Security via State Collapse

Intrinsic Execution Semantics for Security-Native Systems

Honza Rožek

Contents

1 Introduction and Threat Model

This patent specification describes a security architecture in which protection against unauthorized, invalid, or adversarial execution is achieved through intrinsic state collapse rather than external enforcement mechanisms.

Conventional security systems rely on access control, policy enforcement, sandboxing, or runtime monitoring to prevent or detect unauthorized behavior. Such approaches assume that execution may continue in a degraded or partially compromised state.

In contrast, the proposed architecture defines security as a semantic property of execution state validity. Execution is permitted to proceed only while the execution state remains valid under defined security invariants. Violation of these invariants causes deterministic collapse of the execution state.

1.1 Threat Model

The threat model includes:

- unauthorized state modification,
- execution of invalid or malformed operations,
- adversarial attempts to bypass security checks,
- injection of untrusted or inconsistent state.

The threat model explicitly excludes reliance on trusted runtimes, secure perimeters, or external monitoring infrastructure.

1.2 Security Objective

The security objective is to ensure that:

- unauthorized execution cannot continue,
- compromised execution yields no valid result,
- security violations are irreversible and non-recoverable.

Security enforcement is achieved by defining execution semantics such that invalid states are non-executable.

2 State Validity and Security Semantics

This section defines the concept of execution state validity and establishes security as a semantic property of execution rather than as an externally enforced control mechanism.

2.1 Execution State Model

Execution is defined over an explicit execution state space $\mathcal{S}$. Each execution state $s \in \mathcal{S}$ comprises all information required to determine subsequent execution behavior, including but not limited to:

- program or computation state,
- control flow position,
- data values and bindings,
- security-relevant metadata.

Execution proceeds through deterministic state transitions

$$s_0 \to s_1 \to \cdots \to s_n$$

subject to state validity constraints.

2.2 State Validity Predicate

A state validity predicate $V : \mathcal{S} \to \{\text{valid}, \text{invalid}\}$ is defined over the execution state space.

A state s is *valid* if and only if all defined security invariants hold for that state. Validity is a semantic property of the state itself and does not depend on external authorization checks, runtime monitors, or policy engines.

Definition:

Only valid execution states are executable.

2.3 Security as a Semantic Property

Security is defined as preservation of state validity across execution.

Unlike conventional security models, which permit execution to continue while monitoring or enforcing access constraints, the proposed architecture embeds security directly into execution semantics.

Any transition producing an invalid state violates execution semantics and cannot be executed further.

2.4 Validity-Preserving Transitions

An execution transition function

$$T : \mathcal{S} \times I \to \mathcal{S}$$

is defined only for valid states.

Validity Preservation Rule:

> For any valid state s and input i, execution of $T(s, i)$ is permitted only if the resulting state s' satisfies $V(s') = \text{valid}$.

Transitions that would result in invalid states are not executable and trigger state collapse.

2.5 Distinction from Access Control

In access control systems, execution is permitted but constrained by policy checks that may succeed or fail at runtime.

In contrast, the proposed model does not allow execution of invalid states under any circumstances. There is no notion of partially authorized execution or graceful degradation.

Security violations are expressed as semantic impossibility rather than policy failure.

2.6 No Trusted Boundary Assumption

State validity does not rely on trusted runtimes, trusted kernels, or trusted execution boundaries.

Validity is determined solely by execution semantics and invariant evaluation. Any attempt to bypass or manipulate enforcement manifests as invalid state construction and results in collapse.

2.7 Explicit Encoding of Security Conditions

Security-relevant conditions are explicitly encoded in execution semantics and state representation rather than inferred from external context.

Examples include:

- invariants over state structure,
- consistency constraints,
- authorization relationships encoded as state properties.

This explicit encoding ensures that security conditions participate directly in state validity evaluation.

2.8 Consequences of Invalid State Construction

If execution produces or attempts to produce an invalid state:

- the state is non-executable,
- execution cannot proceed,
- no valid execution result can be derived.

Invalid states are not recoverable through rollback, retries, or exception handling.

2.9 Semantic Security Guarantee

The security semantics defined in this section provide the following guarantee:

> Unauthorized or adversarial execution cannot produce a valid execution result, because any violation of security invariants results in non-executable state.

Security is therefore enforced by semantic necessity rather than by runtime intervention.

3 State Collapse Mechanisms

This section defines mechanisms by which execution enters a collapsed state when state validity is violated. State collapse is a deterministic and irreversible semantic outcome that prevents further execution and precludes generation of a valid result.

3.1 Definition of State Collapse

State collapse is defined as a terminal semantic condition in which the execution state becomes non-executable.

Definition:

> A collapsed state is a state for which no valid execution transition exists.

Once a state is collapsed, execution cannot proceed under any circumstances.

3.2 Trigger Conditions for Collapse

State collapse is triggered whenever an execution transition would produce an invalid state according to the state validity predicate.

Trigger conditions include, but are not limited to:

- violation of security invariants,
- unauthorized state modification,
- inconsistent or malformed state construction,
- attempted bypass of semantic constraints,
- execution of undefined or invalid operations.

Collapse is triggered at the moment of invalid state construction or detection.

3.3 Deterministic Collapse Semantics

State collapse occurs deterministically and does not depend on timing, probabilistic checks, or external intervention.

Collapse Rule:

> If execution would result in an invalid state, the system transitions to a collapsed state.

No alternative execution path exists once collapse conditions are met.

3.4 Irreversibility of Collapse

Collapsed states are irreversible.

Irreversibility Property:

There exists no valid transition from a collapsed state to a valid state.

Rollback, exception handling, or recovery mechanisms do not restore execution from collapse.

3.5 Distinction from Exceptions and Errors

State collapse is distinct from exceptions, faults, or recoverable errors.

- Exceptions allow controlled continuation of execution.
- Errors may permit retries or partial results.
- Collapse permanently terminates execution semantics.

State collapse therefore represents a stronger semantic guarantee than runtime error handling.

3.6 Collapse Without Observation

State collapse does not require detection by monitors, logs, or observers.

Collapse occurs as a direct consequence of semantic invalidity and is independent of observability.

3.7 Containment via Collapse

Collapse inherently contains security violations.

Containment Property:

No execution effects beyond the collapsed state are possible.

This prevents propagation of compromised state or leakage of sensitive information.

3.8 Collapse as Security Enforcement

Security enforcement is achieved by defining execution semantics such that unauthorized behavior cannot continue.

Collapse replaces:

- access denials,
- sandbox escapes,
- policy enforcement failures,
- post-hoc intrusion detection.

The outcome of unauthorized execution is always termination without result.

3.9 Granularity of Collapse

Collapse may apply at different granularities depending on embodiment, including:

- instruction-level collapse,
- transaction-level collapse,
- process-level collapse,
- distributed execution collapse.

Granularity does not affect the semantic irreversibility of collapse.

3.10 State Collapse Summary

State collapse mechanisms ensure that:

1. invalid execution cannot proceed,
2. compromised execution yields no valid result,
3. security enforcement is deterministic and intrinsic,
4. recovery from violation is semantically impossible.

State collapse therefore serves as the foundational enforcement mechanism for the security architecture described in this patent.

4 Security-Native Enforcement

This section describes security-native enforcement as an intrinsic property of execution semantics. Enforcement is achieved by defining execution such that only valid states are executable and any violation results in deterministic state collapse.

4.1 Enforcement Embedded in Execution Semantics

In the proposed architecture, enforcement is not performed by external security modules, policy engines, or runtime monitors.

Instead, enforcement is embedded directly into execution semantics through:

- explicit state validity predicates,
- validity-preserving transition rules,
- deterministic state collapse on violation.

Execution is therefore security-aware by construction.

4.2 Elimination of Runtime Security Checks

Conventional systems rely on runtime checks to validate permissions, policies, or invariants during execution.

In contrast, the proposed model eliminates discretionary runtime checks by making invalid execution semantically impossible. Security violations do not produce failed checks; they produce non-executable states.

Property:

There exists no execution path that bypasses security enforcement.

4.3 Security Enforcement Without Observation

Security-native enforcement does not rely on observing execution behavior.

No logs, alerts, or intrusion detection mechanisms are required to enforce security. Enforcement occurs regardless of whether violations are observed or reported.

This property eliminates:

- silent security failures,
- delayed detection,
- reliance on trusted monitoring infrastructure.

4.4 Deterministic Enforcement Outcome

Enforcement outcomes are deterministic.

Enforcement Rule:

> Any execution transition that would violate state validity results in immediate and irreversible state collapse.

There are no conditional enforcement paths, retries, or fallback behaviors.

4.5 Uniform Enforcement Across Execution Contexts

Security-native enforcement applies uniformly across:

- local execution,
- distributed execution,
- virtualized environments,
- embedded systems.

Enforcement behavior does not depend on deployment topology, trust boundaries, or administrative configuration.

4.6 No Privileged Override

The architecture does not permit privileged override of security enforcement.

Administrative privileges, debugging modes, or trusted components cannot resume execution from a collapsed state.

Invariant:

> No actor can authorize execution of an invalid state.

4.7 Comparison to Policy-Based Enforcement

Policy-based enforcement systems evaluate rules to permit or deny actions.

In contrast, security-native enforcement does not evaluate permissions at runtime. Instead, execution semantics define what is executable.

As a result:

- policy misconfiguration cannot weaken enforcement,
- enforcement cannot be disabled,
- enforcement behavior is predictable and verifiable.

4.8 Enforcement as a Technical Effect

Security-native enforcement produces concrete technical effects, including:

- elimination of bypassable security layers,
- deterministic termination of unauthorized execution,
- prevention of partial or compromised results,
- reduction of attack surface.

These effects arise directly from execution semantics rather than from security policy logic.

4.9 Security-Native Enforcement Summary

Security-native enforcement ensures that:

1. security violations cannot progress execution,
2. enforcement is intrinsic and non-bypassable,
3. enforcement outcomes are deterministic and irreversible,
4. execution correctness and security are inseparable.

Security enforcement is therefore an inherent property of execution rather than an external control mechanism.

5 Isolation and Containment

This section describes isolation and containment properties of the security via state collapse architecture. The architecture ensures that state collapse is strictly confined to the affected execution context and does not propagate uncontrolled effects to unrelated execution domains.

5.1 Isolation as a Semantic Property

Isolation is defined as a semantic property of execution state boundaries rather than as a function of process separation, memory protection, or sandboxing.

Each execution context operates over a distinct execution state or state partition. State validity and collapse semantics are evaluated locally within that context.

Definition:

> State collapse affects only the execution context in which invalid state construction occurs.

5.2 Containment of Collapsed States

When state collapse occurs, execution transitions within the affected context cease immediately.

No side effects, state mutations, or outputs beyond the collapsed boundary are produced. Collapsed states do not emit partial results or intermediate effects.

Containment Property:

> Collapsed execution yields no externally observable valid effects.

5.3 Blast Radius Control

The blast radius of a collapse event is explicitly bounded by execution context definition.

Depending on embodiment, an execution context may correspond to:

- a single instruction or operation,
- a transaction,
- a process or task,
- a distributed execution shard.

Collapse semantics ensure that failure does not extend beyond the defined context boundary.

5.4 No Cross-Context Contamination

Collapsed states are non-propagating.

State data, partial computations, or invalid transitions from a collapsed context cannot influence other execution contexts.

This prevents:

- lateral movement,
- privilege escalation across contexts,
- leakage of compromised state.

5.5 Isolation Without Sandboxing

Isolation is achieved without reliance on:

- sandboxing mechanisms,
- virtualization barriers,
- hardware-enforced memory separation.

While such mechanisms may be used in specific embodiments, isolation is fundamentally provided by execution semantics and state validity enforcement.

5.6 Deterministic Termination of Affected Context

Upon collapse, the affected execution context terminates deterministically.

There is no retry, fallback, or degraded execution mode within the same context. Continuation requires explicit reinitialization with a new valid initial state.

5.7 Isolation in Distributed Systems

In distributed embodiments, state collapse on one node or shard does not invalidate execution on other nodes.

Each node or shard enforces state validity independently. Coordination protocols do not propagate collapse beyond the originating execution context unless explicitly defined by system semantics.

5.8 Containment Versus Fault Tolerance

The architecture distinguishes containment from fault tolerance.

Fault-tolerant systems attempt to recover from errors and continue execution. In contrast, containment via state collapse prioritizes security and correctness over continuity.

Principle:

> Security violations are contained, not recovered from.

5.9 Isolation and Containment Summary

Isolation and containment properties ensure that:

1. collapse effects are strictly localized,
2. invalid execution produces no valid outputs,
3. compromised states do not propagate,
4. blast radius is semantically bounded,
5. security violations do not destabilize unrelated execution.

These properties make state collapse a safe and controlled enforcement mechanism rather than a system-wide failure condition.

6 Comparison to Access Control Models

This section compares the proposed security via state collapse architecture to conventional access control models and demonstrates that the invention constitutes a fundamentally different security paradigm.

6.1 Overview of Access Control Models

Access control systems regulate execution by permitting or denying actions based on evaluated policies. Common models include:

- Role-Based Access Control (RBAC),
- Attribute-Based Access Control (ABAC),
- Access Control Lists (ACLs),
- Capability-based authorization.

In such systems, execution remains conceptually valid and may proceed whenever access checks succeed.

6.2 Security Enforcement Timing

Access control enforces security at decision points during execution.

Authorization is evaluated:

- before an operation is performed,
- or at runtime when a protected resource is accessed.

If authorization fails, the operation is denied but execution continues in a controlled manner.

In contrast, the proposed architecture enforces security at the level of state validity. Unauthorized execution attempts result in non-executable states and trigger collapse.

6.3 Continuity Versus Termination

Access control systems are designed to preserve execution continuity.

Even repeated authorization failures do not invalidate the execution state itself. The system remains in a valid state capable of further execution.

In contrast, security via state collapse prioritizes correctness and security over continuity. Any violation of state validity terminates execution semantics.

6.4 Policy Evaluation Versus Semantic Necessity

Access control relies on policy evaluation, rule matching, and decision logic.

Such policies:

- may be misconfigured,
- may be incomplete,
- may be bypassed through implementation errors.

The proposed architecture does not evaluate policies at runtime. Instead, security constraints are encoded directly into execution semantics.

Violation of security constraints is not a failed policy check but a semantic impossibility.

6.5 Trusted Components and Boundaries

Access control models assume trusted components such as:

- policy engines,
- authorization servers,
- trusted runtimes or kernels.

Security via state collapse does not rely on trusted enforcement boundaries. Any attempt to subvert enforcement manifests as invalid state construction and results in collapse.

6.6 Observability and Detection

Access control systems often depend on logs, alerts, or monitoring to detect security violations.

In contrast, state collapse does not require detection or observation. Enforcement occurs regardless of whether violations are logged or reported.

6.7 Recovery and Override

Access control systems commonly support:

- administrative overrides,
- exception handling,
- recovery or retry mechanisms.

The proposed architecture does not permit override or recovery from a collapsed state. Security violations are irreversible at the semantic level.

6.8 Granularity and Scope

Access control typically governs access to resources or operations.

Security via state collapse governs the validity of the entire execution state. Granularity is defined by execution context rather than by resource boundaries.

6.9 Technical Effects of State Collapse

Compared to access control models, the proposed architecture produces distinct technical effects:

- elimination of bypassable security checks,
- deterministic termination of unauthorized execution,
- prevention of partial or compromised results,
- reduction of reliance on trusted security infrastructure.

These effects arise directly from execution semantics rather than from policy evaluation.

6.10 Non-Equivalence to Access Control

Security via state collapse cannot be reduced to or implemented by access control models.

No combination of access control rules, policies, or enforcement layers can replicate the semantic irreversibility and determinism of state collapse without redefining execution semantics themselves.

6.11 Comparison Summary

In summary:

1. access control regulates actions within valid execution,
2. state collapse defines what execution is valid,
3. access control permits continuation after denial,
4. state collapse terminates execution on violation,
5. access control relies on policy enforcement,
6. state collapse relies on semantic necessity.

The proposed architecture therefore represents a distinct and technically superior approach to security enforcement.

7 Formal Properties and Invariants

This section defines formal properties and invariants of the security via state collapse architecture. These invariants characterize execution behavior under both normal and adversarial conditions and distinguish the invention from policy-based or monitoring-based security systems.

7.1 State Validity Invariant

State Validity Invariant:

> At all times during execution, the execution state is either valid or collapsed.
> No invalid but executable state exists.

This invariant ensures that execution never proceeds from a state that violates defined security semantics.

7.2 Validity Preservation Invariant

Validity Preservation Invariant:

> A valid state may transition only to another valid state or to a collapsed state.

Transitions producing invalid intermediate states are not executable and cannot occur within the execution model.

7.3 Collapse Irreversibility Invariant

Collapse Irreversibility Invariant:

> Once an execution state collapses, no transition exists that restores validity.

This invariant prohibits rollback, exception-based recovery, or privileged override following a security violation.

7.4 No Valid Result After Violation

Result Validity Invariant:

> Execution that enters a collapsed state produces no valid execution result.

This invariant ensures that unauthorized or adversarial execution yields no usable output, partial or otherwise.

7.5 Deterministic Enforcement Invariant

Deterministic Enforcement Invariant:

> For any given execution state and input, enforcement outcome is uniquely determined.

Security enforcement does not depend on timing, policy evaluation order, or external observation.

7.6 Context-Local Collapse Invariant

Context-Locality Invariant:

> State collapse affects only the execution context in which the violation occurs.

No execution context may induce collapse in another context unless explicitly defined by execution semantics.

7.7 Non-Bypassability Invariant

Non-Bypassability Invariant:

> No execution path exists that bypasses state validity evaluation.

This invariant ensures that enforcement cannot be disabled, reordered, or circumvented by execution flow manipulation.

7.8 Observability Independence Invariant

Observability Independence Invariant:

> Security enforcement is independent of logging, monitoring, or detection.

Collapse occurs regardless of whether violations are observed or recorded.

7.9 Trusted Component Independence

Trust Independence Invariant:

> Security enforcement does not rely on trusted components external to execution semantics.

Any attempt to compromise enforcement manifests as invalid state construction and results in collapse.

7.10 Invariant Summary

The security via state collapse architecture satisfies the following invariants:

1. no executable invalid state,
2. validity-preserving execution,
3. irreversible collapse on violation,
4. no valid result after security violation,
5. deterministic enforcement outcome,
6. context-local containment,
7. non-bypassable enforcement,
8. independence from observation and trusted boundaries.

Collectively, these invariants define a security architecture in which correctness and security are inseparable properties of execution semantics.

8 Failure Modes

This section analyzes failure modes of the security via state collapse architecture and demonstrates that failures are explicit, detectable, and contained. Failure conditions do not undermine security guarantees and do not permit generation of valid execution results.

8.1 Failure of Execution Progress

Execution may fail to progress due to:

- hardware faults,
- process termination,
- resource exhaustion,
- explicit abort conditions.

Such failures result in termination of execution without producing a valid result. No execution continuation occurs from an invalid or partial state.

8.2 Failure of State Validation

State validation may fail due to:

- violation of security invariants,
- inconsistent state construction,
- malformed or unauthorized operations.

Failure Effect:

Failure of state validation deterministically triggers state collapse.

Validation failure cannot be ignored, bypassed, or deferred.

8.3 Failure of Collapse Triggering

In embodiments where state validity is evaluated continuously, failure of a collapse triggering mechanism is treated as a violation of execution semantics.

Fail-Stop Property:

Inability to determine state validity results in execution invalidation.

Execution does not proceed under uncertain or unverifiable validity conditions.

8.4 Partial Execution and Intermediate States

Partial execution may occur prior to collapse or termination.

Intermediate states that precede collapse:

- are not externally committed,
- do not produce valid outputs,
- cannot be resumed or finalized.

Partial execution cannot be misrepresented as completed execution.

8.5 Denial-of-Service Considerations

State collapse is not a denial-of-service vulnerability.

Collapse affects only the execution context in which invalid state construction occurs. External actors cannot force collapse of unrelated contexts without violating state validity constraints within those contexts.

8.6 Failure in Distributed Execution

In distributed embodiments, failures may occur in individual nodes or shards.

Property:

> Failure or collapse in one execution context does not invalidate execution in other contexts unless explicitly defined by system semantics.

Distributed coordination protocols do not propagate collapse implicitly.

8.7 Failure of Trusted Infrastructure Assumptions

The architecture does not rely on trusted infrastructure for enforcement.

Failure of components traditionally assumed trusted, such as monitoring, logging, or policy services, does not affect enforcement behavior.

Any resulting inconsistency manifests as invalid state and results in collapse.

8.8 Adversarial Failure Scenarios

Adversarial attempts may include:

- forced state corruption,
- execution flow manipulation,
- enforcement bypass attempts,
- partial execution exploitation.

All such attempts result in either immediate collapse or termination without valid result.

8.9 Comparison to Silent Failure Modes

Conventional security systems may experience silent failures, including:

- suppressed alerts,
- incomplete logs,
- policy mis-evaluation.

In contrast, the proposed architecture converts such conditions into explicit and non-recoverable execution failure.

8.10 Failure Mode Summary

Failure modes of the security via state collapse architecture exhibit the following properties:

1. failures are explicit and detectable,
2. invalid execution cannot continue,
3. no valid result is produced after failure,
4. failure effects are context-local and contained,
5. security guarantees remain intact under failure.

Failure therefore reinforces security guarantees rather than weakening them.

9 Example Embodiments

This section describes non-limiting example embodiments of the security via state collapse architecture. The embodiments illustrate applicability across diverse execution environments without limiting the scope of the claims.

9.1 Virtual Machine Execution

In one embodiment, security via state collapse is implemented within a virtual machine.

The virtual machine defines:

- an explicit execution state,
- deterministic state transitions,
- a state validity predicate integrated into execution semantics.

Any instruction that would produce an invalid state triggers immediate collapse, terminating execution without producing a valid result.

9.2 Programming Language Runtime

In another embodiment, the architecture is integrated into a programming language runtime.

Language constructs such as function invocation, state mutation, and control flow transitions are defined only for valid execution states. Attempts to construct invalid states result in runtime-level collapse rather than exceptions or access denials.

9.3 Transactional Systems

In one embodiment, security via state collapse is applied to transactional execution, such as databases or ledger systems.

Each transaction executes within an isolated execution context. Violation of transactional invariants or authorization constraints causes collapse of the transaction context, preventing partial commits or inconsistent state.

9.4 Distributed Execution Environments

In a distributed embodiment, execution is partitioned across nodes, shards, or actors.

Each partition enforces state validity locally. Collapse in one partition does not propagate to others unless explicitly defined by coordination semantics.

This embodiment supports secure distributed computation without reliance on centralized access control.

9.5 Zero-Trust Execution Platforms

In one embodiment, the architecture is used in zero-trust execution environments.

All execution contexts are treated as untrusted. Security enforcement relies exclusively on state validity and collapse semantics rather than on identity, perimeter defenses, or trust assumptions.

9.6 Policy-Governed Execution

In another embodiment, execution is governed by explicit policies encoded as state invariants.

Policy compliance is enforced semantically. Violations result in state collapse rather than policy rejection or logging events.

9.7 Embedded and Resource-Constrained Systems

In one embodiment, security via state collapse is implemented in embedded or resource-constrained systems.

The absence of external monitoring, logging, or policy engines reduces system complexity while preserving strong security guarantees.

9.8 Security-Critical Workflows

In another embodiment, the architecture is applied to security-critical workflows, including:

- credential handling,
- authorization-sensitive computation,
- confidential data processing.

Any violation of security invariants results in immediate termination of the workflow without producing usable output.

9.9 Third-Party Verification and Inspection

In some embodiments, collapsed execution states and validity outcomes are exposed for external inspection.

External systems may verify that no valid result was produced following a security violation, without requiring access to internal execution details.

9.10 Embodiment Summary

The example embodiments demonstrate applicability of security via state collapse to:

1. virtual machines,
2. language runtimes,

3. transactional systems,
4. distributed execution environments,
5. zero-trust platforms,
6. policy-governed execution,
7. embedded systems,
8. security-critical workflows.

These embodiments illustrate that the proposed architecture is broadly applicable and not limited to any specific system design.

10 Conclusion

This patent specification has described a security architecture in which protection against unauthorized, invalid, or adversarial execution is achieved through intrinsic state collapse rather than through external enforcement, monitoring, or policy-based control.

Unlike conventional security systems that permit execution to continue under failed authorization checks or rely on detection and response mechanisms, the proposed architecture defines security as a semantic property of execution state validity. Execution is permitted to proceed only while the execution state remains valid under defined security invariants. Any violation results in deterministic and irreversible collapse of execution state.

State collapse is not an error-handling mechanism, access denial, or recovery path. It is a terminal semantic condition in which no further execution is possible and no valid result can be produced. Security enforcement is therefore achieved by semantic necessity rather than by discretionary runtime checks or trusted enforcement components.

Formal properties and invariants have been defined that guarantee:

- absence of executable invalid states,
- validity-preserving execution semantics,
- irreversible termination upon violation,
- prevention of valid results after unauthorized execution,
- deterministic and non-bypassable enforcement,
- context-local isolation and containment.

Failure modes have been analyzed and shown to reinforce, rather than weaken, security guarantees. Partial execution, validation failure, infrastructure failure, or adversarial manipulation result in explicit and contained execution termination without silent degradation or compromised outcomes.

Example embodiments demonstrate that security via state collapse is applicable across a wide range of systems, including virtual machines, programming language runtimes, transactional systems, distributed execution environments, zero-trust platforms, embedded systems, and security-critical workflows.

Collectively, these properties establish that the proposed architecture constitutes a fundamentally different and technically superior approach to security enforcement. By embedding security directly into execution semantics and making invalid execution non-executable by construction, the invention provides a robust foundation for secure, verifiable, and uncompromisable computation across heterogeneous execution environments.