NOCTURNE - Complete Documentation

Version: 0.1.0
Status: Development (Hybrid Architecture)
License: MIT OR Apache-2.0
Last Updated: December 16, 2025

Table of Contents

Overview

What is NOCTURNE?

NOCTURNE is a modular Rust library for building **distributed state engines** with three core capabilities:

1. **TraumaEngine** - Selective memory management with provable forgetting
2. **MirrorNetwork** - Peer-to-peer state synchronization
3. **ForgetMachine** - Formal verification of forgetting policies

Design Philosophy

NOCTURNE is built on the principle that **distributed systems must forget intelligently**. Unlike traditional databases that aim for infinite retention, NOCTURNE provides:

- **Bounded memory** via configurable forgetting policies

- **Cryptographic proofs** of what was forgotten (auditability)
- **Peer-to-peer mirroring** for redundancy without central coordination
- **Formal verification** of forgetting guarantees (optional Z3 integration)

Use Cases

Use Case	NOCTURNE Components	Why It Fits
Edge Computing	TraumaEngine + ForgetMachine	Bounded memory on resource-constrained devices
Privacy-First Systems	TraumaEngine + ForgetProof	GDPR/LGPD compliance with cryptographic evidence
P2P Networks	MirrorNetwork + TraumaEngine	Decentralized state sync without blockchain overhead
Formal Verification	ForgetMachine + Z3	Prove safety properties of data retention
Time-Series Databases	TraumaEngine (time-based policy)	Auto-expire old data with audit trail

---

Architecture

High-Level Overview

```
┌─────────────────────────────────────────────────────────────┐
│              Application Layer                    │
│  (Your code using NOCTURNE as a library)              │
└──────────────────────────┬──────────────────────────────────┘
          │
   ┌─────────────┴─────────────┐
   │  NOCTURNE Core  │
   └─────────────┬─────────────┘
          │
  ┌──────────────────┼──────────────────┐
  │     │     │
┌───▼───┐ ┌───▼───┐ ┌───▼───┐
│ Trauma │  │ Mirror │  │ Forget │
│ Engine │  │Network │  │Machine │
└───────┘ └───────┘ └───────┘
  │     │     │
  ├──────────────────┼──────────────────┤
  │ Shared: │ Shared: │ Optional:
  │ - State │ - P2P  │ - Z3
  │ - Proof │ - Sync │ - ZKP
  └──────────────────┴──────────────────┴─────────────────
```

Module Structure

```
nocturne/
├── Cargo.toml          # Package manifest + features
├── README.md            # This file
├── DESIGN.md            # Design decisions (see below)
├── CHANGELOG.md         # Version history
├── LICENSE-MIT
├── LICENSE-APACHE
├── src/
│   ├── lib.rs           # Public API + re-exports
│   ├── trauma/
│   │   ├── mod.rs        # TraumaEngine entry point
│   │   ├── engine.rs     # Core engine implementation
│   │   ├── policy.rs     # ForgetPolicy enum
│   │   ├── proof.rs      # ForgetProof struct
│   │   └── cell.rs       # MemoryCell struct
│   ├── mirror/
│   │   ├── mod.rs        # MirrorNetwork entry point
│   │   ├── network.rs    # P2P broadcast logic
│   │   ├── transport.rs  # Generic Transport trait
│   │   ├── peer.rs       # PeerId + PeerInfo
│   │   └── sync.rs       # State synchronization
│   ├── forget/
│   │   ├── mod.rs        # ForgetMachine entry point
│   │   ├── verifier.rs   # Z3 integration (feature-gated)
│   │   └── cache.rs      # LRU cache (optional)
│   └── proof/
│       ├── mod.rs        # Proof API
│       └── placeholder.rs  # ZK proof stub
├── examples/
│   ├── quickstart.rs     # Basic usage
│   ├── time_based.rs     # Time-based forgetting
│   ├── p2p_mirror.rs     # P2P sync demo
│   └── formal_verify.rs  # Z3 verification demo
└── tests/
    ├── integration.rs    # End-to-end tests
    ├── trauma_tests.rs   # TraumaEngine unit tests
    ├── mirror_tests.rs   # MirrorNetwork unit tests
    └── forget_tests.rs   # ForgetMachine unit tests
```

Quick Start

Installation

Add to your `Cargo.toml`:

```toml
[dependencies]
nocturne = "0.1"

# Or with features:
nocturne = { version = "0.1", features = ["async", "zkp", "p2p"] }
```

Hello, NOCTURNE

```rust
```

```rust
use nocturne::{TraumaEngine, ForgetPolicy, MemoryCell};
use std::time::Duration;

fn main() {
    // 1. Create engine with time-based forgetting (max 60 seconds)
    let policy = ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(60),
    };
    let mut engine = TraumaEngine::new(policy);

    // 2. Ingest some data
    engine.ingest(b"hello world".to_vec(), vec!["tag1".to_string()]);
    engine.ingest(b"nocturne".to_vec(), vec!["tag2".to_string()]);

    println!("Buffer size: {}", engine.len()); // 2

    // 3. Run forget cycle (nothing old enough yet)
    engine.forget_cycle();
    println!("After cycle: {}", engine.len()); // Still 2

    // 4. Wait 61 seconds, then forget
    std::thread::sleep(Duration::from_secs(61));
    engine.forget_cycle();
    println!("After aging: {}", engine.len()); // 0 (all forgotten)

    // 5. Export proofs for audit
    let proofs = engine.export_proofs();
    println!("Forgotten items: {}", proofs.len()); // 2
    for proof in proofs {
        println!("  - Cell hash: {}", hex::encode(proof.cell_hash));
    }
}
```

Output:

```
Buffer size: 2
After cycle: 2
After aging: 0
Forgotten items: 2
  - Cell hash: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
  - Cell hash: 8c6744c9d42ec2cb9e8885b54ff744d0eb551d07d6d0f34f5b5e3b6f7f8e8b0a
```

Core Concepts

1. TraumaEngine

The **TraumaEngine** is the core state container with intelligent forgetting.

Key Types

```rust
pub struct TraumaEngine {
    buffer: VecDeque<MemoryCell>,
    policy: ForgetPolicy,
    proofs: Vec<ForgetProof>,
}

pub struct MemoryCell {
    pub data: Vec<u8>,
    pub ts: Instant,
    pub tags: Vec<String>,
}

pub enum ForgetPolicy {
    TimeBased { max_age: Duration },
    EntropyBased { entropy_threshold: u8 },
    Custom(Box<dyn Fn(&MemoryCell) -> bool + Send + Sync>),
}

pub struct ForgetProof {
    pub cell_hash: [u8; 32],
    pub forgotten_at: Instant,
    pub extra: Option<Vec<u8>>,
}
```

Operations

Method	Description	Time Complexity
`new(policy)`	Create engine with policy	O(1)
`ingest(data, tags)`	Add new cell to buffer	O(1)
`forget_cycle()`	Apply policy, remove old cells	O(n)
`export_proofs()`	Get all ForgetProofs	O(1) clone
`len()`	Current buffer size	O(1)

Forgetting Policies

Time-Based (most common):

```rust
let policy = ForgetPolicy::TimeBased {
    max_age: Duration::from_secs(3600), // 1 hour
};
```

Entropy-Based (probabilistic):

```rust
let policy = ForgetPolicy::EntropyBased {
    entropy_threshold: 16, // Keep only cells with ≥16 leading zero bits in hash
};
```

Custom (user-defined):

```rust
let policy = ForgetPolicy::Custom(Box::new(|cell| {
    // Forget if data size > 1KB
    cell.data.len() > 1024
}));
```

---

2. MirrorNetwork

The **MirrorNetwork** handles peer-to-peer state synchronization.

Key Types

```rust
```

```
pub struct MirrorNetwork {
    pub self_id: NodeId,
    pub replicas: ReplicaSet,
    transport: NetworkTransport,
}

pub struct NodeId(pub String);

pub struct ReplicaSet {
    pub replicas: HashMap<NodeId, [u8; 32]>, // NodeId → state hash
}

pub trait Transport {
    fn request_commitment(&self, peer_id: &PeerId) -> Result<[u8; 32]>;
    fn request_state(&self, peer_id: &PeerId) -> Result<TraumaState>;
}
```

Operations

Method	Description	Network Calls
`new(id, transport)`	Create network	0
`add_peer(id)`	Register peer	0
`broadcast_state(hash)`	Send state to all peers	O(n) peers
`sync_peer(engine, peer_id)`	Pull state from peer	2 (commitment + state)
`aggregate_view_hash()`	Global consistency check	0

P2P Workflow

```
Node A                    Node B
  │                         │
  ├─ broadcast_state(hash_a) ──▶ │
  │                         ├─ receive_state(hash_a)
  │                         │  (compare with local hash_b)
  │                         │
  │ ◀─ request: sync needed ─────┤
  │                         │
  ├─ send TraumaState ──────────▶ │
  │                         ├─ validate commitment
  │                         └─ apply to local engine
  │
```

---

3. ForgetMachine

The **ForgetMachine** provides formal verification of forgetting policies.

Key Types

```rust
pub struct ForgetMachine {
    #[cfg(feature = "verification")]
    ctx: Context, // Z3 context
}
```

Operations

Method	Description	Requires Feature
`new()`	Create verifier	`verification` (optional)
`verify_cell(policy, cell)`	Prove cell satisfies policy	`verification`
`verify_batch(policy, cells)`	Prove all cells satisfy policy	`verification`

Z3 Integration Example

```rust
```

```
#[cfg(feature = "verification")]
use nocturne::{ForgetMachine, ForgetPolicy, MemoryCell};
use std::time::Duration;

fn main() {
    let verifier = ForgetMachine::new();
    let policy = ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(60),
    };

    // Create a cell that's 70 seconds old
    let mut cell = MemoryCell {
        data: b"old data".to_vec(),
        ts: Instant::now() - Duration::from_secs(70),
        tags: vec![],
    };

    // Verify it should be forgotten
    let should_forget = verifier.verify_cell(&policy, &cell);
    assert!(should_forget); // Z3 proves: age > max_age

    println!("Z3 verification passed: cell is correctly marked for forgetting");
}
```

API Reference

TraumaEngine

```
rust
```

```
impl TraumaEngine {
    /// Create new engine with forgetting policy
    pub fn new(policy: ForgetPolicy) -> Self;

    /// Add data to buffer
    pub fn ingest(&mut self, data: Vec<u8>, tags: Vec<String>);

    /// Apply forgetting policy to buffer
    pub fn forget_cycle(&mut self);

    /// Export all forgetting proofs
    pub fn export_proofs(&self) -> Vec<ForgetProof>;

    /// Current buffer size
    pub fn len(&self) -> usize;

    /// Check if buffer is empty
    pub fn is_empty(&self) -> bool;

    /// Get buffer capacity
    pub fn capacity(&self) -> usize;

    /// Change forgetting policy (re-evaluates all cells)
    pub fn set_policy(&mut self, policy: ForgetPolicy);
}
```

MirrorNetwork

```
rust
```

```
impl MirrorNetwork {
    /// Create network with node ID and transport
    pub fn new(self_id: NodeId, transport: NetworkTransport) -> Self;

    /// Register a peer
    pub fn add_peer(&mut self, peer: NodeId);

    /// Remove a peer
    pub fn remove_peer(&mut self, id: &NodeId) -> Result<()>;

    /// Broadcast state hash to all peers
    pub fn broadcast_state(&self, state_hash: [u8; 32]);

    /// Sync state with specific peer
    pub fn sync_peer(
        &mut self,
        engine: &mut TraumaEngine,
        peer_id: &NodeId,
    ) -> Result<bool>;

    /// Compute aggregate view hash (all peers)
    pub fn aggregate_view_hash(&self) -> [u8; 32];

    /// Get list of known peers
    pub fn known_peers(&self) -> Vec<&NodeId>;
}
```

ForgetMachine

```rust
```

```
impl ForgetMachine {
    /// Create new verifier
    pub fn new() -> Self;

    /// Verify single cell against policy (requires Z3)
    #[cfg(feature = "verification")]
    pub fn verify_cell(&self, policy: &ForgetPolicy, cell: &MemoryCell) -> bool;

    /// Verify all cells in batch
    #[cfg(feature = "verification")]
    pub fn verify_batch(&self, policy: &ForgetPolicy, cells: &[MemoryCell]) -> Vec<bool>;

    /// Generate counter-example if verification fails
    #[cfg(feature = "verification")]
    pub fn find_counter_example(&self, policy: &ForgetPolicy) -> Option<MemoryCell>;
}
```

Feature Flags

NOCTURNE uses Cargo features to keep the core library lightweight:

Feature	Adds	Dependencies	Use Case
`default`	Core functionality only	`sha2`, `serde`, `hex`	Embedded, WASM
`async`	Async APIs	`tokio`	Async runtimes
`zkp`	Zero-knowledge proofs	(future: `bellman`, `halo2`)	Privacy-preserving proofs
`p2p`	Lattica transport	`lattica` (future)	Production P2P networks
`verification`	Z3 integration	`z3`	Formal verification

Usage

```
toml
```

```
# Minimal (no-std compatible)
nocturne = "0.1"

# With async support
nocturne = { version = "0.1", features = ["async"] }

# Full-featured
nocturne = { version = "0.1", features = ["async", "p2p", "verification"] }
```

Examples

Example 1: Time-Based Forgetting

```rust
use nocturne::{TraumaEngine, ForgetPolicy};
use std::time::Duration;

fn main() {
    // Forget anything older than 5 minutes
    let policy = ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(300),
    };
    let mut engine = TraumaEngine::new(policy);

    // Simulate event stream
    for i in 0..100 {
        engine.ingest(format!("event_{}", i).into_bytes(), vec![]);
        std::thread::sleep(Duration::from_millis(50)); // 50ms between events
    }

    println!("Total events: 100");
    println!("Buffer size before forget: {}", engine.len());

    engine.forget_cycle();
    println!("Buffer size after forget: {}", engine.len());
    println!("Forgotten events: {}", engine.export_proofs().len());
}
```

Example 2: Entropy-Based Pruning

```rust
```

```
use nocturne::{TraumaEngine, ForgetPolicy};

fn main() {
    // Keep only high-entropy data (≥20 leading zero bits in hash)
    let policy = ForgetPolicy::EntropyBased {
        entropy_threshold: 20,
    };
    let mut engine = TraumaEngine::new(policy);

    // Insert 1000 random payloads
    for i in 0..1000 {
        let data = format!("payload_{}", i).into_bytes();
        engine.ingest(data, vec![]);
    }

    println!("Before pruning: {} cells", engine.len());

    engine.forget_cycle();

    println!("After pruning: {} cells", engine.len());
    println!("Kept: ~{}% (probabilistic)", (engine.len() * 100) / 1000);
}
```

Example 3: P2P Mirroring

```
rust
```

```rust
use nocturne::{TraumaEngine, MirrorNetwork, NodeId, ForgetPolicy};
use std::time::Duration;

fn main() {
    // Node A
    let mut engine_a = TraumaEngine::new(ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(60),
    });
    engine_a.ingest(b"data from A".to_vec(), vec![]);

    let transport_a = NetworkTransport::dummy(); // Replace with real transport
    let mut network_a = MirrorNetwork::new(NodeId("node_a".to_string()), transport_a);

    // Node B
    let mut engine_b = TraumaEngine::new(ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(60),
    });
    let transport_b = NetworkTransport::dummy();
    let mut network_b = MirrorNetwork::new(NodeId("node_b".to_string()), transport_b);

    // Register peers
    network_a.add_peer(NodeId("node_b".to_string()));
    network_b.add_peer(NodeId("node_a".to_string()));

    // Sync A → B
    let synced = network_b.sync_peer(&mut engine_b, &NodeId("node_a".to_string()));
    println!("Sync successful: {:?}", synced);
    println!("Node B buffer: {}", engine_b.len()); // Should have A's data
}
```

Example 4: Formal Verification with Z3

```
rust
```

```
#[cfg(feature = "verification")]
use nocturne::{ForgetMachine, ForgetPolicy, MemoryCell};
use std::time::{Duration, Instant};

#[cfg(feature = "verification")]
fn main() {
    let verifier = ForgetMachine::new();
    let policy = ForgetPolicy::TimeBased {
        max_age: Duration::from_secs(60),
    };

    // Test case 1: Old cell (should be forgotten)
    let old_cell = MemoryCell {
        data: b"old".to_vec(),
        ts: Instant::now() - Duration::from_secs(70),
        tags: vec![],
    };
    assert!(verifier.verify_cell(&policy, &old_cell));
    println!("✓ Z3 verified: old cell correctly marked for deletion");

    // Test case 2: Recent cell (should be kept)
    let new_cell = MemoryCell {
        data: b"new".to_vec(),
        ts: Instant::now() - Duration::from_secs(30),
        tags: vec![],
    };
    assert!(!verifier.verify_cell(&policy, &new_cell));
    println!("✓ Z3 verified: recent cell correctly kept");
}

#[cfg(not(feature = "verification"))]
fn main() {
    println!("Enable 'verification' feature to run this example");
}
```

Design Decisions

Why VecDeque for TraumaEngine?

Trade-off: O(n) forget_cycle vs. O(1) ingest

Rationale:

- Most workloads are ingest-heavy (many writes, periodic forget cycles)

- VecDeque provides O(1) push_back (ingest) and efficient iteration (forget)
- Alternative (BTreeMap) would add O(log n) overhead on every ingest

Future: Add `TraumaEngine::with_capacity()` for pre-allocation

Why Generic Transport Trait?

Trade-off: Complexity vs. extensibility

Rationale:

- NOCTURNE doesn't prescribe a transport (TCP, UDP, Lattica, in-memory)
- Users can plug in any transport without forking the library
- Feature flags keep core library lightweight

Example Integration:

```rust
struct LatticaTransport {
    client: LatticaClient,
}

impl Transport for LatticaTransport {
    fn request_commitment(&self, peer: &PeerId) -> Result<[u8; 32]> {
        self.client.send_request(peer, "get_commitment").await
    }
}
```

Why Optional Z3 Integration?

Trade-off: Formal verification vs. binary size (~50MB for Z3)

Rationale:

- Most users don't need formal verification in production
- Research/auditing use cases benefit from Z3 proofs
- Feature flag keeps default build small

Why SHA-256 Instead of BLAKE3?

Trade-off: Speed vs. ubiquity

Rationale:

- SHA-256 is universally supported (hardware acceleration, WASM)

- BLAKE3 is faster but requires additional dependency
- Users can override `sha256()` helper if needed

Future: Add `crypto` feature flag for algorithm choice

Roadmap

Q1 2026 (v0.2.0)

- [ ] Complete ForgetMachine Z3 integration
- [ ] Add LRU cache implementation
- [ ] Lattica transport integration
- [ ] Benchmark suite (latency, throughput, memory)
- [ ] 100% test coverage

Q2 2026 (v0.3.0)

- [ ] Zero-knowledge proof support (bellman/halo2)
- [ ] Merkle tree commitments for proofs
- [ ] WASM compatibility
- [ ] Python bindings (PyO3)

Q3 2026 (v1.0.0)

- [ ] Production-ready status
- [ ] Security audit
- [ ] Performance tuning (SIMD, async)
- [ ] Extended documentation

Future (v2.0.0+)

- [ ] CRDTs for conflict-free replication
- [ ] Quantum-resistant hashing
- [ ] Hardware acceleration (GPU/FPGA)

Contributing

Development Setup

```bash
```

```
# Clone repository
git clone https://github.com/yourorg/nocturne
cd nocturne

# Run tests
cargo test --all-features

# Run benchmarks
cargo bench

# Format code
cargo fmt

# Lint
cargo clippy -- -D warnings

# Build docs
cargo doc --open
```

Testing

```bash
# Unit tests
cargo test

# Integration tests
cargo test --test integration

# With features
cargo test --features verification

# Coverage report
cargo tarpaulin --out Html
```

Contribution Guidelines

1. **Code Style**: Follow `rustfmt` defaults
2. **Tests**: Add tests for new features (min 80% coverage)
3. **Documentation**: Document all public APIs with `///` comments
4. **Performance**: Benchmark performance-critical code
5. **Licensing**: All contributions under MIT OR Apache-2.0

Issue Labels

Label	Meaning
`bug`	Something broken
`enhancement`	New feature request
`performance`	Optimization opportunity
`docs`	Documentation improvement
`good-first-issue`	Beginner-friendly

License

Dual-licensed under MIT OR Apache-2.0.

```
SPDX-License-Identifier: MIT OR Apache-2.0
```

Acknowledgments

NOCTURNE draws inspiration from:

- **GDPR/LGPD** - Right to be forgotten
- **CRDTs** - Conflict-free replicated data types
- **Raft/Paxos** - Consensus algorithms
- **ZK-SNARKs** - Zero-knowledge proofs

Special Thanks:

- Trauma-informed computing research
- Empathic distributed systems literature
- Ethical AI deletion frameworks

Support

- **Documentation**: https://docs.rs/nocturne
- **Issues**: https://github.com/yourorg/nocturne/issues
- **Discussions**: https://github.com/yourorg/nocturne/discussions
- **Email**: nocturne@yourorg.com

Document Version: 1.0.0
Last Updated: December 16, 2025
Maintainers: NOCTURNE Development Team