

# RFC-007: Threshold Signature Governance

**Status:** Implemented **Date:** January 2026 **Author:** Derrell Piper ddp@eludom.net **Implementation:** script.scm, cyberspace.scm

---

## Abstract

This RFC specifies the threshold signature system for Cyberspace governance, enabling K-of-N multi-party authorization for critical operations. Democracy in code: no single point of failure, no rogue administrator.

---

## Motivation

Critical operations require collective authorization:

- **Release signing:** Multiple maintainers must approve
- **Deployment:** Operations team quorum required
- **Key ceremonies:** Distributed trust for root keys
- **Emergency response:** Prevent unilateral action

Traditional approaches fail: – **Shared passwords:** Who has it? Who used it? – **Sudo access:** Root is root – **Approval workflows:** Soft controls, bypassable

Threshold signatures provide cryptographic enforcement:

**K valid signatures required. Not K-1. Not bypass. Mathematics.**

---

## Specification

### Tiered Signing Model

#### SIGNING TIERS

Development	1-of-1	Single developer can iterate
Staging	2-of-2	Peer review required
Production	3-of-5	Governance council quorum
Emergency	5-of-7	Full council for critical ops

## Script Signature Record

```
(define-record-type <script-signature>
  (make-script-signature signer signature timestamp)
  script-signature?
  (signer signature-signer)      ; Ed25519 public key (32 bytes)
  (signature signature-value)    ; Ed25519 signature (64 bytes)
  (timestamp signature-timestamp) ; Unix epoch seconds
```

## Signing a Script

```
(define (sign-script script-content private-key #!optional public-key)
  "Sign script content with a private key"
  ...)
```

**Process:** 1. Convert content to blob if string 2. Derive public key from private (if not provided) 3. Sign content with Ed25519 4. Record timestamp 5. Return script-signature record

## Verifying Single Signature

```
(define (verify-script script-content signature-record)
  "Verify a script signature"
  (ed25519-verify (signature-signer signature-record)
                  content-blob
                  (signature-value signature-record)))
```

## Threshold Verification

```
(define (verify-threshold-script script-content signature-records threshold)
  "Verify threshold signatures on a script
  Returns: #t if at least K signatures are valid"
  (let* ((valid-sigs (filter (lambda (sig)
                              (verify-script script-content sig))
                              signature-records))
         (valid-count (length valid-sigs)))
    (>= valid-count threshold)))
```

---

## Signature File Format

```
;; deploy.sig
((signature "hex-signature" "hex-pubkey" 1767685100)
 (signature "hex-signature" "hex-pubkey" 1767685200)
 (signature "hex-signature" "hex-pubkey" 1767685300))
```

Each entry contains: - Signature bytes (hex-encoded) -  
Signer public key (hex-encoded) - Timestamp (Unix epoch)

---

## CLI Interface

### cyberspace verify

```
$ cyberspace verify deploy.scm deploy.sig \
  --threshold 3 \
  --keys alice.pub bob.pub carol.pub dave.pub eve.pub
```

=== Cyberspace Script Verification ===

```
Script:      deploy.scm
Signatures:  deploy.sig
Threshold:   3
Keys:        5 provided
```

```
Found 3 signature(s) in file
  Signature 1:  VALID (signer: cbc9b260da65f6a7...)
  Signature 2:  VALID (signer: a5f8c9e3d2b1f0e4...)
  Signature 3:  VALID (signer: 7d3e8b2c1a0f5e4d...)
```

SUCCESS: Script verified with 3-of-3 threshold

### cyberspace run

```
$ cyberspace run deploy.scm deploy.sig \
  --threshold 3 \
  --keys alice.pub bob.pub carol.pub dave.pub eve.pub
```

=== Cyberspace Script Verification ===

```
...
  SUCCESS: Script verified with 3-of-5 threshold
```

```
=== Executing Script ===
(seal-release "2.0.0")
```

```
message: "Major release with new governance model"
preserve: #t)
```

---

## Governance Scenarios

### Production Deployment (3-of-5)

```
;; Governance Council: Alice, Bob, Carol, Dave, Eve
;; Threshold: 3 signatures required

(define deployment-script
  "(seal-release \"2.0.0\"
    message: \"Major release\"
    preserve: #t)")

;; Alice signs
(define sig-alice
  (sign-script deployment-script alice-private alice-public))

;; Carol signs
(define sig-carol
  (sign-script deployment-script carol-private carol-public))

;; Dave signs
(define sig-dave
  (sign-script deployment-script dave-private dave-public))

;; Verify threshold
(verify-threshold-script deployment-script
  (list sig-alice sig-carol sig-dave)
  3)

;; => #t
```

### Insufficient Signatures

```
;; Only 2 signatures
(define insufficient-sigs
  (list sig-alice sig-carol))

(verify-threshold-script deployment-script insufficient-sigs 3)
;; => #f (rejected: need 3, got 2)
```

## Tampered Script Detection

```
;; Attacker modifies script
(define tampered-script
  "(seal-release \"2.0.0\"
    message: \"HACKED - deploying malware\"
    preserve: #t)")

(verify-threshold-script tampered-script sufficient-sigs 3)
;; => #f (signatures don't match modified content)
```

---

## Multi-Signature vs Shamir

Two threshold approaches exist:

### Multi-Signature (This RFC)

Each party: own keypair  
Signing: each signs independently  
Verify: count valid signatures  $K$   
Use case: governance, approvals

**Advantages:** – Each party maintains own key – Clear audit trail (who signed) – Simple revocation (by key) – No key reconstruction

### Shamir Splitting (RFC-008)

Single key: split into  $N$  shares  
Signing:  $K$  parties reconstruct, sign once  
Verify: single signature  
Use case: key backup, recovery

**Advantages:** – Single signature output – Key never fully assembled (in advanced schemes) – Smaller signature files

For governance, multi-signature is preferred: – Accountability (which individuals approved) – No reconstruction ceremony – Works asynchronously

---

## Security Considerations

### Threat Model

**Protected against:** – Single compromised key (need K) – Rogue administrator (need quorum) – Script tampering (signatures fail) – Replay of old scripts (timestamps, context)

**Not protected against:** – K compromised keys – All parties colluding – Side-channel on signing devices

### Key Management

1. **Generation:** Secure random via libsodium
2. **Storage:** Hardware tokens preferred, encrypted files acceptable
3. **Distribution:** Out-of-band verification of public keys
4. **Rotation:** New ceremony, revoke old keys

### Threshold Selection

Scenario	Threshold	Rationale
Development	1-of-1	Fast iteration
Staging	2-of-2	Peer review
Production	3-of-5	Majority quorum
Root key	5-of-7	Supermajority
Emergency	N-of-N	Full consensus

## Audit Integration

Every verified execution records:

```
(audit-append
  actor: (threshold-verifier-list)
  action: (list 'script-execute script-hash)
  motivation: "Production deployment authorized"
  context: (list 'threshold 3 'signatures 3))
```

Audit trail shows: – Which signers authorized – What script was executed – When authorization occurred – Threshold requirements met

## Implementation Notes

### Dependencies

- `crypto-ffi` – Ed25519 operations
- `cert` – SPKI integration
- `audit` – Trail recording

### Performance

- Signature verification:  $\sim 10\mu\text{s}$  per Ed25519 verify
  - Threshold check:  $O(N)$  where  $N$  = signature count
  - No network round-trips (offline verification)
- 

## References

1. Boneh, D., et al. (2001). Short Signatures from the Weil Pairing.
  2. Gennaro, R., et al. (2016). Threshold-optimal DSA/ECDSA signatures.
  3. NIST SP 800-57. Recommendation for Key Management.
  4. RFC-004: SPKI Authorization Integration
- 

## Changelog

- **2026-01-06** – Initial specification
- 

**Implementation Status:** Complete **Test Status:** Passing  
(`test-threshold-sig.scm`) **CLI Tool:** `cyberspace verify/run`