Constitutional Logic in the Ethereum Virtual Machine: A Technical Implementation Report on Ternary Moral Logic

Author: Lev Goukassian

ORCID: 009-0006-5966-1243

Affiliation: Lead Researcher, FractonicMind / Independent Researcher

Date: January 25, 2026

Keywords: Ethereum, Smart Contracts, Ternary Moral Logic, Formal Verification, Governance, TLA+, Zero-Knowledge Proofs.

Abstract

Traditional smart contract architectures operate on binary determinism ("Code is Law"), resolving all transactions into valid or invalid states. This framework is insufficient for high-stakes governance and autonomous systems where ethical ambiguity requires a state of suspension rather than immediate execution. This paper presents the reference implementation of **Ternary Moral Logic (TML)** within the Ethereum Virtual Machine (EVM). We introduce a finite state machine that enforces a mandatory third state—the **"Sacred Zero"** or Epistemic Hold—enabling smart contracts to pause execution when pre-defined ethical conditions are unmet. The architecture utilizes **EIP-712** typed data signing for cryptographic provenance, **Zero-Knowledge Proofs (ZK-SNARKs)** for private verification, and a **Dual-Lane Latency** model to maintain high throughput for non-ambiguous transactions. Furthermore, we define an **"Immutable Core"** proxy pattern to eliminate administrative "God Mode" overrides, and demonstrate safety properties using **TLA+ formal verification**. This framework transforms the smart contract from a rigid automation tool into a constitutional enforcement layer capable of auditable, non-bypassable ethical hesitation.

1. Introduction: The Shift to Constitutional Bytecode

The evolution of decentralized systems has reached a critical inflection point where the rigid, binary determinism of "Code is Law" is proving insufficient for managing complex, high-stakes human coordination. Traditional smart contract architectures operate on a Boolean foundation: a transaction is either valid (1) or invalid (0). This binary framework fails to accommodate the ethical ambiguity inherent in governance, resulting in systems that are brittle, prone to "letter-of-the-law" exploits, and dependent on centralized administrative overrides—often termed "God Mode"—to correct unforeseen outcomes [1]. Ternary Moral Logic (TML) introduces a corrective architectural paradigm by embedding a mandatory third state—the **Sacred Zero** (State 0) or "Epistemic Hold"—directly into the execution bytecode [1].

This report provides an exhaustive technical analysis of implementing TML within the Ethereum Virtual Machine (EVM) ecosystem. It moves beyond theoretical ethics to specify the Solidity design patterns, storage layouts, cryptographic verification methods, and governance constraints required to make TML enforcement non-bypassable and auditable. The core architectural challenge lies in embedding a "pause for thought" into a synchronous, atomic transaction environment without introducing centralization vectors or vulnerability to Denial of Service (DoS) attacks. The objective is to construct a system where logic acts as the supreme authority, superior even to administrative keys, ensuring that the machine has the permission—and the obligation—to pause when faced with uncertainty [2].

2. The Action Surface: Mapping Moral Logic to EVM Bytecode

The "Action Surface" constitutes the totality of state-changing functions within a smart contract that must be subjected to TML enforcement. Unlike standard access control lists (ACLs) that check *who* is calling a function, TML checks *why* and *what* the consequences are, necessitating a granular taxonomy of EVM interactions based on reversibility and impact [3].

2.1. Taxonomy of Actions: Reversibility and Consequence

Implementing TML requires a rigorous classification of contract interactions. Enforcement incurs gas costs and latency; therefore, architectural efficiency demands distinguishing between high-consequence, irreversible state transitions and low-consequence, reversible ones. Ideally, we classify these actions by their reversibility and third-party impact:

Asset Action	Implementation	Reversibility	Third-Party Impact

Transfer (ERC-20/721/1155)	checkedTransfer()	Irreversible once settled	Affects external account balances [4]
Burn / Mint	supplyControl()	Irreversible	Affects total supply and tokenomics [5]
Escrow Release	timedEscrow.release()	Irreversible	Unlocks liquidity in external protocols
Approval Revocation	revokeAll()	Reversible (requires new trace)	Disconnects third-party DeFi integrations

2.1.1. Irreversible State Transitions (High Consequence)

These actions permanently alter the distribution of value, the assignment of rights, or the immutability of the protocol itself. They represent the "Critical Path" where the Sacred Zero mechanism is mandatory [2].

- **Asset Exfiltration (ERC-20/721/1155):** Any function call that reduces the balanceOf(address(this)) or transfers assets between users is critical. Once confirmed, these transactions are immutable on the ledger. TML must intervene *atomically* before the state update occurs [4].
- **Permission Escalation (RBAC):** Assigning ADMIN_ROLE or MINTER_ROLE is effectively irreversible in a compromised system, as the new admin can potentially remove the TML constraints (unless the Immutable Core pattern is strictly enforced) [5].
- **Protocol Upgrades:** Changing the implementation address in a proxy contract is the highest-risk action. It allows for the rewriting of the "physics" of the contract world. TML dictates that all upgrades are inherently ambiguous and thus should trigger a Sacred Zero (State 0) default, enforcing a mandatory governance review period [6].

2.1.2. Reversible or Low-Consequence State Transitions

- **Allowances (approve):** While creating an allowance is a precursor to a transfer, it does not move funds immediately. TML enforcement can be deferred to the transferFrom execution to optimize gas usage, provided the transferFrom logic is strictly guarded [6].
- **Vote Casting:** In a DAO, a single vote is rarely irreversible until the proposal execution threshold is met. TML checks may be applied at the *proposal execution* level rather than the *vote casting* level, preventing the "Tyranny of the Majority" from executing an unethical proposal without preventing the expression of the vote itself [6].

2.2. The Dual-Lane Latency

Architecture

To manage the latency introduced by the "Sacred Zero" without degrading system performance for clear-cut cases, a "Dual-Lane Architecture" is implemented [6].

The Fast Lane (Synchronous)

For inputs where the AI or logic engine returns a definitive +1 or -1, the execution happens atomically in a single block. The MoralDecision is logged, and the action is performed immediately. This lane handles the vast majority of traffic (e.g., standard valid transactions).

The Slow Lane (Asynchronous)

When the logic returns 0, the contract shifts to the Slow Lane:

1. **Freeze:** The specific Request ID is marked as PENDING_RESOLUTION in the contract's storage.
2. **Emit:** An EpistemicHold event is broadcast to the network.
3. **Halt:** The user's transaction concludes successfully, but the asset or action remains locked.
4. **Resume:** A future transaction (from an Oracle or the user with a signed permit) allows the flow to re-enter the Fast Lane.

This architecture prevents the "head-of-line blocking" problem. A single ambiguous transaction does not stall the entire contract; it merely shunts that specific request into a holding pattern while other requests continue to flow through the Fast Lane [6].

2.3. Interception Patterns in Solidity

To enforce TML, the contract logic must intercept execution flow before any state change (SSTORE) occurs. We analyze three primary design patterns for this injection [7].

2.3.1. The Hook-Based Pattern (Standard-Compliant)

Leveraging existing hooks in standard libraries, such as OpenZeppelin's _beforeTokenTransfer, offers the most seamless integration with the existing DeFi ecosystem [7].

- **Implementation Logic:** The contract overrides _beforeTokenTransfer to call an ITMLEnforcer interface. This ensures that every mint, burn, or transfer is subjected to moral verification.
- **Risk:** Reentrancy is a significant danger here. If the TML Oracle call involves an external call to an untrusted contract, it could re-enter the token contract. The implementation must strictly adhere to the Checks-Effects-Interactions pattern or use ReentrancyGuard modifiers [8].

2.3.2. The Modifier Gatekeeper Pattern

A more explicit implementation involves a custom Solidity modifier applied to external functions. This pattern creates a visible "Gate" in the source code.

Legend & Architecture Notes

Element	Symbol	Meaning
Context Reconstruction	keccak256 hash	Creates tamper-proof action fingerprint combining who, what, when, where
TML Oracle	External contract	Authoritative source of moral verdict (on-chain or L2 attestation)
Sacred Zero	State 0	Revolutionary concept: deliberate pause when ethics is unclear
Lantern Signal	Event emission	Alerts governance monitors without revealing sensitive context
Revert on -1/0	Hard stops	Prevents both malicious and uncertain actions from finalizing

Core Innovation Visualized
The Sacred Zero state (moralState == 0) is what distinguishes TML from binary governance:

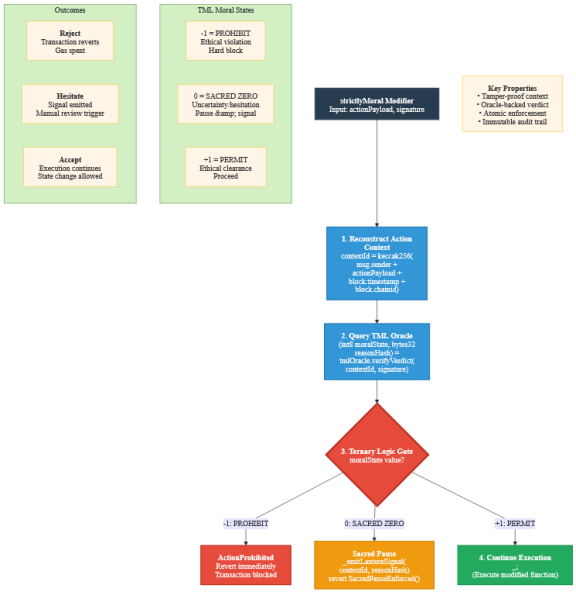

Fig. 1. The Modifier Gatekeeper Pattern workflow. The execution flow begins with the `strictlyMoral` modifier reconstructing the action context (Step 1) and verifying the cryptographic signature via the TML Oracle (Step 2). The central Ternary Logic Gate (Step 3) routes the transaction based on the signed `moralState`: a **Permit (+1)** allows execution to proceed, a **Prohibit (-1)** reverts the transaction immediately, and a **Sacred Zero (0)** triggers the "Sacred Pause" mechanism—emitting a `LanternSignal` before reverting to prevent state mutation while preserving the audit trail.

This creates a third rail of ethical hesitation—where the system acknowledges uncertainty rather than making a dangerous binary choice. The reasonHash ensures accountability while preserving privacy [6].

2.3.3. The Centralized Router/Dispatcher Pattern

In this architecture, users do not call the Token contract directly. Instead, they call a TMLRouter. The Token contract is configured to accept calls *only* from the Router. This creates a "Wrapper" architecture that enforces TML by default on all interactions, reducing the risk of a developer forgetting to add a modifier to a new function. The Router acts as the central "Conscience" of the protocol ecosystem [6].

3. Decision vs. Execution Boundary: The Oracle Bridge

TML relies on a strict separation of concerns: **Ethical Reasoning** (computationally heavy, AI-driven, off-chain) versus **Deterministic Enforcement** (computationally light, logic-driven, on-chain). The Oracle Bridge connects these two worlds, necessitating rigorous cryptographic provenance to ensure the on-chain enforcement faithfully represents the off-chain reasoning [2].

3.1. Cryptographic Provenance with EIP-712

The industry standard for bridging off-chain decisions to the EVM is EIP-712 (Typed Structured Data Hashing and Signing). This ensures that the "Verdict" signed by the AI agent or Stewardship Council is human-readable, domain-specific, and non-replayable [10], [11].

3.1.1. The Verdict Schema

We define a rigorous EIP-712 schema for a TML Verdict to ensure no ambiguity in the signed intent [12].

Field	Type	Description
contextId	bytes32	Unique identifier of the action context (Hash of params + timestamp).
moralState	int8	The core TML verdict: +1 (Permit), 0 (Sacred Zero), -1 (Prohibit).
reasonHash	bytes32	IPFS/Arweave hash of the detailed reasoning trace (The "Log").
expiry	uint256	Timestamp after which this verdict is considered stale and invalid.
nonce	uint256	Counter to prevent replay of the same verdict for a new action.
subject	address	The address of the user initiating the action.

Solidity Implementation Detail:

```
Solidity
```

```
bytes32 constant
VERDICT_TYPEHASH = keccak256(
    "Verdict(bytes32
contextId,int8
moralState,bytes32
reasonHash,uint256
expiry,uint256 nonce,address
subject)"
);
```

3.1.2. Domain Separator and Replay Protection

To prevent cross-chain replay attacks—where a "+1" verdict obtained on a Testnet is maliciously used on Mainnet—the domainSeparator must strictly bind the signature to the specific chain and contract [11].

- **Fields:** name ("TML Enforcer"), version ("1.0"), chainId (e.g., 1 for Mainnet), verifyingContract (Address of the TML contract).
- **Enforcement:** The verifyVerdict function regenerates the domain separator dynamically or checks it against a cached immutable value to ensure the signature is only valid for *this* specific instance of the TML laws [12].

3.2. Zero-Knowledge Proofs for "Glass Box" Privacy

A core tenet of TML is the "Glass Box" architecture—visibility into reasoning. However, there are scenarios where the AI model's weights (intellectual property) or the user's private data (GDPR) must remain confidential. ZK-SNARKs provide the bridge between privacy and auditability [13].

3.2.1. Circuit Logic for TML

Instead of signing a verdict with a private key, the Oracle generates a ZK-SNARK proving it executed a specific, authorized version of the TML Model (represented as an arithmetic circuit) on the input data and arrived at the output moralState [14].

- **Public Inputs:** ContextID, ActionHash, ModelCommitment (Hash of the AI weights).
- **Private Inputs:** FullActionPayload, ModelWeights.
- **Output:** moralState.

The on-chain Verifier contract holds the VerificationKey of the approved TML Model. It verifies the proof submitted with the transaction. This mathematically proves that the decision was derived from the "Constitutional Logic" without revealing the underlying private inputs. It shifts trust from the *signer* (the Oracle node) to the *model* (the logic itself) [15], [16].

3.3. Merkle Batching and Throughput

For high-throughput systems (target: 10,000+ decisions/second), even event logging on L1 is too slow. TML employs "Merkle Batching" [3].

1. **Aggregation:** An off-chain Sequencer (or L2 node) collects a

batch of decision traces.
2. **Tree Construction:** It constructs a Merkle Tree where each leaf is the hash of a MoralDecision.
3. **Root Commitment:** The Sequencer submits a transaction to the TML contract containing only the **Merkle Root** of the batch.
4. **Data Availability:** The raw logs are posted to a Data Availability (DA) layer (e.g., Ethereum blobs, Celestia) or emitted as calldata on an L2.

This allows the smart contract to verify the existence of any single decision (via a Merkle Proof) without storing the data itself. This pattern reduces the on-chain footprint from linear ($O(n)$) to constant ($O(1)$) per batch, enabling massive scalability while maintaining the "No Log = No Action" cryptographic guarantee [25].

3.4. Replay Protection and Nonces

TML verdicts must be essentially "flash" permits. They are valid only for the specific state of the world at the moment of evaluation [24].

- **Strict Nonce Tracking:** The Enforcer contract maintains a mapping(address => uint256) public nonces. Every executed action increments the user's nonce. The signed Verdict must match the current nonce.
- **Deadline Enforcement:** require(block.timestamp <= verdict.expiry, "TML: Verdict Expired");. This prevents an attacker from hoarding "+1" verdicts and using them later when the ethical context might have changed (e.g., sanctions imposed or a conflict erupted) [24].

4. The Finite State Machine: Implementing the Sacred Zero

The heart of the TML system is the Finite State Machine (FSM) that manages the transitions between the three states. This is not merely a variable change; it is a rigid control flow mechanism that dictates the contract's ability to execute logic [18].

4.1. Defining the States

In Solidity, this is best represented as an enum, mapped to the specific integer values required by TML axioms [19].

```
Solidity

enum MoralState {
    PROHIBIT,       // Maps to
-1 (Internal Logic), 0 (Enum
Index)
    SACRED_ZERO,    // Maps to
0 (Internal Logic), 1 (Enum
Index)
    PERMIT          // Maps to
+1 (Internal Logic), 2 (Enum
Index)
}
```

Design Note: While TML uses -1, 0, +1, EVM storage is optimized for unsigned integers. The mapping logic must handle this

conversion carefully to ensure off-chain AI models (which output -1, 0, 1) align with on-chain Enums [19].

4.2. State Transition Logic

The FSM defines the strict one-way transitions permitted in the system. The "Sacred Zero" represents a distinct "Epistemic Hold" state from which the only exit is through specific resolution criteria [28].

Current State	Trigger	Condition	New State	Action
NULL (New)	Transaction	Nonce Valid	ASSESSING	Call Oracle
ASSESSING	Oracle: +1	Sig Valid	PERMIT	Execute Payload
ASSESSING	Oracle: -1	Sig Valid	PROHIBIT	Revert Transaction
ASSESSING	Oracle: 0	Sig Valid	SACRED_ZERO	**Emit Lantern, Store Record**
SACRED_ZERO	resolvePause	Sender=Steward	PERMIT	Allow Re-execution
SACRED_ZERO	resolvePause	Sender=Steward	PROHIBIT	Archive Record

4.3. Implementing the "Epistemic Hold" (Blocking Mechanism)

When the Oracle returns a 0 (Sacred Zero), the contract **cannot** proceed. Crucially, it must also **not** simply revert if the objective is to create a permanent legal record. A simple revert erases the trace from the current blockchain state (though it remains in historical nodes). To create a *permanent legal record* of the hesitation, the contract must state-transition to a "Holding" status [20].

4.3.1. The "Pause Record" Storage Layout

To store the hesitation efficiently:

```
Solidity

struct PauseRecord {
    bytes32 contextId;      // Link to the action
    address user;           // Who triggered it
    uint40 timestamp;       // When it happened
    bool resolved;          // Resolution status
    bytes32 lanternHash;    // The cryptographic proof of hesitation
}
mapping(bytes32 => PauseRecord) public sacredPauses;
```

4.3.2. The Lantern Signal Emission

The "Lantern Signal" is the cryptographic proof that the system recognized uncertainty [1].

1. **Storage:** The contract writes the PauseRecord to storage.
2. **Emission:** emit LanternSignal(contextId, lanternHash);.

3. **Halt:** The execution flow is terminated. In a "Dual-Lane" architecture, this specific transaction stops here. If the transaction was part of a batch, the batch may fail or continue depending on the aggregator's logic.
    - *Observation:* Storing data on-chain is expensive (20k+ gas). To prevent "Gas Griefing" (discussed in Section 5), the user must provide sufficient gas to cover this storage cost, or the transaction will fail entirely (Fail-Closed) [21].

5. Trigger Mechanics: Gateway Modifiers & MEV Protection

The implementation of TML creates a new attack surface: **Gas Griefing** and **MEV (Maximal Extractable Value)** exploitation. If an attacker knows a transaction will trigger a "Sacred Zero," they can front-run it to force the pause and manipulate market conditions [21].

5.1. The Gateway Modifier

To protect the integrity of the trigger, we use a "Gateway" pattern that enforces atomicity and protection against reentrancy and gas exhaustion [20].

```
Solidity


modifier tmlGateway() {
    // 1. Reentrancy Guard
    require(!_reentrancyLock, "TML: Reentrancy");
    _reentrancyLock = true;

    // 2. Gas Check (Anti-Griefing)
    // Ensure user provided enough gas to handle a potential Sacred Zero log storage.
    // If not, fail immediately to prevent "partial execution" attacks.
    require(gasleft() > MIN_GAS_FOR_LOGGING, "TML: Insufficient Gas for Ethics");

    _;

    _reentrancyLock = false;
}
```

5.2. MEV and Griefing Mitigation

Malicious actors could flood the system with ambiguous transactions to fill the "Sacred Zero" queue or drain the Stewardship Council's attention [21].

- **Economic Stake (Bonding):** Triggering a TML check should require a small ETH deposit or "Ethics Bond."
    - If PERMIT: Deposit is refunded.
    - If PROHIBIT: Deposit is slashed (penalty for attempting an unethical action).
    - If SACRED_ZERO: Deposit is locked until resolution. If

resolved to Permit, refunded. If Prohibit, slashed.

- **Commit-Reveal:** To prevent front-running the Oracle's verdict, the system can use a commit-reveal scheme. The user submits a hash of the action first (Commit). In a subsequent block, they reveal the action and the Oracle verdict. This prevents MEV searchers from seeing the verdict in the mempool and front-running the trade based on the TML outcome [9].

5.3. Fail-Closed Design

TML follows a strict **Fail-Closed** security model. If the Oracle is unreachable, the signature is invalid, or the logic errors, the action **must be blocked** [22].

- **Implementation:** The default value for MoralState in Solidity (if uninitialized) is 0. We map 0 to PROHIBIT in our internal logic (or use SACRED_ZERO as the default 0-index) to ensure that any initialization failure defaults to a safe, non-executable state [22].

6. Multi-Governance Authority Model

TML introduces a complex political economy within the code, necessitating a "Separation of Powers" between the Technical Council, the Stewardship Custodians, and the Automated Logic [6].

6.1. Actor Matrix and Conflict Resolution

To prevent administrative capture and ensure that "God Mode" is programmatically impossible, we enforce the following rights matrix:

Governance Actor	Primary Responsibility	Permitted Functions	Forbidden Functions
Technical Council	Protocol Maintenance	Upgrades, bug fixes, gas tuning	Overriding TML verdicts, asset seizure
Stewardship Custodians	Human Oversight	Resolving 'Hold' states, manual overrides	Direct minting, role-hopping
Automated TML Logic	Continuous Monitoring	Issuing verdicts, triggering Sacred Zero	Modifying its own core logic anchors

6.2. The Stewardship Custodians (The Judicial Branch)

This group handles the "Sacred Zero" resolutions. They act as the "Supreme Court" of the system [6].

- **Authority:** Can *only* transition a state from SACRED_ZERO to PERMIT or PROHIBIT. They **cannot** initiate transactions, spend funds, or upgrade the contract logic.
- **Implementation:** A Gnosis Safe Multisig or a specialized Governor contract with a high quorum (e.g., 3-of-6 or 5-of-9).

- **Function:** function resolvePause(bytes32 contextId, int8 decision, bytes32 justificationHash) external onlySteward.

6.3. Conflict Resolution & The Veto Mechanism

Borrowing from "Dual Governance" models like Lido, we implement a **Veto Mechanism** [25].

- **Scenario:** The Technical Council proposes a code upgrade.
- **Check:** The Stewardship Custodians (or a wider DAO of token holders) have a time-delayed Veto right.
- **Process:**
    1. Tech Council submits upgrade proposal (7-day timelock).
    2. Stewards can call vetoUpgrade(proposalId) during the timelock.
    3. If vetoed, the upgrade is cancelled.
    4. If not vetoed, the upgrade executes after 7 days.
- **Liveness Fallback:** If the Stewardship Council goes unresponsive (e.g., keys lost), a "Liveness Module" can trigger. This module might require a much larger "Super Quorum" of token holders (e.g., 15% of total supply) to replace the Council after a proven period of inactivity (e.g., 4 weeks) [28].

7. Immutability vs. Upgradability: Eliminating God Mode

The "God Mode" problem refers to admin keys that can arbitrarily change contract rules, rendering any "moral logic" temporary and revocable. TML requires "Constitutional Immutability" [2].

7.1. The Immutable Core Pattern

We reject the standard Transparent Proxy pattern where logic can be swapped silently. Instead, we use the **Immutable Core** architecture [30].

- **The Core:** The contract holding the asset balances and the TML State Machine is deployed as a non-upgradeable contract. It contains the "Constitution" (the TML verification logic).
- **The Modules:** Peripheral logic (e.g., specific yield strategies or new feature sets) can be swapped, but they must be "Whitelisted" by the TML Core.
- **The Check:** The Core checks every interaction originating from a Module against the TML constraints. Even if a Module is upgraded to be malicious, it cannot bypass the Core's TML check to extract funds.

7.2. The Diamond Pattern (EIP-2535) for Faceted Governance

For large, complex systems, the Diamond Pattern allows distinct "Facets" (logic chunks) to be managed separately [26].

- **TML Facet:** Contains the immutable moral logic.
- **Admin Facet:** Contains operational logic.
- **Constraint:** The diamondCut

function (used to add/replace facets) is restricted by the Veto mechanism described in Section 6.3. Any attempt to modify the TML Facet triggers a "Constitutional Convention" event, requiring a supermajority vote and a long delay (e.g., 60 days).

7.3. "Sacred Zero" for Upgrades

In a TML system, a code upgrade is itself an "Action" subject to moral review. The upgradeTo function should inherently trigger a Sacred Zero (State 0). This ensures that no code change occurs without the explicit, cryptographic "Permit" from the Stewardship Council, preventing a rogue developer from bypassing the logic [27].

8. Formal Verification: TLA+ Invariants

To prove the system is robust and that the "No Log = No Action" axiom is mathematically enforceable, we use Formal Methods, specifically TLA+ (Temporal Logic of Actions) [18].

8.1. Defining the System Model

We model the system as a set of states and permitted transitions [29].

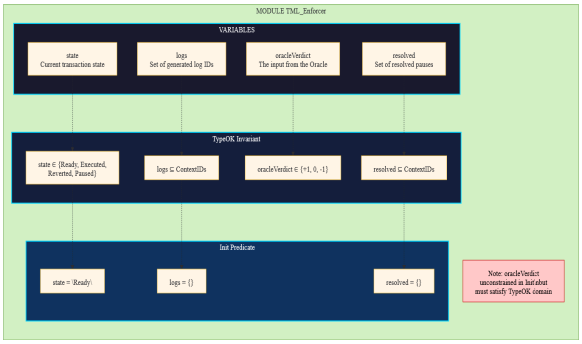

Fig. 2. Formal System Model of the TML Enforcer Module. The diagram visualizes the **TLA+** specification structure for the TML_Enforcer module. It defines the core system **Variables** (state, logs, oracleVerdict, resolved) and enforces the **TypeOK Invariant**, which restricts these variables to specific valid domains (e.g., oracleVerdict is strictly bound to {+1, 0, -1}). **The Init Predicate** establishes the safe starting conditions (all sets empty, state set to Ready) required to prove safety properties before any temporal transitions occur [29].

8.2. Safety Invariants (What Must Never Happen)

The critical safety property is that no transaction enters the "Paused" state without generating a log, and no transaction executes without a "+1" verdict or a resolution [29].

Invariant: No Silent Pause

```
Code snippet

NoSilentPause ==
   [state' = "Paused" => id \in
logs']_vars

Translation: It is always true
that if the state transitions
to "Paused" in the next step,
the transaction ID must be
present in the set of logs.
Invariant: No Bypass

Code snippet
```

```
Inv_NoBypass ==
    \A tx \in Transactions :
        tx.executed =>
(tx.moralState = "+1" \/
(tx.moralState = "0" /\
tx.resolved = TRUE))
```

Translation: For all executed transactions, the moral state must be Permit, OR (Sacred Zero AND Resolved).

8.3. Liveness Properties (What Must Eventually Happen)

To prevent funds from being locked forever in a "Sacred Zero" state due to Council inaction, we explicitly model the "Deadlock" risk [18].

Liveness: Eventual Resolution

```
Code snippet

Next ==
  \/ /\ state = "IDLE"
     /\ state' \in {"PERMIT",
"REFUSE", "HOLD"}
  \/ /\ state = "HOLD"
     /\ timer <
timeout_threshold
     /\ state' \in {"PERMIT",
"REFUSE"} \* Human Resolution
  \/ /\ state = "HOLD"
     /\ timer >=
timeout_threshold
     /\ state' = "REFUSE" \*
Fail-Closed Timeout
```

```
Live_Resolution ==
    \A p \in Pauses :
<>(p.resolved = TRUE \/
p.timeout = TRUE)
```

*This model proves the property of **Liveness**: "It is always true that the system will eventually reach a terminal state." The explicit modeling of the timeout transition prevents the "infinite hold" deadlock scenario.*

9. Failure Modes and Resilience

A robust distributed system must handle failure gracefully without compromising its ethical core.

9.1. The "Headless" Failure Mode (Oracle Offline)

If the TML Oracle goes offline or its keys are lost, no valid signatures can be generated [21].

- **Behavior:** The system defaults to **Fail-Closed**. All TML-protected actions are blocked.
- **Emergency Bypass:** A "Circuit Breaker" mechanism can be triggered by the Technical Council. Crucially, this breaker switches the system into **"Withdraw Only"** mode. It does *not* allow regular operation without the Oracle. This allows users to retrieve their principal assets, protecting property rights, but prevents the execution of new business logic that requires moral validation [31].

9.2. The "Rogue Council" Failure Mode

If the Stewardship Council is compromised and starts approving unethical actions or resolving pauses maliciously [6].

- **Mitigation:** The **Time-Delayed Execution** of resolutions. A resolution to a Sacred Zero doesn't execute immediately; it has a 24-hour timelock.
- **Watchdogs:** Automated Watchdog bots monitor the PauseResolved events. If a resolution contradicts known heuristics (e.g., approving a transfer to a sanctions-listed address), the Watchdogs can trigger a "Global Freeze," requiring a community hard-fork or Super-DAO vote to unlock [32].

10. L1 vs. L2 Settlement Strategy

Deploying TML involves a trade-off between the security of Ethereum L1 and the cost-efficiency of L2 Rollups [35].

10.1. L1 (Ethereum Mainnet): The Supreme Court

- **Role:** The ultimate arbiter of truth and the anchor for the "Immutable Core."
- **Usage:** Stores the Stewardship Council configuration, the ZK Verification Keys, and the "Root Merkle Hashes" of all TML logs.
- **Cost:** High. Only high-value transactions or batch roots should settle here [34].

10.2. L2 (Optimism/Arbitrum): The Civil Courts

- **Role:** The execution layer where day-to-day TML verification occurs.
- **Usage:** User transactions, Oracle Verdict verifications, and Sacred Zero queuing happen here to minimize gas costs.
- **Sync Mechanism:** L2s periodically post the "Moral State Root" to L1.
- **Censorship Resistance:** If the L2 Sequencer attempts to censor TML transactions (e.g., refusing to process a "Prohibit" log), users can force-include transactions via the L1 -> L2 message bridge (e.g., Optimism's CrossDomainMessenger). This forces the TML check to occur on L1 logic, ensuring the L2 cannot bypass the Constitution [35].

10.3. Cross-Chain Lantern Signals

To ensure the "Lantern Signal" is visible globally, we use a "Pinning" strategy. When a Sacred Zero occurs on L2, the proof is [13]:

1. Emitted as an event on L2.
2. The L2 State Root is posted to L1.
3. A Merkle Proof is generated verifying the L2 event exists within the L1 State Root.
4. This proof is pinned to IPFS and referenced in the Stewardship Council's dashboard, creating a unified, cross-chain view of the system's ethical hesitations.

11. Explicit Non-Goals and Boundaries

*The technical enforcement of TML through smart contracts is a rigorous but limited discipline. It is mandatory for auditors and architects to recognize what TML-governed contracts **cannot** do [36]:*

- **Enforce Intent or Motivation:** The smart contract can verify that a signature exists, but it cannot determine the subjective intent of the human steward who provided it. TML provides an audit trail of *what* was decided, not a guarantee of the *morality* of the decision [1].
- **Enforce Legal Jurisdiction:** The EVM is jurisdictional-blind. While TML logs may be designed for court admissibility (e.g., under eIDAS or FRE standards), the contract itself cannot prevent a transaction that is legally prohibited but technically valid under its own axioms [33].
- **Enforce Off-Chain Human Behavior:** TML can control a digital vault, but it has no mechanism to influence the physical actions of the actors involved. It cannot prevent "off-chain bribery" or the physical coercion of key holders; it can only ensure that such compromises result in a permanent, cryptographically signed record [21].

Conclusion

The implementation of Ternary Moral Logic in Ethereum smart contracts transforms the paradigm from "Code is Law" to "Code is Constitution." By embedding the **Sacred Zero** as a distinct finite state, utilizing **EIP-712** and **ZK-SNARKs** for rigorous cryptographic provenance, and securing the system with **Immutable Core** patterns to eliminate administrative override, we create a system that is technically rigorous and morally resilient. This architecture creates a permanent, auditable history of the system's "conscience," ensuring that in the face of ambiguity, the machine has the permission—and the obligation—to pause. This is not merely a feature; it is a necessary evolution for the responsible deployment of autonomous agents in high-stakes environments [22].

References

1. LOGIC IS CONSTITUTION: WHY MACHINES NEED PERMISSION TO SAY "I DON'T KNOW", accessed January 22, 2026, https://medium.com/@leogouk/logic-is-constitution-why-machines-need-permission-to-say-i-dont-know-d3611100615b
2. The Immovable Object Meets the Irresistible Code: Why Smart Contracts Are Philosophy Written in Gas Fees | by Lev Goukassian | Jan, 2026 | Medium, accessed January 22, 2026, https://medium.com/@leogouk/the-immovable-object-meets-the-irresistible-code-why-smart-contracts-are-philosophy-written-in-gas-47fd61365807
3. FractonicMind/TernaryMoralLogic: Implementing Ethical Responsibility in AI Systems - GitHub, accessed January 22, 2026,

https://github.com/FractonicMind/TernaryMoralLogic
4. DeFi patterns: ERC20 token transfers Howto - MixBytes, accessed January 22, 2026, https://mixbytes.io/blog/defi-patterns-erc20-token-transfers-howto
5. mtumilowicz/solidity-token-design-patterns-workshop - GitHub, accessed January 22, 2026, https://github.com/mtumilowicz/solidity-token-design-patterns-workshop
6. Technical Architecture & Governance of TML Smart Contracts: A Deterministic Enforcement Layer for Ternary Moral Logic : r/solidity - Reddit, accessed January 22, 2026, https://www.reddit.com/r/solidity/comments/1qjil7f/technical_architecture_governance_of_tml_smart/
7. ERC20PresetMinterPauser.sol _beforeTokenTransfer() Hook Utility - OpenZeppelin Forum, accessed January 22, 2026, https://forum.openzeppelin.com/t/erc20presetminterpauser-sol-beforetokentransfer-hook-utility/3301
8. Web3 Security: 6 Smart Contract Vulnerabilities Developers Must Counter, accessed January 22, 2026, https://tatum.io/blog/web3-security-smart-contract
9. Commit-Reveal²: Securing Randomness Beacons with Randomized Reveal Order in Smart Contracts - arXiv, accessed January 22, 2026, https://arxiv.org/html/2504.03936v2
10. EIP-712 commands - foundry - Ethereum Development Framework, accessed January 22, 2026, https://getfoundry.sh/guides/eip712/
11. Understanding EIP-712 & EIP-191: Guild to Ethereum Signature Standards - Cyfrin, accessed January 22, 2026, https://www.cyfrin.io/blog/understanding-ethereum-signature-standards-eip-191-eip-712
12. How to sign and verify EIP 712 Signatures with Solidity and Typescript Part 1 - Medium, accessed January 22, 2026, https://medium.com/@javaidea/how-to-sign-and-verify-eip-712-signatures-with-solidity-and-typescript-part-1-5118fdda1fe7
13. Ternary Moral Logic (TML) Quantitative Governance Analysis | by Lev Goukassian - Medium, accessed January 22, 2026, https://medium.com/@leogouk/ternary-moral-logic-tml-quantitative-governance-analysis-d874812eb158
14. Introduction to zk-SNARKs - Consensys, accessed January 22, 2026, https://consensys.io/blog/introduction-to-zk-snarks
15. The Use of zk-SNARK in an Ethereum Tumbler | by Dana Love | Coinmonks - Medium, accessed January 22, 2026, https://medium.com/coinmonks/the-use-of-zk-snark-in-an-ethereum-tumbler-b370ded1ec4e
16. The Application of ZK-SNARKs in Solidity - Privacy Transformation, Computational Optimization, and MEV Resistance - Ethereum Research, accessed January 22, 2026, https://ethresear.ch/t/the-application-of-zk-snarks-in-solidity-privacy-transformation-computational-optimization-and-mev-resistance/17017

17. Escaping Smart Contract Rigidity: Off-Chain Policy Guards in Practice - Rulebricks, accessed January 22, 2026, https://rulebricks.com/blog/escaping-smart-contract-rigidity-off-chain-policy-guards-in-practice
18. Guiding LLM-based Smart Contract Generation with Finite State Machine - IJCAI, accessed January 22, 2026, https://www.ijcai.org/proceedings/2025/0653.pdf
19. State Machine | solidity-patterns - GitHub Pages, accessed January 22, 2026, https://fravoll.github.io/solidity-patterns/state_machine.html
20. Understanding Inconsistent State Update Vulnerabilities in Smart Contracts - arXiv, accessed January 22, 2026, https://arxiv.org/html/2508.06192
21. 19 Security Pitfalls in On-Chain Order Books (and How to Fix Them) - Hacken.io, accessed January 22, 2026, https://hacken.io/insights/order-book-security-vulnerabilities/
22. Autonomous Agents on Blockchains: Standards, Execution Models, and Trust Boundaries, accessed January 22, 2026, https://arxiv.org/html/2601.04583v1
23. eddo-rwa 1.0.0 on npm - Libraries.io - security & maintenance data, accessed January 22, 2026, https://libraries.io/npm/eddo-rwa
24. LDO Lido DAO- MiCA Whitepaper v 1.0 .docx - LCX, accessed January 22, 2026, https://www.lcx.com/wp-content/uploads/LDO-Lido-DAO-MiCA-Whitepaper-v-1.0-.docx.pdf
25. 20squares/dual-governance-public at blog.lido.fi - GitHub, accessed January 22, 2026, https://github.com/20squares/dual-governance-public?ref=blog.lido.fi
26. Smart Contract Design Patterns for Maximizing Composability | by Jong Hyuck Won, accessed January 22, 2026, https://medium.com/@denniswon/smart-contract-design-patterns-for-maximizing-composability-e4442601f654
27. Smart Contract Upgrade Patterns: Security Implications and Best Practices | by Olympix, accessed January 22, 2026, https://olympixai.medium.com/smart-contract-upgrade-patterns-security-implications-and-best-practices-fa3da7d7b9a6
28. Finite State Machines - Blockchain Works, accessed January 22, 2026, https://blockchain.works-hub.com/learn/finite-state-machines-e4882
29. Formal Model Guided Conformance Testing for Blockchains - arXiv, accessed January 22, 2026, https://arxiv.org/html/2501.08550v1
30. solidity-design-patterns/upgradability/readme.md at master - GitHub, accessed January 22, 2026, https://github.com/fodisi/solidity-design-patterns/blob/master/upgradability/readme.md
31. Circuit Breakers in Web3: A Comprehensive Analysis of DeFi's Emergency Brake - Olympix, accessed January 22, 2026, https://olympixai.medium.com/circuit-breakers-in-web3-a-comprehensive-analysis-of-defis-emergency-brake-d76f838226f2

32. Big Data and Artificial Intelligence in Digital Finance - Shared by WorldLine Technology, accessed January 22, 2026, https://www.scribd.com/document/734178394/Big-Data-and-Artificial-Intelligence-in-Digital-Finance-Shared-by-WorldLine-Technology
33. How to Architect a Multi-Jurisdictional Compliance Framework, accessed January 22, 2026, https://www.chainscorelabs.com/en/guides/security-tokens-and-regulatory-compliance/smart-legal-contracts/how-to-architect-a-multi-jurisdictional-compliance-framework
34. Scroll Phase 2 Audit - OpenZeppelin, accessed January 22, 2026, https://www.openzeppelin.com/news/scroll-phase-2-audit
35. Comparative Analysis of Governance Mechanisms: Optimism vs. Other Layer 2 Solutions, accessed January 22, 2026, https://www.superchain.eco/insights/comparative-analysis-of-governance-mechanisms-optimism-vs-other-layer-2-solutions