The Inversion of Trust:

Vocabulary-Bounded Security in Protocol-Governed Systems

Bachi (aka Bhash Ganti) Contact: bachipeachy@gmail.com

Abstract

This paper analyzes the security properties that emerge structurally from protocol-governed architecture. Building on the deterministic enforcement model (Paper 6) [Bachi, 2026f] and the computation-mutation separation (Paper 7) [Bachi, 2026g], we demonstrate that protocol governance inverts conventional trust assumptions.

Rather than trusting implementation code, protocol-governed systems trust ratified behavioral law. Rather than defending against implicit behavior, they eliminate undeclared behavior. Rather than constraining execution with runtime guards, they bound mutation through vocabulary and constitutional enforcement.

This paper formalizes vocabulary-bounded security, mutation surface bounding, absence of ambient authority, and trace-based accountability. We compare this model to capability-based security and identify the security failure modes specific to protocol-governed systems.

This paper does not introduce cryptographic protocols, specific threat models, or economic analysis (Paper 9). Its purpose is architectural: to demonstrate that security emerges from structural constraints rather than defensive layering.

Categories: Software Engineering (cs.SE), Computer Security (cs.CR), Software Architecture

Keywords: security architecture, vocabulary-bounded security, trust inversion, ambient authority, attack surface reduction, capability security, governance-rooted trust, structural security

---

1. Position in the Series

Paper 6 defined deterministic enforcement: how execution engines compile, route, and execute protocol artifacts while emitting structured traces [Bachi, 2026f]. Paper 7 defined the computation-mutation boundary: the constitutional separation between CT_ (pure transforms) and CS_ (governed side effects) [Bachi, 2026g].

This paper asks a fundamental question:

> *Given deterministic enforcement and bounded mutation, what becomes the locus of trust?*

The answer: **trust shifts from runtime discretion to constitutional vocabulary**.

In conventional systems, trust is distributed across:

- Developers (to write correct code)
- Code reviews (to catch errors)
- Runtime guards (to prevent exploits)
- Operational monitoring (to detect anomalies)

Protocol-governed systems relocate trust:

- From implementation code to ratified protocol artifacts
- From runtime guards to governance validation
- From monitoring to structural impossibility

This relocation is not a policy choice. It is an architectural consequence of the separations established in Papers 6 and 7.

---

2. The Traditional Trust Model

2.1 Trust in Conventional Systems

In conventional software systems, the trust model operates under a flawed premise: that securing execution is equivalent to securing behavior.

When code is the source of truth, the trust model requires:

Trust Requirement	Description
Trust developers	Code authors must write secure code
Trust reviews	Reviewers must catch all vulnerabilities
Trust runtime	Execution environment must prevent exploits
Trust operations	Monitoring must detect anomalies
Trust all layers	Every component must maintain security

This model fails at scale because trust requirements are unbounded. Any component that can affect execution can potentially alter semantics. The attack surface includes application code, frameworks, libraries, runtimes, operating systems, hypervisors, and hardware.

2.2 Consequences of Conflation

When code defines both what a system means and how it executes, security faces structural problems:

Attack surface explosion: Every execution component is a potential semantic attack vector.

Defense sprawl: Security must be applied at every layer because compromise at any layer could alter meaning.

Verification impossibility: Proving behavior matches intent requires proving all execution pathways preserve intent—a problem that grows combinatorially with system complexity.

Recovery complexity: After breach, determining semantic damage requires reconstructing what the system did, which requires trusting execution records that may themselves be compromised.

2.3 The Zero-Trust Paradox

Zero-trust architecture attempts to address these limitations by assuming all components may be compromised. Every access requires verification; nothing is implicitly trusted.

Zero-trust is valuable but does not solve the fundamental problem: when code defines meaning, verifying execution requires understanding all code paths. Zero-trust verifies identity and access but cannot verify semantic integrity without inspecting behavior.

The paradox: zero-trust requires trusting the verification mechanisms, which are themselves code.

---

3. The Inversion of Trust

Protocol-governed systems invert the traditional trust model by separating semantic authority from execution.

3.1 Two Trust Planes

The architecture creates two distinct trust planes:

Semantic Plane (Protocol Layer) - Protocol artifacts define what the system means and may do - Artifacts are static, versioned, and immutable once ratified - Semantic authority resides exclusively in this plane

Execution Plane (Execution Layer and below) - Execution engines interpret protocol without defining meaning - Implementations are replaceable if trace-equivalent - Execution affects availability and data, not semantic authority

3.2 The Separation Principle

The separation principle states:

> **Compromise of the execution plane does not grant semantic authority.**

An attacker who fully controls the execution environment can:

- Prevent execution (denial of service)
- Observe execution (data exposure)
- Corrupt execution outputs (data integrity)

An attacker who controls the execution environment cannot:

- Define new behaviors (semantic authority requires ratified artifacts)
- Authorize new operations (governance authority requires ratification)
- Alter protocol meaning (semantic integrity is artifact-based)

The attacker can break the system but cannot make it do something it was not authorized to do.

3.3 Trust Relocation

Trust relocates from distributed runtime components to concentrated governance:

Traditional Trust	Protocol-Governed Trust
Trust all execution code	Trust ratified artifacts
Trust runtime guards	Trust governance validation
Trust monitoring to detect	Trust structure to prevent
Trust defense in depth	Trust vocabulary closure

This relocation transforms security from an unbounded problem (secure all execution) to a bounded problem (secure protocol artifacts and ensure execution faithfulness).

---

4. Vocabulary-Bounded Security

4.1 Definition

Let V be the constitutionally enumerated set of concerns and mutation primitives.

Vocabulary-bounded security states:

$$\text{Permissible behavior} \subseteq \text{Expressible in } V$$

All behavior that the system may exhibit must be expressible within the declared vocabulary. If a behavior is not declared in vocabulary, it cannot execute.

4.2 The Vocabulary Closure Invariant

I-S1 (Vocabulary Closure) No behavior may occur outside the declared concern vocabulary.

$$\forall \text{behavior } b : \text{Executed}(b) \Rightarrow b \in V$$

This invariant is enforced through:

1. **Prefix enforcement:** All artifacts must carry recognized concern prefixes
2. **Schema validation:** All artifacts must conform to their concern's schema
3. **Referential integrity:** All artifact references must resolve to declared artifacts
4. **Routing determinism:** Execution routes by prefix to known handlers

Behavior not expressible in vocabulary cannot pass governance, cannot be loaded, and cannot execute.

4.3 Vocabulary as Security Perimeter

Traditional security defines perimeters around networks, systems, or applications. Protocol governance defines the perimeter around vocabulary:

- All system behavior flows from protocol artifacts
- Protocol artifacts are expressible only within vocabulary
- Protecting vocabulary protects behavioral possibility
- Vocabulary is finite, static, and inspectable

The vocabulary is the constitutional boundary of the system's behavioral universe.

4.4 Vocabulary Enumeration

The concern vocabulary is explicitly enumerated (Paper 3) [Bachi, 2026c]:

Prefix	Concern	Security Relevance
AC_	Actors	Authority declaration
IN_	Intents	Authorization requests
WF_	Workflows	Behavioral orchestration
CC_	Capability Contracts	Capability declaration
RB_	Runtime Bindings	Implementation binding
EV_	Events	Observation declaration
CT_	Capability Transforms	Pure computation
CS_	Capability Side Effects	Mutation declaration

This enumeration is finite. Expanding vocabulary requires constitutional amendment through governance—a deliberately expensive process.

---

5. Absence of Ambient Authority

5.1 The Ambient Authority Problem

Ambient authority occurs when code inherits permissions implicitly from its execution context. In conventional systems:

- Running as root grants root permissions
- Running in a network zone grants network access
- Running with credentials grants credentialed operations
- Environment variables grant configuration authority

Ambient authority creates confused deputy problems: code with legitimate purpose is tricked into misusing its ambient permissions.

5.2 The No Ambient Authority Invariant

I-S2 (No Ambient Authority) Authority must be derivable from explicit AC_ artifacts and workflow declarations.

$$\text{Authority}(\text{operation}) = \text{Derive}(\text{AC\_ artifacts}, \text{WF\_ declarations})$$

Execution context does not grant capabilities. An operation has exactly the authority declared in its artifacts—no more.

5.3 Authority Derivation

Authority in protocol-governed systems is derivable:

1. **Actor declaration:** AC_ artifacts declare actors and their permissions
2. **Intent authorization:** IN_ artifacts declare what actors may request
3. **Workflow authorization:** WF_ artifacts declare what operations may occur
4. **Capability declaration:** CC_ artifacts declare what operations may do

At any point, the authority an operation possesses is derivable from artifact inspection. No runtime context grants additional authority.

5.4 Elimination of Confused Deputy

The confused deputy problem requires ambient authority: a trusted program is tricked into misusing permissions it holds implicitly.

Without ambient authority:

- Operations have only declared permissions
- Permissions are granted by artifacts, not context
- Deception cannot grant undeclared permissions
- The deputy has nothing implicit to be confused about

Confused deputy attacks are structurally eliminated, not merely defended against.

---

6. Mutation Surface Bounding

6.1 The Mutation Surface

From Paper 7, the mutation surface is defined:

$$\text{MutationSurface} = \{s : s \in \text{CS\_}\}$$

The mutation surface is the total set of operations that may change world state. Every state change occurs through a declared CS_ artifact.

6.2 The Bounded Mutation Surface Invariant

I-S3 (Bounded Mutation Surface) All state changes must occur through enumerated side-effect capabilities.

$$\text{StateChange}(op) \Rightarrow op \in \text{CS\_}$$

There is no implicit write path. No state change occurs outside the mutation surface.

6.3 Attack Surface Reduction

In conventional systems, attack surface grows with:

- Codebase size (more code, more vulnerabilities)
- Dynamic features (reflection, code loading)
- Hidden dependencies (transitive vulnerabilities)
- Interaction complexity (combinatorial edge cases)

In protocol-governed systems, attack surface equals:

$$|\text{Attack Surface}| = |\text{CS\_}| + |\text{AC\_}| + |\text{RB\_}|$$

That is: - Mutation primitives (what can change state) - Authority artifacts (what authority exists) - Binding declarations (what implementations are used)

No other vector is structurally admissible. Attack surface is finite and enumerable.

6.4 Structural Attack Surface Quantification

Attack surface can be quantified:

Metric	Definition
Mutation primitives	Count of CS_ artifacts
Authority scope	Permissions declared in AC_ artifacts
Binding surface	Count of RB_ artifacts
Total attack surface	Sum of above metrics

This quantification enables risk assessment based on structure, not code inspection.

---

7. Elimination of Undeclared Behavior

7.1 Sources of Undeclared Behavior

In conventional systems, undeclared behavior arises from:

Source	Description
Reflection	Runtime introspection that discovers and invokes code
Dynamic loading	Loading code at runtime based on data
Implicit dispatch	Polymorphism that routes to unknown implementations
Hidden control flow	Callbacks, events, interrupts that alter flow
Configuration interpretation	Configuration that becomes logic

These mechanisms enable behavior that was not explicitly declared at authoring time.

7.2 The No Undeclared Invocation Invariant

I-S4 (No Undeclared Invocation) All invocation paths must originate from ratified workflow declarations.

$$\text{Invoked}(op) \Rightarrow \exists \text{WF} : \text{Declares}(\text{WF}, op)$$

Execution may not discover or infer behavior. All invocation is declared.

7.3 Structural Prevention

Protocol governance structurally prevents undeclared behavior:

No reflection: Execution engines interpret declared structure; they cannot inspect or modify artifacts at runtime.

No dynamic loading: All artifacts are loaded at initialization from governance-ratified sources; no runtime artifact generation.

No implicit dispatch: Routing is deterministic by concern prefix; there is no polymorphic dispatch to unknown implementations.

No hidden control flow: All workflow control flow is explicitly declared in WF_ artifacts; no implicit callbacks or events.

7.4 Logic Injection Elimination

Logic injection attacks (SQL injection, command injection, XSS) exploit the conflation of data and code. User input becomes executable logic.

Protocol governance eliminates logic injection:

- Behavioral logic is defined in protocol artifacts, not derived from input
- Input is data that flows through declared operations
- Operations are fixed at authoring time, not constructed at runtime
- No amount of crafted input can introduce new operations

Logic injection is structurally impossible, not merely defended against.

---

8. Trace-Based Accountability

8.1 The Accountability Requirement

Security requires accountability: the ability to determine what happened, who caused it, and whether it was authorized.

Traditional accountability relies on logs—records that may be incomplete, inconsistent, or compromised.

8.2 The Total Trace Accountability Invariant

I-S5 (Total Trace Accountability) Every state mutation and computation step is attributable to a versioned artifact and actor.

$$\forall \text{step} \in \text{Execution} : \text{Attributable}(\text{step}, \text{artifact}, \text{actor})$$

Attribution is structural, not reconstructed. Every trace event records:

- Which artifact authorized the operation
- Which actor initiated the workflow

- Which capability was invoked
- What inputs and outputs occurred

8.3 Evidence-First Security

Protocol governance implements evidence-first security:

Traditional Approach	Protocol-Governed Approach
Detect anomalies	Prove authorization
Investigate incidents	Replay execution
Reconstruct events	Read trace evidence
Trust log integrity	Verify trace against artifacts

Security is not detection-first. It is evidence-first.

8.4 Forensic Advantage

Traces provide forensic advantages:

Replay capability: Traces contain sufficient information to replay execution (for CT_) or verify outcomes (for CS_).

Semantic integrity verification: Compare executed protocol to enacted protocol to detect any semantic deviations.

Tamper detection: Hash chains in traces detect modification (Paper 6, I-E18).

Attribution certainty: Every action is attributable to declared artifacts and actors.

Post-incident analysis shifts from "what might have happened" to "what exactly happened."

---

9. Implementation Replaceability and Trust

9.1 Execution Engine Replaceability

From Papers 1 and 6, execution engines are replaceable if trace-equivalent:

$$\text{Conforming}(E) \Rightarrow \forall w, i, b : \text{Trace}_E(w, i, b) \equiv \text{Trace}_{\text{reference}}(w, i, b)$$

Different engines may execute the same protocol with identical behavioral results.

9.2 The Implementation Non-Authority Invariant

I-S6 (Implementation Non-Authority) Security guarantees derive from protocol artifacts, not from specific execution engine implementations.

$$\text{SecurityGuarantees} = f(\text{Protocol Artifacts})$$

$$\text{SecurityGuarantees} \neq f(\text{Execution Implementation})$$

Trust is relocated upward to the protocol layer. The execution engine is a low-trust utility.

9.3 Implications for Trust

Implementation replaceability transforms trust:

Execution as utility: Execution infrastructure is treated as a low-trust utility. Harden it for availability, but do not depend on it for semantic integrity.

Artifact as authority: Protocol artifacts are the semantic crown jewels. Protect them with intensity traditionally applied to code.

Engine-independent security: Security analysis applies to artifacts, not implementations. A security property proven for artifacts holds regardless of execution engine.

9.4 Bounded Execution Compromise

When execution is compromised:

Impact	Bounded By
Availability	Can be disrupted
Data confidentiality	Data in execution may be exposed
Data integrity	Outputs may be corrupted
Semantic authority	Cannot grant new behaviors
Governance authority	Cannot ratify new artifacts

Execution compromise is an availability and data problem, not a semantic integrity catastrophe.

---

10. Governance as Security Root

10.1 Governance Enforcement

The Governance layer (Paper 4) enforces [Bachi, 2026d]:

- Schema compliance (artifacts conform to declared structure)
- Prefix uniqueness (no prefix collisions)
- Referential integrity (all references resolve)
- Vocabulary boundedness (all concerns are declared)

These enforcement mechanisms provide security properties.

10.2 The Governance-Rooted Trust Invariant

I-S7 (Governance-Rooted Trust) The root of system trust is constitutional governance, not runtime monitoring.

$$\text{TrustRoot} = \text{Governance}$$

Security begins with governance. If governance is sound, the system can only exhibit declared behavior. If governance is compromised, semantic authority is compromised.

10.3 Governance as Security Stack

Governance provides a security stack:

Level	Description
Constitutional	Fundamental invariants that cannot be overridden
Vocabulary	Concern definitions and relationships
Schema	Structural requirements for each concern
Artifact	Individual artifact validation

Higher levels constrain lower levels; security properties cascade down.

10.4 Security as Compliance

When security invariants are governance-enforced:

$$\text{Security} = \text{Governance Compliance}$$

- Compliant artifacts satisfy security requirements
- Non-compliant artifacts cannot be ratified
- Security verification is artifact validation
- Security audit is governance audit

Security is not a separate concern; it is embedded in governance.

---

11. Comparison to Capability-Based Security

11.1 Capability-Based Security Model

Capability-based security restricts access through unforgeable tokens that combine designation (what resource) with authority (what may be done).

Key principles:

- **No ambient authority:** Authority only through explicit capability
- **Designation + authority:** Capabilities combine what and how
- **Attenuation only:** Capabilities can be narrowed but not widened
- **Unforgeable:** Capabilities cannot be manufactured

11.2 Protocol Governance as Capability Security

Protocol governance implements capability security at architectural scale:

Capability Principle	Protocol Implementation
No ambient authority	I-S2: Authority from artifacts only
Explicit designation	CC_ artifacts designate operations
Explicit authority	AC_ artifacts declare permissions
Attenuation	Workflows may constrain, not expand
Unforgeability	Artifacts require governance ratification

11.3 Key Differences

Scale of operation: - Capability systems operate at object/resource scale - Protocol governance operates at architectural/system scale

Mechanism of authority: - Capability systems transfer tokens at runtime - Protocol governance declares authority in ratified artifacts

Verification timing: - Capability systems verify at access time - Protocol governance validates at ratification time

11.4 Complementary Relationship

Protocol governance and capability security are complementary:

- Protocol governance bounds what capabilities may exist
- Capability security enforces capability discipline at runtime
- Both eliminate ambient authority
- Both enable least-authority design

Protocol governance provides the constitutional framework within which capability discipline operates.

---

12. Security Failure Modes

12.1 Structural Risk Categories

Protocol governance does not eliminate all security risk. It relocates risk to governance:

Failure Mode	Description	Risk Category
Vocabulary inflation	Adding too many concerns, expanding attack surface	Governance
Overbroad CS_ definitions	Side effects with excessive scope	Governance
Governance capture	Malicious actors gain governance authority	Governance
Binding misconfiguration	Wrong implementations bound to contracts	Configuration
Artifact tampering	Modification of ratified artifacts	Infrastructure

12.2 Governance as Primary Risk Vector

Governance failures become the primary security risk vector:

- If governance is compromised, semantic authority is compromised
- If vocabulary expands without discipline, attack surface grows
- If CS_ scopes are too broad, mutation is under-constrained

This shifts security discipline toward constitutional integrity.

12.3 Mitigations

Failure Mode	Mitigation
Vocabulary inflation	Constitutional amendment requirement; high barrier to vocabulary change
Overbroad CS_	Scope review in governance; least-privilege validation
Governance capture	Multi-party governance; separation of authorities
Binding misconfiguration	Binding validation; environment-specific governance

Failure Mode	Mitigation
Artifact tampering	Cryptographic integrity; signature verification

12.4 The Remaining Attack Surface

What attack surface remains?

- **Protocol artifact attacks:** Compromising artifact storage, distribution, or loading
- **Governance process attacks:** Manipulating authoring and approval processes
- **Execution faithfulness attacks:** Making execution deviate while appearing faithful
- **Data attacks:** Compromising confidentiality or integrity within authorized operations

This reduced surface is defensible with focused measures.

---

13. What This Paper Does Not Claim

This paper does not:

- **Prove cryptographic safety:** Cryptographic mechanisms are assumed, not specified
- **Eliminate all vulnerabilities:** Implementation bugs, side channels, and physical attacks remain possible
- **Replace authentication mechanisms:** Identity verification is orthogonal to behavioral governance
- **Remove need for network security:** Transport security remains necessary
- **Eliminate insider threats:** Malicious actors with governance authority remain a risk

The paper claims **structural bounding of behavioral authority**—that the vocabulary of possible behavior is finite, declared, and governable.

---

14. Structural Consequences

14.1 Bounded Behavioral Surface

The system's behavioral surface is bounded:

$$\text{Possible Behavior} \subseteq V$$

All possible behavior is expressible in vocabulary. Behavior outside vocabulary is structurally impossible.

14.2 Enumerated Mutation Pathways

All mutation pathways are enumerable:

$$\text{Mutation Pathways} = \text{CS\_}$$

Security analysis can enumerate every operation that may change state.

14.3 Explicit Authority Declaration

All authority is explicitly declared:

$$\text{Authority} = \text{Derive}(\text{AC\_}, \text{WF\_})$$

No implicit or ambient authority exists.

14.4 Deterministic Trace Evidence

All execution produces trace evidence:

$$\text{Execution} \Rightarrow \text{Trace}$$

Every operation is attributable and auditable.

14.5 Replaceable Implementation Trust

Trust is implementation-independent:

$$\text{Trust} = f(\text{Artifacts}) \neq f(\text{Implementation})$$

Security guarantees derive from artifacts, not execution engines.

Security becomes an architectural property, not a defensive overlay.

---

15. Relationship to Paper 7

Paper 7 bounded mutation [Bachi, 2026g]. Paper 8 shows why bounded mutation inverts trust.

15.1 Foundation Dependency

Without Paper 7's mutation boundary:

- Mutation surface would be unbounded
- CT_/CS_ separation would not exist
- Replay semantics would be undefined
- Attack surface quantification would be impossible

Paper 7 provides the structural foundation for security claims.

15.2 From Structure to Security

Paper 7 established:

- CT_ is pure; CS_ mutates
- Mutation surface is finite
- Composition is explicit

Paper 8 derives:

- Pure operations cannot attack state
- Finite mutation surface bounds attack surface
- Explicit composition eliminates hidden pathways

Security claims are structural, not rhetorical.

---

16. Conclusion

Protocol-governed systems invert trust by relocating authority from implementation code to ratified behavioral law.

This paper established seven invariants governing security architecture:

Invariant	Statement
I-S1	No behavior may occur outside the declared concern vocabulary
I-S2	Authority must be derivable from explicit AC_ artifacts and workflow declarations
I-S3	All state changes must occur through enumerated side-effect capabilities
I-S4	All invocation paths must originate from ratified workflow declarations
I-S5	Every mutation and computation step is attributable to a versioned artifact and actor
I-S6	Security guarantees derive from protocol artifacts, not execution implementations
I-S7	The root of system trust is constitutional governance

These invariants yield structural security properties:

- **Vocabulary closure** bounds behavioral possibility
- **No ambient authority** eliminates confused deputy attacks
- **Bounded mutation surface** enables attack surface quantification
- **No undeclared invocation** eliminates logic injection
- **Total trace accountability** enables forensic certainty
- **Implementation non-authority** makes execution a low-trust utility
- **Governance-rooted trust** concentrates security in constitutional artifacts

The relationship between papers establishes the complete security model:

> **Governance (Paper 4) makes behavior admissible. Protocol (Paper 5) declares behavioral law. Execution (Paper 6) enforces behavioral law deterministically. Mutation Boundary (Paper 7) separates computation from state change. Security (this paper) emerges from structural constraints.**

The inversion of trust transforms security from an unbounded problem (secure all execution pathways) to a bounded problem (secure protocol artifacts and ensure execution faithfulness). This transformation is architectural, not procedural.

Trust shifts from code to constitution. Security becomes a structural property, not a defensive overlay.

License and Use

Author Information

Bachi (aka Bhash Ganti) Contact: bachipeachy@gmail.com

Conflict of Interest: The author is developing commercial implementations of the described architecture.

References

Bachi aka Bhash Ganti (2026a). Protocol-Governed Systems: An architectural foundation for the AI era. *Zenodo Working Paper. DOI: https://doi.org/10.5281/zenodo.18715516*

Bachi aka Bhash Ganti (2026b). Protocol-Governed Systems: A constitutional realization of Turing-complete systems. *Zenodo Working Paper. DOI: https://doi.org/10.5281/zenodo.18718409*

Bachi aka Bhash Ganti (2026c). The Layer-Concern Constitutional Model: A formal structural taxonomy for protocol-governed systems. *Zenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18719589*

Bachi aka Bhash Ganti (2026d). Governance and Authoring: The legislative process of behavioral law. *SZenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18929868* .

Bachi aka Bhash Ganti (2026e). Protocol as Law: Behavioral specification and versioned authority. *Zenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18930048.*

Bachi aka Bhash Ganti (2026f). Deterministic Enforcement: Runtime binding, execution, and trace conformance. *Zenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18930314.*

Bachi aka Bhash Ganti (2026g). Pure Computation and Governed Mutation: Capability transforms and side effects in protocol-governed systems. *Zenodo Working Paper. DOI: https://zenodo.org/doi/10.5281/zenodo.18930423.

Dennis, J.B. and Van Horn, E.C. (1966). Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155.

Lampson, B.W. (1974). Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24.

Miller, M.S., Yee, K.-P., and Shapiro, J. (2003). Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University.

Saltzer, J.H. and Schroeder, M.D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.

Shapiro, J.S., Smith, J.M., and Farber, D.J. (1999). EROS: A fast capability system. *ACM SIGOPS Operating Systems Review*, 33(5):170–185.

Thompson, K. (1984). Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763.

Watson, R.N.M., Woodruff, J., Neumann, P.G., et al. (2015). CHERI: A hybrid capability-system architecture for scalable software compartmentalization. *IEEE Symposium on Security and Privacy*, pages 20–37.