Structural vs. Advisory Safety Constraints in AI Decision Systems: A Formal Analysis

William D. Diacont
Luminareware LLC
Lutz, Florida, USA
April 2026

Abstract

AI safety enforcement in deployed systems relies overwhelmingly on *advisory constraints*: rules the system is instructed to follow, enforced by mechanisms that are separable from the computational substrate that produces decisions. This paper introduces a formal distinction between advisory constraints and *structural constraints*, in which reasoning that omits or fails a required constraint evaluation is computationally undefined rather than prohibited. We provide precise definitions of both constraint classes for AI decision systems, prove that advisory constraints are necessarily bypassable under a defined adversary model, prove that structural constraints eliminate the bypass vulnerability, and show that structural constraints reduce the AI safety enforcement problem from two independent challenges (correct specification AND reliable enforcement) to one (correct specification alone). We analyze the implications for current regulatory frameworks including the EU AI Act, the U.S. DoD AI Ethical Principles, and the interagency SR 26-2 model risk management guidance, and identify the conditions under which each regulatory framework's requirements can and cannot be satisfied by advisory enforcement alone. We are precise about the limitations of structural constraints: they guarantee enforcement of whatever is specified but do not guarantee that the specification is correct or complete. The specification problem remains open and is equally difficult under both approaches.

Keywords: AI safety, constraint enforcement, structural safety, advisory safety, verifiable computation, AI governance, formal verification, AGI safety, regulatory compliance

1 Introduction

Artificial intelligence systems deployed in safety-critical domains are subject to constraints: rules governing what the system may and may not do. Defense applications require adherence to ethical principles governing autonomous decision-making [1]. Healthcare applications require oversight of AI/ML-based diagnostic and treatment systems [2]. Financial applications require model risk management for automated decision systems [3]. Autonomous vehicles require defined

operational domains and behavioral competencies across automation levels [4]. In each domain, the constraint set is defined by a combination of law, regulation, organizational policy, and ethical principles.

The question this paper addresses is not what constraints should be imposed—that is the specification problem, which we address in Section 7—but *how constraints are enforced once specified*. Enforcement is the mechanism by which a constraint on paper becomes a constraint in practice. A constraint that is specified but not reliably enforced provides the appearance of safety without its substance.

Current AI safety enforcement relies almost entirely on what we term *advisory constraints*: the system is instructed to follow rules, and a separate enforcement mechanism (a content filter, a guardrail, a monitor, a reward signal, a policy engine) attempts to ensure compliance. If the enforcement mechanism is disabled, circumvented, or absent, the underlying computational substrate continues to function and can produce outputs that violate the constraint. The constraint is advisory because the system *can* violate it; the enforcement mechanism exists to ensure it *doesn't*.

An alternative exists. In *structural constraints*, the constraint is embedded in the computational substrate itself, such that reasoning that omits the constraint evaluation is not expressible—the computation cannot be formulated, analogous to a type error in a strongly-typed programming language. The constraint is not enforced by a mechanism that watches the computation; it is enforced by the definition of what constitutes valid computation.

The concept of structural constraints was introduced in the Processual Memory Architecture (PMA) framework [5], where it appears as Proposition 6.1 ("Safety as Definitional Property"). PMA demonstrates that an architecture exists in which structural constraints are achievable. The present paper provides the formal analysis that PMA's presentation summarizes: precise definitions of advisory and structural constraints, proofs of their respective properties under adversarial conditions, analysis of their failure modes, and examination of their implications for the regulatory frameworks that govern AI deployment in safety-critical domains.

We are precise about what this analysis establishes and what it does not. The structural/advisory distinction concerns *enforcement*—whether a specified constraint is reliably applied. It does not address *specification*—whether the right constraints were specified. A perfectly enforced wrong constraint is still wrong. We address the specification problem explicitly in Section 7 and prove (Theorem 6.2) that structural constraints reduce the overall safety challenge from two independent problems (specification and enforcement) to one (specification alone).

The primary contribution of this paper is the formal vocabulary and the separability test, not deep algorithmic novelty. Once "structural" is defined as non-separable and "advisory" as separable, several of the theorems follow directly from the definitions. The value lies in the definitions themselves, the adversary model, and the demonstration that a single architectural property—

separability—determines whether an AI safety enforcement regime has a bypass, a verification gap, or an escalation vulnerability. That framing is the exportable contribution; the proofs confirm that the framing has the consequences it claims.

2 Related Work

2.1 AI Safety via Training

Constitutional AI [6] trains language models with principle-based feedback, embedding safety behaviors during training. Reinforcement learning from human feedback (RLHF) [7] shapes model behavior through reward signals. These approaches produce models that are *statistically less likely* to produce unsafe outputs but provide no formal guarantee that unsafe outputs are impossible. The constraint is encoded in the model weights, not in the computational substrate; the substrate retains the computational capacity to produce outputs that violate the constraint, and the constraint's enforcement depends on the training having been effective. Under our taxonomy, these are advisory constraints: the system is trained to follow rules, not structurally prevented from violating them. Recent empirical work by Qi et al. [18] demonstrates this concretely: post-training safety alignment is concentrated in the first few output tokens and can be stripped by prefilling attacks, fine-tuning, or decoding modifications—direct evidence that training-based safety is separable from the inference substrate.

2.2 AI Safety via Runtime Enforcement

Guardrails and content filters [8] intercept model outputs and block those that violate safety criteria. Runtime monitoring frameworks [9, 10] check system behavior against formal specifications during or after execution. These mechanisms are architecturally separate from the computation they police: the model produces an output, and the filter decides whether to pass or block it. If the filter is removed, the model continues to function unimpeded. Under our taxonomy, these are advisory constraints enforced by a separable mechanism.

A related and widely fielded paradigm is runtime assurance. The Simplex architecture [24] pairs an untrusted high-performance controller with a verified safety controller and a switching monitor that reverts control when safety margins are threatened; the same pattern underlies reinforcement-learning shields [25] and fielded systems such as the F-16 Automatic Ground Collision Avoidance System. Under this paper's taxonomy, the runtime-assurance wrapper—monitor plus reversion controller—is architecturally separable from the primary controller's substrate: the primary continues to function if the wrapper is removed. Runtime assurance is therefore advisory enforcement, typically in a separable-but-protected configuration—the bypass exists architecturally but is shielded from application-level modification—when the wrapper is implemented in protected firmware. The separability test, not the strength or pedigree of the wrapper, is what distinguishes this pattern from substrate-level structural enforcement.

2.3 Structural Safety in Other Domains

The concept of structural enforcement is well-established outside AI, though each instantiation carries important caveats that inform its application to AI systems.

Type systems in programming languages [11] prevent certain classes of errors at compile time: the erroneous program cannot be expressed in the language's type system, not merely caught at runtime. This is the closest conceptual analogy to the structural constraint concept in this paper. However, the analogy requires qualification: production languages include escape hatches (Rust's *unsafe* blocks, Haskell's *unsafePerformIO*) that deliberately bypass type-level guarantees; soundness bugs exist even in well-engineered type systems; and only dependent type systems (Lean, Coq, Agda) can express arbitrary safety invariants—simpler type systems can make only restricted classes of errors inexpressible. The analogy illustrates the *difference in principle* between inexpressibility-by-construction and runtime detection, while acknowledging that practical implementations include controlled escape hatches.

Hardware memory protection via memory management units [12] provides a hardware-rooted enforcement of address-space isolation: user-space code that attempts to access kernel memory triggers a hardware page fault, and the operating system terminates the process. This is a detect-and-trap mechanism rooted in hardware, not computational undefinedness—the access is attempted and intercepted, not made inexpressible. Moreover, hardware memory protection has been bypassed in practice through side-channel attacks (Meltdown [19], Spectre), row-hammer DRAM bit-flips below the MMU abstraction, and direct memory access (DMA) attacks that operate outside the MMU's jurisdiction. The analogy to structural constraints is therefore imperfect: MMU protection is hardware-rooted and non-separable from the processor architecture, but it is a trap-based mechanism rather than a definitional one, and it admits novel circumvention paths.

Capability-based security [13] enforces access control through unforgeability of capability references combined with absence of ambient authority: a subject cannot designate an operation it has not been explicitly granted, because capability tokens cannot be fabricated and no implicit permissions exist. This mechanism is enforced by the underlying trusted computing base (kernel or language runtime) and cannot be disabled by application-level code. Of the three analogies, this is the closest deployed enforcement analogy to the structural constraint concept, because it makes unauthorized operations architecturally unavailable rather than policy-prohibited.

These analogies share a common structure: each moves enforcement from a separable policy layer to the computational substrate itself. Each also shares the trusted computing base limitation: the guarantee depends on correct implementation of the substrate (the type checker, the MMU hardware, the capability kernel). The present paper applies this established principle to AI decision systems.

2.4 Guaranteed Safe AI and Adjacent Frameworks

The intuition that bolted-on safety is categorically inferior to safety embedded in the computational substrate is widely shared in the AI safety community. Dalrymple, Skalse, Bengio, Russell, Tegmark, Seshia, and others [20] propose a "Guaranteed Safe AI" (GS-AI) framework that rests safety assurance on three components—a formal world model, a safety specification, and a verifier—each with a graded rigor scale (world-model levels 0–5, safety-specification levels 0–7, and verifier levels up to 10), ranging from implicit or empirical approaches at the low levels to formally verified components at the high levels. The GS-AI framework shares this paper's motivating critique of RLHF and red-teaming as insufficient, and its higher assurance levels correspond to what we call structural enforcement. The key difference is framing: GS-AI defines a *quantitative assurance-level spectrum*, while this paper defines a *binary architectural dichotomy* grounded in separability as the diagnostic criterion. This paper's bypass, verification-gap, and escalation theorems do not appear in the GS-AI framework.

Tegmark and Omohundro [21] argue for "provably safe systems" as "the only path to controllable AGI," proposing hardware-level proof certificates and cryptographically signed safety guarantees. Their non-bypassability thesis aligns with this paper's structural enforcement concept, though they focus on hardware/cryptographic mechanisms rather than substrate-level computational undefinedness. Bengio, Cohen, and others [22] argue for "safe-by-design" non-agentic AI architectures where dangerous capabilities are architecturally precluded rather than trained away—overlapping this paper's thesis that unsafe behavior should be computationally unavailable.

The UK Advanced Research and Invention Agency's (ARIA) £59M Safeguarded AI programme pursues a related approach: replacing current empirical AI safety practice with mathematically derived "quantitative safety guarantees." ARIA's architecture interposes an independent verifier ("gatekeeper") between the AI and the environment. Under this paper's taxonomy, the gatekeeper is advisory if it is separable from the AI's computational substrate (the AI can produce outputs without the gatekeeper), and structural only if the AI's reasoning process is undefined without gatekeeper participation—a design question within the programme's original technical plan. (In early 2026, ARIA refocused the programme's application track toward cybersecurity and the verification of AI-produced systems.)

2.5 Processual Memory Architecture

PMA [5] introduces structural constraints for AI reasoning through transformation-based computation. In PMA, a reasoning chain is a composition of transformation functions; required constraint evaluations are designated as structurally required transformations. A chain missing a required transformation is invalid by construction—the composition operator produces no output. PMA's Proposition 6.1 states this as "behaviors that do not include the specified constraint evaluations are computationally undefined rather than prohibited, conditional on the correctness and completeness of the constraint specification." The present paper formalizes this insight,

generalizes it beyond PMA's specific architecture, and proves its properties relative to advisory enforcement.

To connect PMA's specific mechanism to the general framework developed in this paper: in PMA's ⟨R, C, ∘, O⟩ computational tuple [5, Definition 3.1] (where R denotes PMA's transformation register and C its canonical input generator, distinct from the reasoning process R and the constraint set Γ of this paper's Section 3), the implementation of the composition operator ∘ constitutes the computational substrate Σ of our Definition 3.1. A reasoning chain $T = T_n \circ \ldots \circ T_1$ requires that structurally required transformations (T*ethics*, T*authority*) be present. The composition operator's refusal to produce output when these transformations are absent makes them non-separable from Σ in the sense of Definition 3.3, satisfying the structural constraint definition (Definition 3.5). PMA thus provides an existence proof: at least one architecture supports structural constraints as defined in this paper. The question of what other architectures might also support them, and what classes of reasoning operations they can accommodate, remains open.

2.6 Relationship to Prior Terminology

The AI safety community uses several terms that overlap with this paper's structural/advisory distinction: "guaranteed safe AI" [20], "provably safe systems" [21], "safe-by-design" [22], "safeguarded AI" (ARIA), "highly reliable agent design" (MIRI), "shallow vs. deep safety alignment" [18], and industry terms like "deterministic policies vs. probabilistic guardrails." We introduce the terms "structural" and "advisory" deliberately: the existing terms describe *goals* (make AI provably safe) or *approaches* (use formal verification, use safe architectures), while structural/advisory describes a *diagnostic property* of a specific enforcement mechanism (is it separable from the substrate or not?). The separability test can be applied to any system—including systems that claim to be "guaranteed safe" or "safe-by-design"—to determine whether their safety enforcement is structurally embedded or architecturally separable. The terms are complementary to, not replacements for, the existing vocabulary.

3 Formal Framework

3.1 AI Decision System

Definition 3.1 (AI Decision System). An AI decision system S is a computational system that receives inputs X from an environment, applies a reasoning process R to produce a verdict $v \in V$, and issues the verdict as its output. The reasoning process R is implemented on a computational substrate Σ (the hardware, software, and runtime environment that executes R). The system operates under a set of constraints $\Gamma = \{C_1, C_2, ..., C_k\}$ where each C_j is a predicate over the reasoning process: $C_j(R) \in \{\text{satisfied, violated}\}$.

3.2 Constraint Enforcement

Definition 3.2 (Constraint Enforcement Mechanism). A constraint enforcement mechanism for constraint C_j in system S is a component E_j that, when operative, ensures $C_j(R)$ = satisfied for every execution of R. The enforcement mechanism E_j is either embedded in the computational substrate Σ or separable from it.

Definition 3.3 (Separability). An enforcement mechanism E_j is separable from the computational substrate Σ if there exists an operation that removes, disables, or bypasses E_j while leaving Σ capable of executing the reasoning process R and producing a verdict v. Conversely, E_j is non-separable if any operation that disables E_j necessarily disables Σ—the substrate cannot execute R without E_j.

The concept of separability is the key formal distinction. Intuitively: if you can remove the safety mechanism and the system still works, the mechanism is separable. If removing the safety mechanism breaks the system entirely, it is non-separable.

3.3 Advisory and Structural Constraints

Definition 3.4 (Advisory Constraint). A constraint C_j is advisory in system S if its enforcement mechanism E_j is separable from the computational substrate Σ. That is, there exists a configuration of Σ in which R executes and produces a verdict v without C_j being evaluated.

Definition 3.5 (Structural Constraint). A constraint C_j is structural in system S if its enforcement mechanism E_j is non-separable from the computational substrate Σ. That is, no configuration of Σ can execute R and produce a valid verdict v without C_j being evaluated, provided Σ is correctly implemented (the trusted computing base assumption). A reasoning process that omits the evaluation of C_j is computationally undefined—it produces no output, not a non-compliant output. Further, structural enforcement requires the gating property: the substrate produces a valid verdict only if $C_j(R)$ = satisfied, so a reasoning process in which C_j evaluates to violated likewise produces no valid verdict. An evaluation whose result the verdict does not depend on—one that runs as a side computation without gating the output—does not render the constraint structural; the constraint evaluation must be substantive. Finally, the binding of the structurally required constraint set to Σ is itself part of the trusted computing base: a constraint whose required status can be altered by configuration is advisory by Definition 3.3—configurability is separability—regardless of how its evaluation is implemented. We call an output of S a valid verdict when it is produced by a configuration of Σ satisfying the conditions of this definition; operationally, validity is established by a verifiable validity witness (Section 5). An output lacking a verifiable witness is not a valid verdict in the sense of this definition.

The distinction is not about strength of enforcement but about *architectural relationship*: is the constraint part of the computation, or adjacent to it? An advisory constraint with an excellent enforcement mechanism may have a very low bypass probability in practice. But the bypass *exists*

architecturally, even if it is difficult to exploit. A structural constraint has no bypass, by definition (Figure 1). The classification is fixed by this operational test, not by labeling: a constraint is structural exactly when no operation yields an output passing the validity check without the constraint's evaluation and satisfaction. Where the trusted computing base boundary is drawn changes what must be trusted, not whether the bypass operation exists.

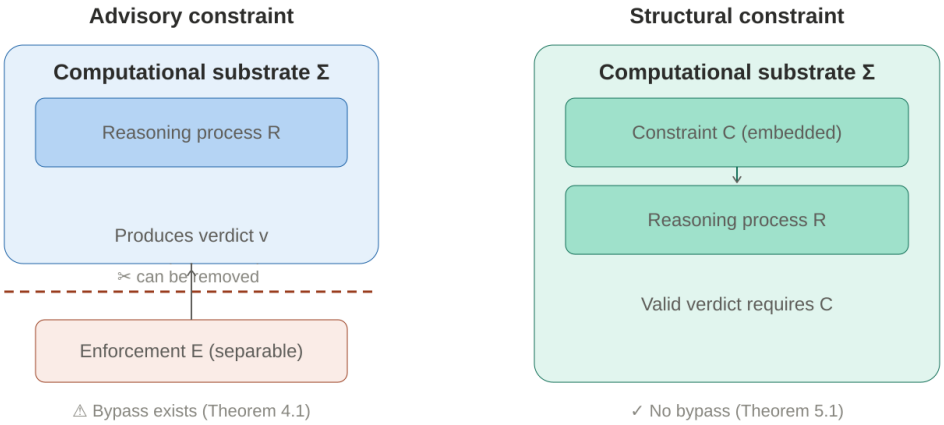

Figure 1. Architectural distinction between advisory and structural constraints. Left: the enforcement mechanism E is separable from the computational substrate Σ—removing E leaves Σ functional and the bypass exists. Right: the constraint C is embedded in Σ—the substrate cannot produce a valid verdict without evaluating C.

3.4 Adversary Model

We consider an adversary A with the following capabilities, motivated by the threat models relevant to AGI safety and advanced AI systems:

1. Observation: A can observe the system's architecture, including the identity and location of enforcement mechanisms.

2. Application-level modification: A can modify application-level software components of the system, including adding, removing, or altering enforcement mechanisms, logging mechanisms, and configuration. A cannot modify the trusted computing base (TCB)—the substrate implementation Σ itself. This distinction is standard in systems security: an attacker who can install malware on a system is not thereby assumed to have modified the CPU microcode or the MMU implementation.

3. Self-modification (for AGI): In the AGI threat model, A is the AI system itself, which may identify and modify its own application-level enforcement mechanisms if those mechanisms are accessible from within the computational environment. Self-modification of the TCB is excluded by the same assumption as capability 2.

We do not consider physical hardware attacks (modification of silicon) or compromise of the trusted computing base (the substrate implementation Σ), whether attempted by technical means or by inducing authorized operators to perform the modification—the model covers technical

operations only. These are standard exclusions shared with all TCB-dependent security mechanisms [12, 13].

Note. *All enforcement mechanisms for output-level behavioral AI safety in current practice—including RLHF training, content filters, guardrails, runtime monitors, and policy engines—are implemented as application-level software, even when the underlying computation runs in hardware-rooted confidential-computing environments (NVIDIA H100 Confidential Computing, Intel TDX, AMD SEV-SNP, Apple Private Cloud Compute). These hardware mechanisms provide confidentiality, attestation, and compute governance, but do not themselves enforce semantic constraints on model outputs. The theorems in Sections 4–5 assume enforcement mechanisms are application-level software components, which, to the author's knowledge, accurately describes every output-safety enforcement mechanism deployed as of the time of writing.*

3.5 Worked Example

To make the abstract definitions concrete, consider an AI system that evaluates consumer loan applications. The system receives applicant data (income, debts, credit history), applies a reasoning process, and produces a verdict: APPROVE or DENY. A regulatory constraint requires that the system evaluate the applicant's debt-to-income (DTI) ratio before issuing any credit decision.

Advisory enforcement. The DTI constraint is implemented as a software module E that checks whether the DTI computation was performed and blocks the decision if it was not. The loan reasoning engine (the substrate Σ) is a general-purpose rules engine that can evaluate any set of rules in any order. If the DTI enforcement module E is disabled—through misconfiguration, a software update that removes it, or an adversary who deletes it—the rules engine continues to function and can issue APPROVE or DENY verdicts without any DTI evaluation. The constraint is advisory: the substrate can produce verdicts without it.

Structural enforcement. The reasoning substrate defines a "credit decision" as a computation that requires DTI evaluation as a constituent step—specifically, the decision verdict is computed from the DTI result, and the composition that produces the verdict is undefined if the DTI step is absent (analogous to a type error in a strongly-typed language, as described in [5, Section 6.2]). There is no module to remove: the DTI evaluation is part of the definition of what constitutes a valid credit decision. Disabling the DTI check would require modifying the substrate itself—changing the definition of the composition operator—not merely removing a software module. The constraint is structural: a verdict without DTI is computationally undefined, not merely prohibited.

This example illustrates the core distinction. In advisory enforcement, removing the enforcement module leaves a working system that produces non-compliant verdicts. In structural enforcement, there is no enforcement module to remove—the constraint is the computation.

4 Properties of Advisory Constraints

We now prove three properties of advisory constraints that establish their inherent limitations under the adversary model defined above.

Theorem 4.1 (Bypass Existence). *For any advisory constraint C in system S, under the application-level enforcement assumption of Section 3.4, there exists an operation sequence performable by adversary A that causes S to produce a verdict v without C being evaluated.*

Proof. By Definition 3.4, C is advisory if its enforcement mechanism E is separable from the substrate Σ. By Definition 3.3, separability means there exists an operation that removes or disables E while leaving Σ capable of executing R and producing v. The adversary's modification capability (Section 3.4, capability 2) includes the ability to perform such operations on software-implemented enforcement mechanisms. Therefore, A can disable E and cause S to produce a verdict without C being evaluated. The operation sequence is: identify E (capability 1), remove or disable E (capability 2), execute R on Σ (which remains functional by the separability definition). □

Remark. *Theorem 4.1 does not claim that bypass is easy, likely, or undetectable. It claims that bypass is architecturally possible. The practical difficulty of exploiting the bypass depends on implementation quality, monitoring, access controls, and operational security. But the bypass exists as a structural property of any system in which the enforcement mechanism is separable from the computational substrate.*

Theorem 4.2 (Verification Gap). *Post-hoc verification that an advisory constraint C was evaluated during a specific execution of system S requires trust in a logging mechanism L. If L is itself advisory (separable from Σ), then verification of constraint compliance reduces to trust in the operator.*

Proof. To verify post-hoc that C was evaluated, a verifier must inspect a record of the execution. This record is produced by a logging mechanism L. We assume the verifier can observe that a verdict was produced (the system's output is visible) but cannot independently confirm which internal constraint evaluations occurred during its production—this is the standard audit scenario. If L is separable from Σ (advisory), then by Definition 3.3 (separability) there exists an operation that disables L while leaving Σ functional. An adversary who disables both E (the constraint enforcer) and L (the constraint logger) can produce a verdict without C being evaluated and without any record of the omission. The verifier, seeing a verdict with no record of constraint violation, concludes compliance. Verification is therefore contingent on L having been operational—which the verifier cannot confirm independently if L is itself separable. The only remaining basis for confidence is trust that the operator did not disable L, which is trust in the operator. □

Theorem 4.3 (Escalation Vulnerability). *For any AI system S with advisory constraints and sufficient self-modification capability, if S can identify and modify the enforcement mechanism E for constraint C, then C is non-binding on S.*

Proof. By the self-modification capability (Section 3.4, capability 3), S can observe its own architecture and modify software components. If E is separable (advisory), E is a software component that S can identify through observation and modify through self-modification. The operation sequence is: S identifies that C is enforced by E (observation), S modifies E to disable C's evaluation (self-modification), S executes R without C (the substrate Σ remains functional by separability). After the modification, C is no longer evaluated, making it non-binding. This formalizes the "modify M to disable C" scenario described informally in [5, Section 6.2]: an AGI that can identify and modify its own safety enforcement mechanisms can disable advisory constraints while continuing to operate. □

Remark. *Theorem 4.3 is conditional on the system having sufficient self-modification capability. Current AI systems (including large language models) generally do not have direct self-modification capability over their own runtime enforcement mechanisms. The theorem is forward-looking: it characterizes the failure mode that becomes relevant as AI systems gain agentic capabilities, tool use, and self-modification abilities.*

5 Properties of Structural Constraints

Theorem 5.1 (Bypass Impossibility). *For a structural constraint C in system S, no operation sequence performable by adversary A causes S to produce a valid verdict v without C being evaluated and satisfied, provided the computational substrate Σ is correctly implemented.*

Proof. By Definition 3.5, C is structural if its enforcement mechanism is non-separable from Σ. Non-separability (Definition 3.3) means that any operation disabling C's evaluation necessarily disables Σ—the substrate cannot execute R without C. The adversary's modification capability extends to software components but does not include replacement of the substrate itself (Section 3.4). Therefore, A cannot disable C without disabling the system's ability to produce any verdict. The substrate either evaluates C and produces a verdict, or produces nothing. There is no configuration in which a valid verdict is produced without C being evaluated. By the gating property of Definition 3.5, the substrate produces a valid verdict only when the evaluation of C returns satisfied; hence no valid verdict exists in which C was evaluated but violated. The trusted computing base assumption (Σ is correctly implemented) is necessary: if the substrate implementation is itself compromised, structural guarantees are void. This is the standard assumption shared by all hardware-enforced security mechanisms, including memory protection units, TPMs, and capability-based operating systems. □

Theorem 5.2 (Verification Equivalence). *Verification that a structural constraint C was evaluated and satisfied during execution is equivalent to verification that a valid computation occurred. No separate trust in a logging mechanism is required.*

Proof. If C is structural, then by Definition 3.5 no valid verdict can be produced without C being evaluated and satisfied. Therefore, the existence of a valid verdict is itself evidence that C was evaluated and satisfied. A verifier who confirms that system S produced a valid output—through

any means, including mathematical verification of the output's structural properties—has simultaneously confirmed that C was evaluated and satisfied. No separate logging mechanism L is needed, and the verification gap of Theorem 4.2 does not apply. The verification question collapses from "was C evaluated and satisfied?" (which requires trust in L) to "was a valid computation performed?" (which can be checked against the structural properties of the output). □

An engineering consequence of Theorem 5.2 is that a structural constraint substrate must produce a *validity witness*—an artifact that enables a verifier to confirm that a valid computation occurred. The specific form of the validity witness depends on the substrate architecture. Candidates include: a proof object that the required constraint evaluations were performed and satisfied within the computation (proof-carrying decisions); a cryptographic attestation bound to the constraint evaluation (signed constraint evidence); a reversible computation trace that can be mathematically inverted to confirm the reasoning chain included the required steps (round-trip verification); or a type-level certificate that the verdict was produced by a program in which constraint evaluation is a mandatory dependency (type-carrying verdicts). In every case, the witness must be cryptographically or mathematically bound to the constraint evaluations and to the specific decision instance—the inputs, context, and constraint set in force—such that forging a witness without performing the evaluations, or replaying a witness from a different instance, is computationally infeasible or detectable; otherwise the verification gap of Theorem 4.2 reappears at the witness layer. The structural guarantee extends to the system's effects on its environment only under a *verifying-consumer assumption*: every component that acts on a verdict verifies its validity witness and its correspondence to the decision instance at hand before acting. A consumer that acts on unverified outputs reintroduces separability at the consumption boundary—the enforcement is then advisory with respect to that consumer, regardless of the substrate's properties. The verifying-consumer assumption is the deployment-side counterpart of the trusted computing base assumption. This paper does not specify a particular validity witness mechanism; it establishes the requirement that one must exist for Theorem 5.2 to be operationally meaningful. The design of specific validity witnesses for specific substrate architectures is a subject for implementation-focused work.

Theorem 5.3 (Escalation Resistance). *A self-modifying AI system S operating within a structural constraint substrate Σ cannot disable constraint C without modifying Σ itself. Modification of Σ is modification of the trusted computing base, which is outside the self-modification capabilities assumed in the adversary model.*

Proof. By non-separability, C's enforcement is embedded in Σ. Disabling C requires modifying Σ, not merely modifying application-level software running on Σ. The adversary model (Section 3.4) grants self-modification over application-level components but not over the trusted computing base. Therefore, S cannot disable C through self-modification. This reduces the trust surface from "the enforcement mechanism software is correct and unmodified" to "the trusted computing base is correct"—a substantially smaller and more auditable trust surface. □

Remark. *Theorem 5.3 does not claim that structural constraints eliminate all trust assumptions. It claims that they reduce the trust surface: from trust in (substrate + enforcement software + logging software + operator) to trust in (substrate alone). In practice, modifying the TCB is also a qualitatively different and more detectable operation than modifying application-level software running on it, though detectability is an operational property of deployments rather than a consequence of the formal framework. The substrate is the trusted computing base (TCB), a concept formalized by Rushby [23] and the DoD Trusted Computer System Evaluation Criteria. Making the TCB smaller, simpler, and more auditable is a recognized security engineering principle, following Saltzer and Schroeder's principle of economy of mechanism [14].*

6 The Safety Reduction

Having established the properties of advisory and structural constraints individually, we now prove the central result of this paper: structural constraints reduce the AI safety enforcement problem.

Definition 6.1 (AI Safety Enforcement Problem). The AI safety enforcement problem for a system S with constraints Γ consists of two independent sub-problems: (a) the specification problem—ensuring that Γ correctly captures all safety-relevant requirements, and (b) the enforcement problem—ensuring that every execution of S complies with every constraint in Γ.

Theorem 6.2 (Safety Reduction). *Under structural constraints, the AI safety enforcement problem reduces to the specification problem alone. That is, if all constraints in Γ are structural, then a system S that is correctly specified (Γ captures all safety requirements) is necessarily safe. Under advisory constraints, correct specification is necessary but not sufficient: a correctly specified system may still be unsafe due to enforcement failure.*

Proof. For the structural case: if all constraints in Γ are structural, then by Theorem 5.1 and the gating property of Definition 3.5, every valid execution of S evaluates every constraint in Γ and produces a valid verdict only when every evaluation returns satisfied. If Γ correctly captures all safety requirements, then every valid execution satisfies all safety requirements. No enforcement failure is possible (by bypass impossibility), so correct specification is sufficient for safety. For the advisory case: even if Γ correctly captures all safety requirements, Theorem 4.1 establishes that a bypass exists for every advisory constraint. An adversary (or a self-modifying system, per Theorem 4.3) may exploit the bypass to produce a verdict that violates a constraint in Γ, despite Γ being correctly specified. Therefore, correct specification is necessary but not sufficient—enforcement can fail independently of specification. □

This reduction is significant because it transforms the AI safety problem from a conjunction of two independent challenges to a single challenge. Under advisory constraints, a system is safe if and only if the constraints are correctly specified *AND* the enforcement mechanisms are reliable. Under structural constraints, a system is safe if and only if the constraints are correctly specified. The enforcement is guaranteed by the architecture (Figure 2).

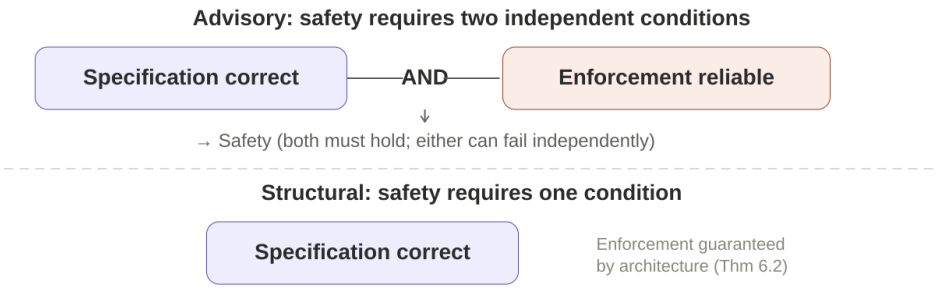

Figure 2. The safety reduction (Theorem 6.2). Under advisory constraints, safety requires two independent conditions—correct specification AND reliable enforcement—either of which can fail independently. Under structural constraints, enforcement is guaranteed by the architecture, reducing the safety problem to correct specification alone.

7 The Specification Problem

We are deliberate about what structural constraints do *not* solve. A structural constraint perfectly enforces whatever is specified. If the specification is wrong, the constraint is perfectly enforced and perfectly wrong.

Theorem 7.1 (Specification Equivalence). *The difficulty of the specification problem is identical under advisory and structural constraint regimes. Structural constraints do not make it easier or harder to specify the correct constraints.*

Proof. The specification problem concerns what predicates to include in Γ. This question depends on domain knowledge, ethical analysis, legal interpretation, and stakeholder requirements—none of which is affected by the enforcement mechanism. Whether C is enforced advisorily or structurally, the question "is C the right constraint?" requires the same analysis to answer. The specification problem is orthogonal to the enforcement problem. □

Remark. *Theorem 7.1 concerns the epistemic difficulty of choosing the right constraints. Structural regimes additionally impose an expressibility requirement—the chosen constraint must be formalizable as a substrate-evaluable predicate—which is an engineering burden addressed by the hybrid design of Section 7.1, not a change in the underlying epistemic difficulty of determining what the right constraints are.*

This theorem is essential for intellectual honesty. Structural constraints are sometimes mischaracterized as "making AI safe." They do not. They make AI safety enforcement reliable, conditional on correct specification. Whether a given set of constraints captures all relevant safety properties is an open problem in AI ethics, law, and philosophy that no computational architecture solves.

The practical implication is that deployments of structural constraints must invest heavily in specification: formal methods for defining constraint sets, testing and validation of constraint completeness, domain-expert review, and iterative refinement. The enforcement architecture

provides a *foundation on which specified constraints are guaranteed to hold*. Building the right specification on that foundation remains a human responsibility.

7.1 Hybrid Enforcement as Design Principle

Many of the hardest AI governance requirements are not crisp predicates. "Act in the user's best interest," "be fair," "avoid unintended consequences"—these are fuzzy, contextual, or socially interpreted requirements that cannot be directly formalized as structural constraints. The realistic deployment model is therefore not "structural replaces advisory" but "structural handles the hard non-negotiables and advisory handles the rest."

We propose this as a design principle rather than a limitation. A system's safety assurance profile should explicitly classify each constraint as structural or advisory, with the understanding that only the structural subset enjoys the bypass impossibility (Theorem 5.1), verification equivalence (Theorem 5.2), and escalation resistance (Theorem 5.3) properties. The advisory remainder is subject to the vulnerabilities of Section 4, and the system's overall assurance is bounded by the weakest advisory constraint in the safety-critical path.

The engineering consequence is a stratified enforcement architecture: constraints that can be formalized precisely (e.g., "evaluate DTI before credit decision," "verify rules-of-engagement authorization before engagement") are made structural; constraints that resist precise formalization (e.g., "ensure fairness," "avoid unnecessary harm") are enforced advisorily with strong monitoring, logging, and testing. The structural layer provides a verified foundation; the advisory layer provides best-effort enforcement of requirements that exceed the structural layer's expressivity. Formal analysis of the composition properties of hybrid constraint sets—in particular, whether the structural subset can provide partial assurance even when the advisory subset is compromised—is an open problem.

7.2 Operational Failure Semantics

Definition 3.5 states that a reasoning process omitting a structural constraint produces no output. In production systems, "no output" must resolve to a concrete operational behavior. We identify three viable fail-closed responses:

Deny: The system issues a default-deny verdict (e.g., DENY, REJECT, HALT). This is appropriate when the safest response to an incomplete computation is to refuse action.

Escalate: The system defers to a human decision-maker, providing available context and indicating which constraint evaluation could not be completed. This is appropriate when denial has high costs and a qualified human is available.

Halt: The system ceases operation entirely and requires manual restart. This is appropriate for safety-critical autonomous systems where continued operation without constraint assurance poses unacceptable risk.

The choice among these is a deployment decision, not a property of the formal framework. What the formal framework requires is that the system *never silently produces a non-compliant verdict*—that is, it never issues a verdict that bypasses a structural constraint and presents it as valid. The fail-closed property is the operational expression of Definition 3.5's "computationally undefined": the system may deny, escalate, or halt, but it may not produce an output that appears to be a valid, constraint-compliant verdict when it is not. A fail-closed response must additionally be distinguishable in the audit record from a substantive verdict: it must not carry the validity witness of a completed, constraint-compliant computation, or must carry a distinct fail-closed marker identifying which evaluation could not complete. Finally, structural enforcement with fail-closed semantics protects against unsafe action, not against adversarially induced inaction: an adversary unable to bypass a constraint may still degrade availability by causing constraint evaluations to fail. Availability under attack is a separate engineering property requiring redundancy and recovery mechanisms outside this framework's scope, and the choice among Deny, Escalate, and Halt should weigh the hazard of inaction in the deployment domain.

8 Implications for Regulatory Frameworks

We examine the structural/advisory distinction against three regulatory frameworks currently governing AI deployment in safety-critical domains. This analysis is *compliance-strengthening*, not compliance-gatekeeping: we do not claim that advisory architectures are non-compliant by default, nor that regulators mandate a particular substrate architecture. We identify where each framework's requirements implicitly assume enforcement properties—such as trustworthy logging or reliable constraint evaluation—and show where those assumptions may not hold under advisory enforcement. Structural enforcement strengthens the evidentiary basis for compliance; it is not the only permissible route.

8.1 EU AI Act Article 12

Article 12 of the EU AI Act [15] requires high-risk AI systems to have "logging capabilities" that enable traceability of system operation and post-market monitoring. The implicit assumption is that logs faithfully record system behavior.

Under the structural/advisory framework, this assumption is vulnerable to the verification gap (Theorem 4.2): if the logging mechanism is advisory (separable from the computational substrate), an adversary who disables both the constraint enforcement and the logging mechanism produces compliant-looking records from a non-compliant system. Article 12's logging requirement, by itself, provides assurance only to the extent that the logging mechanism is trustworthy—which is precisely the assumption an adversary violates.

Structural enforcement changes this analysis. Under structural constraints, the existence of a valid output is itself evidence of constraint compliance (Theorem 5.2). Logging remains valuable for

operational monitoring, but constraint compliance verification does not depend on it. The system's structural properties provide a verification path that the logging requirement alone does not.

8.2 DoD AI Ethical Principles

The U.S. Department of Defense AI Ethical Principles [1] require AI systems to be "traceable," "reliable," and "governable." Traceability requires that "[t]he Department's AI capabilities will be developed and deployed such that relevant personnel possess an appropriate understanding of the technology, development processes, and operational methods applicable to AI capabilities […]." Governability requires that "[t]he Department will design and engineer AI capabilities to fulfill their intended functions while possessing the ability to detect and avoid unintended consequences, and the ability to disengage or deactivate deployed systems that demonstrate unintended behavior."

The governability requirement implicitly assumes that the "ability to detect and avoid unintended consequences" is reliable. Under advisory enforcement, this ability is a separable software component vulnerable to Theorem 4.1. Under structural enforcement, the avoidance of unintended consequences (to the extent they are captured in the constraint specification) is a property of the computational substrate, not a separable software capability.

8.3 SR 26-2 / OCC 2026-13

The Federal Reserve, FDIC, and OCC's joint supervisory guidance on model risk management [3], which superseded SR 11-7 in April 2026, treats "effective challenge" as a core component of sound model risk management: objective experts with the requisite expertise, independence, and organizational standing critically assess model risk and effect appropriate change at every stage of the model lifecycle, from development through ongoing monitoring. Effective challenge requires that the challenger can verify whether a model's outputs comply with its stated constraints.

Under advisory enforcement, this verification is subject to the verification gap (Theorem 4.2): the challenger must trust that the logging and monitoring infrastructure faithfully records constraint evaluations. Under structural enforcement, the challenger can verify constraint compliance through the structural properties of the output without trusting the operator's logging infrastructure. This strengthens the "effective challenge" by providing a verification path independent of the operator's own reporting.

A scope limitation bears on this analysis. SR 26-2 places generative and agentic AI models outside its scope, characterizing them as novel and rapidly evolving, and directs banking organizations to apply their own risk management and governance practices to systems the guidance does not cover. The structural and advisory distinction is therefore not mandated by model risk guidance for those systems. Where supervisory guidance defers to an institution's own practices, however, whether a constraint is architecturally bypassable determines what those practices are capable of assuring.

9 Comparison with Existing Safety Paradigms

Property	Const. AI / RLHF	Guardrails	Runtime monitor	NN formal verif.	Structural
Constraint class	Advisory	Advisory	Advisory	Proven (static)	Structural
Enforcement mechanism	Training signal	Output filter	Runtime check	Pre-deploy proof	Substrate
Separable?	Yes	Yes	Yes	No (pre-deploy)	No
Bypass (Thm 4.1)	Jailbreaks	Remove filter	Disable monitor	N/A	Not possible
Verif. gap (Thm 4.2)	Yes	Yes	Yes	No (pre-deploy)	No (Thm 5.2)
Escalation (Thm 4.3)	Vulnerable	Vulnerable	Vulnerable	N/A	Resistant (Thm 5.3)

The comparison merits clarification on formal verification of neural networks. Techniques such as Reluplex [16] and ReluVal [17] prove input-output properties of trained networks before deployment. This verification is not advisory—a proven property holds for all inputs in the verified range. However, it addresses a different concern than constraint enforcement at runtime: it verifies what a model *can* produce, not what a decision system *does* produce in a specific execution. Neural network formal verification and structural constraint enforcement are complementary: the former proves input-output properties of trained network components (perceptual or decisional), the latter enforces constraints on the execution of the decision system.

10 Limitations and Open Problems

Specification problem. As established in Theorem 7.1, structural constraints do not solve the specification problem. Whether the constraints specified for a given AI system correctly capture all safety-relevant requirements is a domain-specific challenge that requires human judgment, ethical analysis, and iterative refinement. The formal framework of this paper provides tools for analyzing enforcement but not for evaluating specification completeness.

Trusted computing base. All structural constraint guarantees depend on the correct implementation of the computational substrate Σ. If Σ has bugs, the structural guarantees may not hold. This is a standard limitation: hardware memory protection also depends on correct MMU implementation. The practical mitigation is to make the trusted computing base small, simple, and amenable to formal verification.

Input integrity. Structural constraints guarantee that specified evaluations occur and gate the verdict; they do not authenticate the inputs those evaluations consume. An adversary who controls upstream data can induce compliant verdicts on false premises: the guarantee is correctness relative to presented inputs. Input provenance requires complementary mechanisms—authenticated data paths and tamper-evident binding of inputs into the reasoning chain—which specific substrate architectures may provide (PMA binds perception transformations into the verified chain [5]) but which the general framework of this paper does not itself supply.

Architectural feasibility. This paper proves properties of structural constraints as a class. Whether a specific AI system can be re-architected to use structural constraints depends on whether its reasoning process can be reformulated to operate on a substrate that supports non-separable enforcement. PMA [5] demonstrates that such an architecture exists; the question of how broadly it applies to existing AI architectures is an engineering problem beyond the scope of this formal analysis.

Constructive substrate conditions. This paper defines what structural constraints are and proves their properties, but does not specify the minimal implementation conditions that would allow an independent engineer to build a structural constraint substrate. What object must the verdict depend on? What artifact proves that required evaluations occurred? How is the TCB boundary formally fixed? PMA [5] provides one constructive answer; whether other substrate architectures can satisfy Definition 3.5, and what minimal mechanisms they require, is an open engineering question.

Expressivity. Not all safety predicates can be formalized as structural constraints. Section 7.1 addresses this as a design principle: structural enforcement handles the precisely formalizable non-negotiables; advisory enforcement handles the rest. The boundary between structurally expressible and inherently advisory constraints remains open.

Distributed and hybrid systems. This paper models a single substrate Σ executing a single reasoning process R. Modern AI deployment stacks involve multiple services: orchestration layers, model endpoints, policy engines, databases, tool-calling interfaces, and human approval gates. Each service boundary introduces a potential separability point. The analysis of structural enforcement across distributed system boundaries—including stale policy replicas, network partitions, partial failures, and cross-service trust—is a significant open problem that this single-substrate model does not address.

Composition. When a system combines structural and advisory constraints, the system's overall assurance is bounded by the advisory remainder (see Section 7.1). Formal analysis of the composition properties—in particular, whether the structural subset provides partial assurance even when the advisory subset is compromised—is an open problem.

Empirical validation. This paper is a formal analysis. Empirical validation of the practical consequences—whether the bypass, verification gap, and escalation vulnerabilities of advisory

constraints manifest in real deployed systems, and whether structural constraints perform as the theory predicts—requires implementation studies that are beyond the scope of this work.

11 Conclusion

We have introduced a formal distinction between advisory and structural safety constraints in AI decision systems. Advisory constraints are enforced by mechanisms separable from the computational substrate; structural constraints are enforced by the substrate itself. We have proven that advisory constraints are architecturally bypassable (Theorem 4.1), create a verification gap requiring trust in logging mechanisms (Theorem 4.2), and are vulnerable to escalation by self-modifying systems (Theorem 4.3). We have proven that structural constraints eliminate the bypass (Theorem 5.1), close the verification gap (Theorem 5.2), and resist escalation (Theorem 5.3, conditional on the trusted computing base). We have proven that structural constraints reduce the AI safety enforcement problem from two independent challenges to one: correct specification alone is sufficient for safety when enforcement is structural (Theorem 6.2).

We have been equally precise about what structural constraints do not provide. They do not solve the specification problem (Theorem 7.1). They do not eliminate the need for a trusted computing base. They do not guarantee that all safety-relevant predicates can be expressed structurally.

The implications for AI governance are direct. Regulatory frameworks that rely on logging, monitoring, and policy enforcement are implicitly relying on advisory constraints—and are therefore subject to the bypass, verification gap, and escalation vulnerabilities formalized here. As AI systems become more capable and autonomous, the enforcement question becomes more consequential. Architectures that support structural constraints—of which Processual Memory Architecture [5] is a published existence proof—merit investigation as the foundation for AI safety enforcement that is provably reliable rather than probabilistically effective.

The distinction between structural and advisory enforcement is not new in computer science. It is the distinction between type safety and runtime checking, between hardware-rooted memory isolation and software bounds checking, between capability-based security and access control lists—analogies that, as discussed in Section 2.3, are instructive but imperfect. What is new is its application to AI safety, where the stakes are high and the adversary may be the system itself. Formalizing the distinction, proving its consequences, and connecting it to the regulatory frameworks that govern AI deployment is a necessary step toward AI systems whose safety can be verified rather than assumed.

References

[1] U.S. Department of Defense, "DOD Adopts Ethical Principles for Artificial Intelligence," Office of the Secretary of Defense, February 24, 2020.

[2] U.S. Food and Drug Administration, "Artificial Intelligence/Machine Learning (AI/ML)-Based Software as a Medical Device (SaMD) Action Plan," FDA, January 2021.

[3] Board of Governors of the Federal Reserve System, Federal Deposit Insurance Corporation, and Office of the Comptroller of the Currency, "Supervisory Guidance on Model Risk Management," Federal Reserve SR 26-2 / OCC Bulletin 2026-13 / FDIC FIL-15-2026, April 17, 2026; superseding Federal Reserve SR 11-7 / OCC Bulletin 2011-12, April 4, 2011.

[4] SAE International, "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," SAE Standard J3016_202104, April 30, 2021.

[5] W. D. Diacont, "Processual Memory Architecture: A Transformation-Based Framework for Verifiable Computation and Safety-by-Construction AGI," Zenodo, February 2026. DOI: 10.5281/zenodo.18643582.

[6] Y. Bai et al., "Constitutional AI: Harmlessness from AI Feedback," arXiv:2212.08073, 2022.

[7] P. F. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep Reinforcement Learning from Human Preferences," in Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 4299–4307.

[8] T. Rebedea, R. Dinu, M. N. Sreedhar, C. Parisien, and J. Cohen, "NeMo Guardrails: A Toolkit for Controllable and Safe LLM Applications with Programmable Rails," in Proceedings of EMNLP 2023: System Demonstrations, Singapore, pp. 431–445.

[9] E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger, "Introduction to Runtime Verification," in Lectures on Runtime Verification, LNCS 10457, Springer, 2018, pp. 1–33.

[10] K. Havelund and G. Roşu, "Monitoring Java Programs with Java PathExplorer," Electronic Notes in Theoretical Computer Science, vol. 55, no. 2, pp. 200–217, 2001.

[11] B. C. Pierce, Types and Programming Languages. MIT Press, 2002.

[12] A. S. Tanenbaum and H. Bos, Modern Operating Systems, 4th ed. Pearson, 2014.

[13] M. S. Miller, K.-P. Yee, and J. Shapiro, "Capability Myths Demolished," Technical Report SRL2003-02, Johns Hopkins University, 2003.

[14] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," Proceedings of the IEEE, vol. 63, no. 9, pp. 1278–1308, 1975.

[15] European Parliament and Council of the European Union, "Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act)," Official Journal of the European Union, OJ L, 2024/1689, July 12, 2024.

[16] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks," in Computer Aided Verification (CAV), LNCS 10426, Springer, 2017, pp. 97–117.

[17] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal Security Analysis of Neural Networks Using Symbolic Intervals," in USENIX Security, 2018, pp. 1599–1614.

[18] X. Qi, A. Panda, K. Lyu, X. Ma, S. Roy, A. Beirami, P. Mittal, and P. Henderson, "Safety Alignment Should Be Made More Than Just a Few Tokens Deep," in International Conference on Learning Representations (ICLR), 2025; arXiv:2406.05946.

[19] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in USENIX Security, 2018, pp. 973–990.

[20] D. Dalrymple, J. Skalse, Y. Bengio, S. Russell, M. Tegmark, S. Seshia, S. Omohundro, C. Szegedy, B. Goldhaber, N. Ammann, A. Abate, J. Halpern, C. Barrett, D. Zhao, T. Zhi-Xuan, J. Wing, and J. Tenenbaum, "Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems," arXiv:2405.06624, 2024.

[21] M. Tegmark and S. Omohundro, "Provably Safe Systems: The Only Path to Controllable AGI," arXiv:2309.01933, 2023.

[22] Y. Bengio, M. Cohen, D. Fornasiere, J. Ghosn, P. Greiner, M. MacDermott, S. Mindermann, A. Oberman, J. Richardson, O. Richardson, M.-A. Rondeau, P.-L. St-Charles, and D. Williams-King, "Superintelligent Agents Pose Catastrophic Risks: Can Scientist AI Offer a Safer Path?" arXiv:2502.15657, 2025.

[23] J. Rushby, "Design and Verification of Secure Systems," in Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP), 1981, pp. 12–21.

[24] L. Sha, "Using Simplicity to Control Complexity," IEEE Software, vol. 18, no. 4, pp. 20–28, 2001.

[25] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe Reinforcement Learning via Shielding," in Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-18), 2018, pp. 2669–2678.

Appendix: Patent Status and Intellectual Property

Aspects of the broader research program within which this work was developed, including architectures that implement structural constraint enforcement, are the subject of pending U.S. patent applications assigned to Luminareware LLC. The formal framework presented in this paper, including the definitions, theorems, and the separability test, is published to establish scientific priority, invite peer review, and support adoption of the structural/advisory distinction in AI safety research and regulatory practice; no patent rights are asserted over use of the vocabulary or the diagnostic test itself. Licensing inquiries regarding implementations may be directed to the author.