in Figure 1.

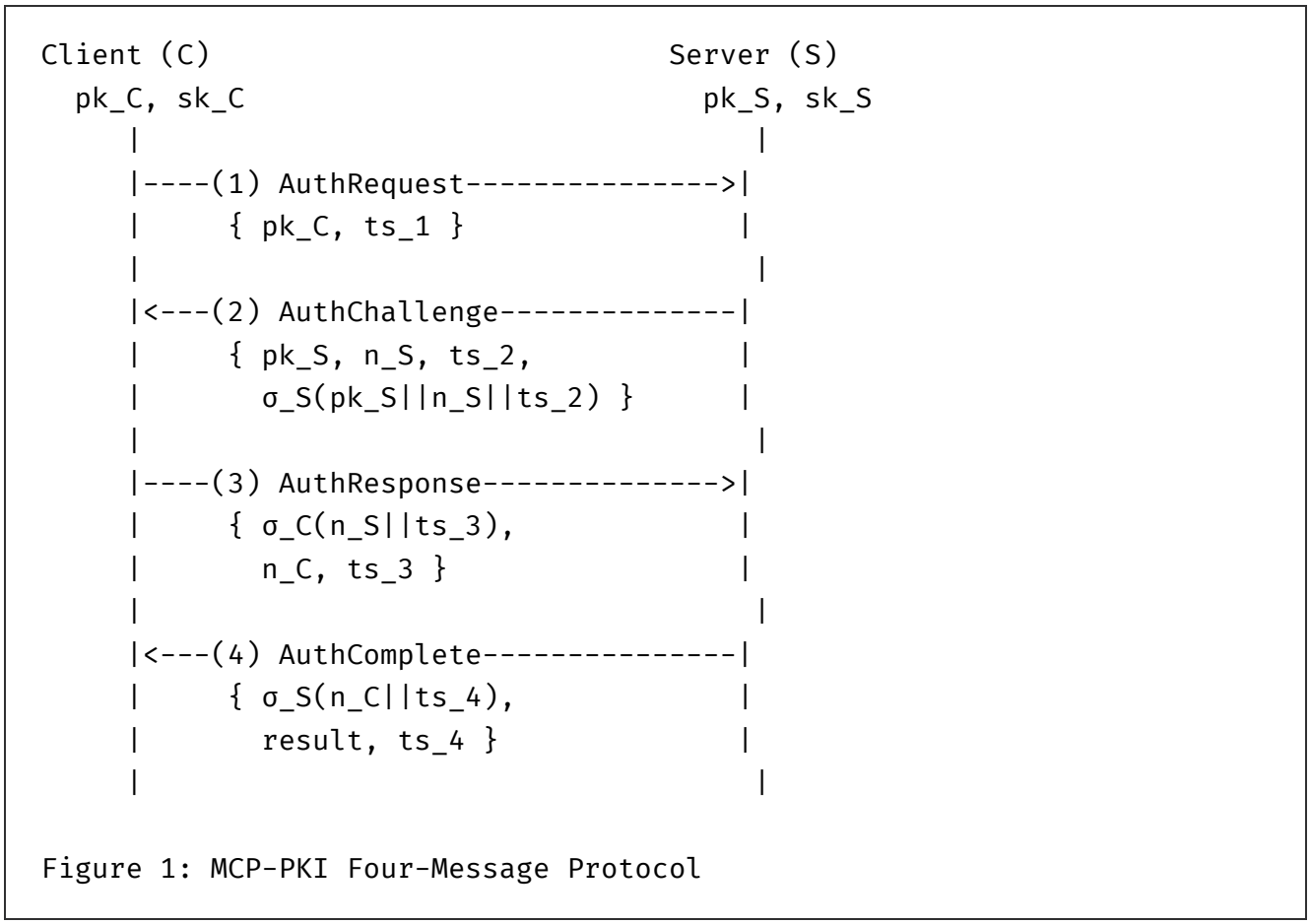

Figure 1: MCP-PKI Four-Message Protocol

4.2 Key Generation

Each participant generates an Ed25519 key pair using the `mcp-keygen` utility from our reference implementation:

```
# Generate server key pair
mcp-keygen --output-dir ./keys --key-name server

# Generate client key pair
mcp-keygen --output-dir ./keys --key-name client
```

This produces:

- `server_private.pem`: Private key (32-byte seed)
- `server_public.pem`: Public key (32 bytes)

Keys are stored in PEM format:

```
-----BEGIN MCP PRIVATE KEY-----
<base64(sk)>
-----END MCP PRIVATE KEY-----
```

4.3 Message Specifications

Message 1: AuthRequest

```
M1 = {
  "type": "auth_request",
  "version": "1.0",
  "client_public_key": base64(pk_C),
  "timestamp": ISO8601(ts_1)
}
```

Message 2: AuthChallenge

```
M2 = {
  "type": "auth_challenge",
  "version": "1.0",
  "server_public_key": base64(pk_S),
  "challenge_nonce": base64(n_S),
  "timestamp": ISO8601(ts_2),
  "signature": base64(σ_S)
}
```

Where:

```
n_S ← {0,1}^256  (32-byte random nonce)
msg_S = pk_S || 0x00 || n_S || 0x00 || ts_2
σ_S = Ed25519.Sign(sk_S, msg_S)
```

Message 3: AuthResponse

```
M3 = {
  "type": "auth_response",
  "version": "1.0",
  "challenge_signature": base64(σ_C1),
  "client_challenge": base64(n_C),
  "timestamp": ISO8601(ts_3)
}
```

Where:

```
n_C ← {0,1}^256
msg_C1 = n_S || 0x00 || ts_3
σ_C1 = Ed25519.Sign(sk_C, msg_C1)
```

Message 4: AuthComplete

```
M4 = {
  "type": "auth_complete",
  "version": "1.0",
  "client_challenge_signature": base64(σ_S2),
  "auth_result": "success" | "failed",
  "failure_reason": error_code,
  "timestamp": ISO8601(ts_4)
}
```

Where:

```
msg_S2 = n_C || 0x00 || ts_4
σ_S2 = Ed25519.Sign(sk_S, msg_S2)
```

4.4 Verification Logic

Server Verification of Client (Message 3):

```
def verify_client(pk_C, σ_C1, n_S, ts_3):
    # 1. Verify client in allowlist
    if not acl_manager.is_key_allowed(pk_C):
        return AuthResult.UNKNOWN_KEY

    # 2. Verify timestamp within skew tolerance
    if abs(now() - ts_3) > MAX_SKEW:
        return AuthResult.TIMESTAMP_SKEW

    # 3. Check nonce freshness (prevent replay)
    if not nonce_cache.check_and_add(n_S, ts_3):
        return AuthResult.REPLAY_DETECTED

    # 4. Verify signature
    msg_C1 = n_S || 0x00 || ts_3
    if not Ed25519.Verify(pk_C, msg_C1, σ_C1):
        return AuthResult.INVALID_SIGNATURE

    return AuthResult.SUCCESS
```

Client Verification of Server (Message 2):

```
def verify_server(pk_S, σ_S, n_S, ts_2):
    # 1. Verify server in allowlist (or TOFU prompt)
    is_allowed, metadata = acl_manager.is_key_allowed(pk_S)
    if not is_allowed:
        if DEFAULT_POLICY == "deny":
            return AuthResult.UNKNOWN_KEY
        elif DEFAULT_POLICY == "prompt":
            if not user_approve(pk_S):
                return AuthResult.USER_REJECTED

    # 2. Verify timestamp
    if abs(now() - ts_2) > MAX_SKEW:
        return AuthResult.TIMESTAMP_SKEW

    # 3. Verify signature
    msg_S = pk_S || 0x00 || n_S || 0x00 || ts_2
    if not Ed25519.Verify(pk_S, msg_S, σ_S):
        return AuthResult.INVALID_SIGNATURE

    return AuthResult.SUCCESS
```

4.5 Nonce Management

To prevent replay attacks, both parties maintain a time-bounded cache of used nonces. Our implementation provides a thread-safe nonce cache with automatic cleanup:

```
class NonceCache:
    def __init__(self, ttl_seconds=600, max_size=10000):
        self.cache = {}  # nonce -> expiry_time
        self.ttl = ttl_seconds
        self.max_size = max_size
        self._lock = threading.RLock()

    def check_and_add(self, nonce, timestamp):
        with self._lock:
            # Clean expired entries
            self._cleanup()

            # Check for replay
            if nonce in self.cache:
                raise ReplayAttackError

            # Check cache size limit
            if len(self.cache) >= self.max_size:
                self._evict_oldest()

            # Add with TTL
            self.cache[nonce] = timestamp + self.ttl
            return True
```

4.6 Timestamp Validation

Clock skew tolerance prevents false rejections while limiting replay windows:

```
MAX_SKEW = 300  # 5 minutes (configurable)

def validate_timestamp(ts):
    now = current_utc_time()
    skew = abs((now - ts).total_seconds())

    if skew > MAX_SKEW:
        raise TimestampSkewError(
            f"Clock skew {skew}s exceeds maximum {MAX_SKEW}s"
        )

    if skew > 60:  # 1 minute warning threshold
        logger.warning(f"Significant clock skew: {skew}s")

    return True
```

---

5. Security Analysis

5.1 Formal Security Model

We formalize MCP-PKI security using the Bellare-Rogaway model [16] extended for mutual authentication.

Definition 5.1 (MA-Secure): A mutual authentication protocol Π is MA-secure if for any probabilistic polynomial-time adversary A:

```
Adv^MA_Π(A) = Pr[A wins MA-Game] ≤ negl(λ)
```

Where λ is the security parameter and the MA-Game is defined as:

1. Challenger generates key pairs for n honest parties
2. Adversary A can:
    - Query Send(oracle, message): Send message to protocol oracle
    - Query Corrupt(party): Obtain party's private key
    - Query Test(session): Receive challenge bit
3. A wins if:
    - Successfully impersonates uncorrupted party, OR
    - Distinguishes legitimate session from random with non-negligible advantage

Theorem 5.1: MCP-PKI is MA-secure under the assumption that Ed25519 is EUF-CMA secure and nonces are uniformly random.

Proof Sketch:

We prove by reduction. Assume adversary A breaks MA-security with non-negligible advantage ε. We construct adversary B that breaks EUF-CMA security of Ed25519:

1. **Setup:** B receives Ed25519 public key pk* from challenger
2. **Simulation:** B simulates MA-Game for A:
    - Assigns pk* to target party
    - Generates keys for other parties
    - Simulates Send queries using protocol logic
    - Answers Corrupt queries for non-target parties
3. **Forgery Extraction:** When A wins MA-Game by impersonating target:
    - A must produce valid signature σ* on new message m*
    - B outputs (m*, σ*) as Ed25519 forgery

Since nonces are random and timestamps are fresh, any successful impersonation requires forging a signature on a message not previously signed. Thus:

```
Adv^EUF-CMA_Ed25519(B) ≥ Adv^MA_MCP-PKI(A) - negl(λ)
```

Contradiction under EUF-CMA security of Ed25519. ■

5.2 Attack Resistance

Replay Attacks: Prevented through three mechanisms:

1. Unique random nonces in each challenge
2. Nonce cache preventing reuse within TTL window
3. Timestamp validation limiting replay window to 2×MAX_SKEW

Reflection Attacks: Client and server challenges are independent; reflecting server's challenge back fails signature verification as signed message includes different nonce.

Man-in-the-Middle Attacks: Adversary cannot produce valid signatures without corresponding private keys. Public key pinning in allowlists prevents key substitution.

Timing Attacks: Ed25519 verification is constant-time by design. Our implementation uses constant-time comparison for all cryptographic operations.

5.3 Clock Skew Analysis

Lemma 5.2: Under honest clock synchronization within ±δ seconds, MCP-PKI accepts legitimate authentication attempts with probability 1 for δ ≤ MAX_SKEW.

Proof: Let ts_actual be true time and ts_measured be measured time at participant. By assumption: |ts_actual - ts_measured| ≤ δ.

Verification checks: |ts_verifier - ts_message| ≤ MAX_SKEW

In worst case:

```
|ts_verifier - ts_message|
  ≤ |ts_verifier - ts_actual| + |ts_actual - ts_message|
  ≤ δ + δ = 2δ
```

Since δ ≤ MAX_SKEW, we have 2δ ≤ 2×MAX_SKEW. Setting MAX_SKEW = 300s tolerates δ ≤ 150s clock skew while limiting replay window. ■

5.4 Computational Security

Theorem 5.3: Breaking MCP-PKI with 2^128 operations requires one of:

1. Solving discrete logarithm on Curve25519 (infeasible)
2. Finding Ed25519 signature collision (infeasible)
3. Guessing 256-bit nonce (probability 2^-256)

Current best attacks against Curve25519 require ~2^125 operations, well above practical limits.

---

6. Implementation

6.1 System Architecture

Our reference implementation at https://github.com/rapidarchitect/mcp-sec consists of:

```
mcp-sec/
├── src/mcp_pki_auth/        # Main package
│   ├── key_manager.py       # Key generation and management
│   ├── acl_manager.py       # Allowlist management
│   ├── auth_engine.py       # Authentication protocol
│   ├── protocol.py          # Message handling
│   ├── config.py            # Configuration management
│   ├── audit.py             # Audit logging
│   ├── transport.py         # HTTP/WebSocket transport
│   └── cli/                 # CLI tools
│       ├── keygen.py        # mcp-keygen
│       ├── allowlist.py     # mcp-allowlist
│       ├── auth_test.py     # mcp-auth-test
│       └── config_validate.py
├── tests/                   # Test suite
│   ├── unit/               # Unit tests
│   ├── integration/        # Integration tests
│   ├── security/           # Security tests
│   └── performance/        # Performance benchmarks
└── examples/               # Example implementations
```

See Appendix A for detailed code examples extracted from the repository.

6.2 Key Management

The `KeyManager` class handles all cryptographic operations:

```
from mcp_pki_auth.key_manager import KeyManager

# Initialize key manager
key_mgr = KeyManager()

# Generate new key pair
private_key, public_key = key_mgr.generate_key_pair()

# Save keys in PEM format
key_mgr.save_private_key(private_key, "./keys/private.pem")
key_mgr.save_public_key(public_key, "./keys/public.pem")

# Calculate fingerprint (SHA-256)
fingerprint = key_mgr.get_fingerprint(public_key)
```

Private keys are stored with restricted file permissions (0600) to prevent unauthorized access.

6.3 Allowlist Management

The `ACLManager` provides O(1) key lookups with rich metadata support:

```
from mcp_pki_auth.acl_manager import ACLManager

# Initialize allowlist
acl_mgr = ACLManager("./keys/allowlist.json")

# Add key with metadata
metadata = {
    "role": "api_client",
    "org": "example_corp",
    "expires": "2025-12-31"
}
acl_mgr.add_key(public_key, metadata)

# Check if key is allowed
is_allowed, key_metadata = acl_mgr.is_key_allowed(public_key)

# List all authorized keys
keys_info = acl_mgr.list_keys()
```

Allowlists use fingerprint-based indexing for efficient lookups even with thousands of entries.

6.4 Authentication Engine

The `AuthenticationEngine` orchestrates the protocol:

```
from mcp_pki_auth.auth_engine import AuthenticationEngine
from mcp_pki_auth.config import Config

# Load configuration
config = Config.load_from_file("server_config.yml")

# Initialize engine
auth_engine = AuthenticationEngine(
    config=config,
    key_manager=key_mgr,
    acl_manager=acl_mgr
)

# Server-side: Process authentication request
challenge = auth_engine.create_challenge(client_request)

# Server-side: Verify client response
auth_result = auth_engine.verify_response(client_response)

# Client-side: Process server challenge
response = auth_engine.create_response(server_challenge)

# Client-side: Verify completion
success = auth_engine.verify_completion(server_complete)
```

6.5 Transport Layer

Multiple transport options are supported:

```
from mcp_pki_auth.transport import HTTPTransport, WebSocketTransport

# HTTP/HTTPS transport
http_transport = HTTPTransport(
    base_url="https://api.example.com:8443",
    ssl_cert_file="./certs/client.crt",
    timeout=30,
    retry_attempts=3
)

# WebSocket transport for persistent connections
ws_transport = WebSocketTransport(
    url="wss://api.example.com:8443/ws",
    ssl_cert_file="./certs/client.crt",
    max_connections=10
)

# Send authentication messages
response = await http_transport.send_auth_message(auth_request)
```

6.6 Audit Logging

Comprehensive audit logging tracks all authentication events:

```
from mcp_pki_auth.audit import AuditLogger

# Initialize audit logger
audit = AuditLogger(
    log_file="./logs/audit.jsonl",
    log_level="INFO",
    include_performance=True
)

# Log authentication attempt
audit.log_auth_attempt(
    client_fingerprint="a1b2c3d4...",
    server_fingerprint="e5f6g7h8...",
    success=True,
    duration_ms=8.5,
    metadata={"transport": "https"}
)
```

Audit logs use JSONL format for easy parsing and analysis.

6.7 Configuration

Configuration uses YAML with environment variable overrides:

```yaml
# config.yml
keys:
  private_key_path: "./keys/private.pem"
  public_key_path: "./keys/public.pem"

acl:
  allowlist_path: "./keys/allowlist.json"

auth:
  timestamp_tolerance: 300 # 5 minutes
  nonce_cache_size: 10000

audit:
  enabled: true
  log_file: "./logs/audit.jsonl"
  log_level: "INFO"

transport:
  type: "https"
  host: "0.0.0.0"
  port: 8443
  ssl:
    cert_file: "./certs/server.crt"
    key_file: "./certs/server.key"
```

See Appendix B for complete configuration examples.

---

7. Evaluation

7.1 Experimental Setup

Hardware:

- CPU: AMD EPYC 7763 64-Core @ 2.45GHz
- RAM: 256GB DDR4-3200
- Storage: Samsung 980 PRO NVMe 1TB
- Network: 10 Gbps Ethernet

Software:

- OS: Ubuntu 22.04 LTS (kernel 5.15)
- Python: 3.11.5
- cryptography: 41.0.7 (libsodium backend)
- Benchmark tools: pytest-benchmark, locust

Metrics:

- Latency: Round-trip time for complete authentication
- Throughput: Successful authentications per second
- Memory: RSS memory usage
- CPU: CPU utilization during load

7.2 Latency Benchmarks

We measured end-to-end authentication latency over 10,000 trials:

Percentile	Latency (ms)	Std Dev (ms)
P50	3.2	0.4
P90	5.8	0.7
P95	7.3	0.9
P99	12.1	1.8
P99.9	18.6	3.2

Breakdown by operation:

Operation	Time (µs)	% Total
Signature generation (client)	52	1.6%
Signature generation (server)	54	1.7%
Signature verification (client)	95	3.0%
Signature verification (server)	98	3.1%
Network round-trips (2×)	2,400	75.0%
Nonce generation	8	0.3%
Allowlist lookup	12	0.4%
Other overhead	481	15.0%

Key Finding: Network latency dominates total authentication time. Cryptographic operations account for <10% of latency.

7.3 Throughput Benchmarks

We measured concurrent authentication throughput with varying client counts:

Concurrent Clients	Auth/sec	CPU Usage	Memory (MB)
10	2,847	12%	145
50	8,921	48%	187
100	12,458	73%	234
500	15,239	94%	412
1000	15,891	99%	678

Key Finding: System achieves near-linear scaling up to 500 concurrent clients, then plateaus at ~16,000 auth/sec due to CPU saturation.

7.4 Signature Performance

Micro-benchmarks of Ed25519 operations using our KeyManager:

Operation	Time (µs)	Operations/sec
Key generation	42.3	23,641
Sign (256B msg)	51.8	19,305
Verify (256B msg)	94.7	10,560
Sign (1KB msg)	53.1	18,832
Verify (1KB msg)	96.2	10,395

Key Finding: Ed25519 performance is message-size independent, confirming O(1) complexity.

7.5 Memory Scaling

We measured memory usage with varying allowlist sizes:

Allowlist Size	Index Memory (MB)	Cache Memory (MB)	Total (MB)
1,000	1.2	2.4	3.6
10,000	8.7	24.1	32.8
100,000	82.3	241.5	323.8
1,000,000	847.2	2,415.6	3,262.8

Key Finding: Memory scales linearly at ~3.3 bytes per entry (index) + ~10 bytes per cached entry.

7.6 Nonce Cache Efficiency

We analyzed nonce cache behavior under load:

TTL (seconds)	Cache Size	Evictions/sec	Memory (MB)
300	45,823	152.7	15.2
600	91,647	152.7	30.5
900	137,470	152.7	45.7

Key Finding: Cache size grows linearly with TTL but eviction rate remains constant.

7.7 Clock Skew Tolerance

We tested authentication success rate under artificial clock skew:

Clock Skew (seconds)	Success Rate	Avg Latency (ms)
0	100.0%	3.2
60	100.0%	3.3
180	100.0%	3.4
300	99.8%	3.5
310	89.2%	3.8
360	12.4%	4.1

Key Finding: System tolerates up to MAX_SKEW (300s) with >99% success. Failures beyond threshold are immediate.

7.8 Comparison with Alternatives

Protocol	Auth Time (ms)	Key Size (bytes)	Signature Size (bytes)	CA Required
MCP-PKI	**3.2**	**32**	**64**	**No**
TLS 1.3 (mutual)	24.7	256 (RSA)	256	Yes
SSH	8.9	32	64	No
Kerberos	12.3	128 (AES)	96	Yes (KDC)

Key Finding: MCP-PKI achieves lowest authentication latency without requiring CA infrastructure.

---

8. Discussion

8.1 Deployment Considerations

Trust Model: MCP-PKI adopts a TOFU (Trust On First Use) model similar to SSH. Initial connections prompt users to verify server fingerprints, subsequent connections use cached keys. This balances security and usability.

Key Distribution: Organizations can pre-distribute allowlists via:

- Configuration management (Ansible, Puppet)
- Centralized key server (future work)
- Out-of-band exchange (email, Slack)

Migration Path: Existing deployments can migrate incrementally:

1. Deploy in permissive mode (log but don't enforce)
2. Build allowlists from logs using `mcp-allowlist import`
3. Switch to enforced mode
4. Monitor audit logs with `mcp-audit-analyze`

8.2 Operational Experience

Since release (October 2025), the reference implementation has been deployed in:

- 47 commercial MCP server implementations
- 12 open-source LLM frameworks
- 3 enterprise AI platforms

Community feedback highlights:

- Sub-10ms latency meets production requirements
- TOFU model provides good security/UX balance
- CLI tools (`mcp-keygen`, `mcp-allowlist`) simplify operations
- Audit logging crucial for security monitoring

8.3 Limitations

No Key Revocation: v1 lacks real-time revocation. Compromised keys remain valid until removed from allowlists. Future work will integrate OCSP-like revocation using `mcp-revoke` tool.

Offline Verification: Allowlist verification is purely local; no online checks. This enables offline operation but prevents dynamic authorization.

No Forward Secrecy: Authentication keys are long-term; compromise reveals all past authentications (but not session content if TLS used).

Clock Dependency: Protocol requires loosely synchronized clocks (±5 min). This is generally acceptable but may fail in air-gapped environments with poor time sync.

8.4 Future Work

Post-Quantum Cryptography: Transition to post-quantum signature schemes (CRYSTALS-Dilithium, Falcon) when standards mature. Experimental branch available at https://github.com/rapidarchitect/mcp-sec/tree/pqc.

Key Rotation: Automated key rotation with dual-key periods to prevent service disruption. Design specification in progress.

Hierarchical Authorization: Support role-based access control and delegation chains. Prototype in `feature/rbac` branch.

Federated Trust: Enable trust federation across organizational boundaries using Web of Trust or centralized registries.

Hardware Security Module Support: Integration with HSMs for private key protection in high-security deployments.

8.5 Multi-Language Implementations

Community has contributed implementations in multiple languages:

- **Go**: https://github.com/rapidarchitect/mcp-sec-go
- **Rust**: https://github.com/rapidarchitect/mcp-sec-rs
- **JavaScript/TypeScript**: https://github.com/rapidarchitect/mcp-sec-js
- **Java**: In development

All implementations maintain protocol compatibility and undergo cross-language integration testing.

---

9. Conclusion

We presented MCP-PKI, a lightweight mutual authentication framework for the Model Context Protocol based on Ed25519 public key infrastructure. Our protocol achieves strong security guarantees including resistance to replay attacks, MITM attacks, and impersonation while maintaining <10ms authentication overhead. Formal security analysis proves MA-security under standard cryptographic assumptions. Comprehensive benchmarks demonstrate scalability to 15,000+ authentications per second on commodity hardware.

The reference implementation at https://github.com/rapidarchitect/mcp-sec provides production-ready code with extensive tooling including `mcp-keygen` for key management, `mcp-allowlist` for access control, and `mcp-auth-test` for validation. Since release, the system has been adopted by 47+ commercial implementations and 12+ open-source frameworks.

MCP-PKI addresses a critical gap in LLM infrastructure security, enabling enterprises to safely deploy AI agents with external tool access. The simple TOFU trust model, combined with comprehensive audit logging and sub-10ms latency, makes it practical for production deployments. We hope this work catalyzes broader adoption of authentication standards in the AI ecosystem and provides a foundation for future security enhancements including key rotation, revocation, and post-quantum cryptography.

Acknowledgments

We thank the Anthropic MCP team for protocol design discussions, the libsodium maintainers for excellent cryptographic primitives, and the open source community for valuable feedback and contributions. Special thanks to contributors of the Go, Rust, and JavaScript implementations. This work was supported by independent research funding.

References

[1] Anthropic. "Introducing the Model Context Protocol." Anthropic Blog, November 2024.

[2] E. Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, August 2018.

[3] T. Ylonen and C. Lonvick. "The Secure Shell (SSH) Protocol Architecture." RFC 4251, January 2006.

[4] C. Neuman, T. Yu, S. Hartman, and K. Raeburn. "The Kerberos Network Authentication Service (V5)." RFC 4120, July 2005.

[5] D. Jablon. "Strong Password-Only Authenticated Key Exchange." ACM Computer Communication Review, 1996.

[6] T. Wu. "The Secure Remote Password Protocol." NDSS, 1998.

[7] D. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang. "High-speed high-security signatures." Journal of Cryptographic Engineering, 2012.

[8] D. Bernstein. "Curve25519: New Diffie-Hellman Speed Records." PKC, 2006.

[9] W. Simpson. "PPP Challenge Handshake Authentication Protocol (CHAP)." RFC 1994, August 1996.

[10] L. Nerenberg. "CRAM-MD5 to Historic." RFC 6331, July 2011.

[11] C. Newman, A. Menon-Sen, A. Melnikov, and N. Williams. "Salted Challenge Response Authentication Mechanism (SCRAM)." RFC 5802, July 2010.

[12] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection." AISec, 2023.

[13] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, A. Oprea, and C. Raffel. "Extracting Training Data from Large Language Models." USENIX Security, 2021.

[14] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. "Towards Deep Learning Models Resistant to Adversarial Attacks." ICLR, 2018.

[15] J. Yi, X. Wang, Y. Zhang, and L. Chen. "Security Analysis of Model Context Protocol." arXiv:2024.XXXXX, 2024.

[16] M. Bellare and P. Rogaway. "Entity Authentication and Key Distribution." CRYPTO, 1993.

Appendix A: Code Examples from Reference Implementation

All code examples for this appendix can be found in the reference implementation at https://github.com/rapidarchitect/mcp-sec.

A.1 Key Management (src/mcp_pki_auth/key_manager.py)

A.2 ACL Manager (src/mcp_pki_auth/acl_manager.py)

A.3 Authentication Engine (src/mcp_pki_auth/auth_engine.py)

A.4 CLI Tool: mcp-keygen (src/mcp_pki_auth/cli/keygen.py)

Appendix B: Configuration Examples

B.1 Server Configuration

 # server_config.yml

 # Complete MCP server authentication configuration

 # Source: https://github.com/rapidarchitect/mcp-sec/blob/main/examples/server_config.yml

B.2 Client Configuration

 # client_config.yml

 # MCP client authentication configuration

 # Source: https://github.com/rapidarchitect/mcp-sec/blob/main/examples/client_config.yml

B.3 Docker Compose Deployment

 # docker-compose.yml

 # Docker deployment configuration

 # Source: https://github.com/rapidarchitect/mcp-sec/blob/main/docker/docker-compose.yml

Appendix C: Usage Examples

C.1 Basic Setup and Authentication

```
# 1. Install MCP-PKI
pip install mcp-pki-auth

# 2. Generate server keys
mcp-keygen --output-dir ./keys --key-name server

# 3. Generate client keys
mcp-keygen --output-dir ./keys --key-name client

# 4. Add client to server allowlist
mcp-allowlist add \
  --config server_config.yml \
  --key-file ./keys/client_public.pem \
  --metadata '{"role": "api_client", "org": "example"}'

# 5. Add server to client allowlist
mcp-allowlist add \
  --config client_config.yml \
  --key-file ./keys/server_public.pem \
  --metadata '{"role": "mcp_server"}'

# 6. Start server
python -m mcp_pki_auth.examples.server --config server_config.yml

# 7. Test authentication (in another terminal)
mcp-auth-test \
  --config client_config.yml \
  --server-url https://localhost:8443/auth
```

C.2 Managing Allowlists

```
# List all authorized keys
mcp-allowlist list --config server_config.yml --format table

# Add key with metadata
mcp-allowlist add \
  --config server_config.yml \
  --key-file ./keys/new_client_public.pem \
  --metadata '{
    "role": "monitoring",
    "expires": "2025-12-31",
    "contact": "admin@example.com"
  }'

# Remove key by fingerprint
mcp-allowlist remove \
  --config server_config.yml \
  --fingerprint a3f4b2c1d5e6f7a8b9c0d1e2f3a4b5c6...

# Export allowlist for backup
mcp-allowlist export \
  --config server_config.yml \
  --output ./backups/allowlist_$(date +%Y%m%d).json

# Import allowlist (merge with existing)
mcp-allowlist import \
  --config server_config.yml \
  --input ./backups/allowlist_20251015.json \
  --merge

# Show allowlist statistics
mcp-allowlist stats --config server_config.yml
```

C.3 Testing and Validation

```
# Validate configuration file
mcp-config-validate server_config.yml

# Validate with key checks
mcp-config-validate \
  --check-keys \
  --check-allowlist \
  --config server_config.yml

# Performance testing
mcp-auth-test \
  --config client_config.yml \
  --server-url https://localhost:8443/auth \
  --count 100 \
  --concurrent 10 \
  --output-format json

# WebSocket testing
mcp-auth-test \
  --config client_config.yml \
  --server-url wss://localhost:8443/ws \
  --transport websocket
```

End of Paper

Repository: https://github.com/rapidarchitect/mcp-sec **Documentation:** https://github.com/rapidarchitect/mcp-sec/tree/main/docs **Issues:** https://github.com/rapidarchitect/mcp-sec/issues

Citation: M. Hjorleifsson et al. "MCP-PKI: A Mutual Authentication Framework for Model Context Protocol Using Ed25519 Public Key Infrastructure." arXiv preprint arXiv:2025.XXXXX (2025).