ZelC: A Cybersecurity-Native Programming Language for Intent-Gated, Capability-Scoped, Evidence-Generating Operations in the Agentic AI Era

Technical Report

Haja Mo
Rocheston
research@rocheston.com

February 10, 2026

Abstract

Security engineering has shifted from "write software, deploy once" to continuous, adversarial operations: detect, correlate, contain, recover, and prove. General-purpose languages (GPLs) can automate these workflows, but they model security actuation as ordinary side effects of arbitrary code, leaving safety, governance, and evidentiary integrity to convention. This paper presents ZelC, a cybersecurity-native language and runtime in which *(i)* observation is separated from actuation ("kinetic execution"), *(ii)* authority is capability-scoped rather than ambient, *(iii)* actuation is constrained by machine-checkable intent contracts with formal blast-radius analysis, *(iv)* evidence artifacts are generated as a semantic output of execution and optionally anchored into verifiable append-only logs, and *(v)* information-flow control via taint tracking prevents untrusted data from reaching sensitive operational sinks.

We present the complete formal model: abstract syntax, static and dynamic semantics via typing rules and small-step operational semantics, a capability algebra with monotonic non-escalation proofs, intent contracts formalized as constraint-satisfaction problems, blast-radius bounding via abstract interpretation, a cryptographic evidence chain with Merkle-tree anchoring and inclusion-proof verification, and a conformance predicate that provides agent-agnostic safety guarantees. We include concrete ZelC program listings for ransomware containment, cloud key rotation, brute-force defense, CI/CD integrity verification, and zero-trust network gating to demonstrate that the formal model maps directly to operational reality.

We explicitly distinguish foundational primitives—capability security [1, 2], effect typing [3, 4], admission-control policy gates [5–7], information-flow control [10, 16, 17], tamper-evident logging [11–13], Merkle-tree transparency logs [14, 15], proof-carrying code [18], and abstract interpretation [9]—from ZelC's novel contribution: an *integrated programming model* that binds these mechanisms into one coherent operational semantics for cybersecurity operations in the agentic AI era.

Keywords: cybersecurity operations · incident response · policy-as-code · capability security · effect systems · information-flow control · tamper-evident logs · Merkle trees · proof-carrying code · agentic AI safety · abstract interpretation · domain-specific languages

Contents

1 Introduction

1.1 Operational Context

Modern security work is continuous operations under time pressure: isolate endpoints, rotate keys, revoke sessions, quarantine artifacts, snapshot disks, notify responders, and produce evidence that stands up to auditing and legal scrutiny. The correctness criterion is not only "did the action execute," but "was it authorized, bounded, reviewable, and provable."

General-purpose languages model execution as a sequence of steps with broad ambient authority: a process executing a script may implicitly hold credentials, filesystem rights, network access, or cloud tokens. In such environments, "rotate a key" and "delete production data" may differ only by parameters, not by language semantics. Consider the Python fragment:

```python
import boto3
ec2 = boto3.client('ec2')
ec2.terminate_instances(InstanceIds=['i-prod-critical'])
# No safety gate. No evidence. No blast-radius check.
# Production is gone.
```

There is no structural distinction between this catastrophic action and a benign health-check. The language offers no mechanism to express intent, enforce blast-radius limits, or generate evidence.

1.2 Design Thesis: The Kinetic Gap

We define the *kinetic gap* as the mismatch between the operational demands of security actuation and how typical programming languages structure authority, effects, and evidence.

Definition 1.1 (Kinetic Gap)**.** The kinetic gap $\mathcal{G}$ is the four-dimensional mismatch between operational security requirements and language-level semantics:

$$\mathcal{G} = \langle \mathcal{G}_{\text{act}}, \mathcal{G}_{\text{auth}}, \mathcal{G}_{\text{gov}}, \mathcal{G}_{\text{ev}} \rangle \tag{1}$$

where each component represents a distinct semantic deficiency:

$$\begin{aligned} \mathcal{G}_{\text{act}} &: \text{ Side effects are not isolated from analytic computation.} \\ \mathcal{G}_{\text{auth}} &: \text{ Permissions exist as ambient privilege, not explicit objects.} \\ \mathcal{G}_{\text{gov}} &: \text{ Blast-radius constraints are external to code semantics.} \\ \mathcal{G}_{\text{ev}} &: \text{ Logs are mutable, optional, and frequently incomplete.} \end{aligned} \tag{2}$$

ZelC's core thesis: cybersecurity operations deserve a *language-level semantics* that makes actuation explicit, bounded, and evidentiary by construction. Formally, ZelC aims to close $\mathcal{G}$ such that each component is reduced to $\perp$ (null gap) under its operational semantics:

$$[\![\textsc{ZelC}]\!] \models \mathcal{G}_{\text{act}} = \mathcal{G}_{\text{auth}} = \mathcal{G}_{\text{gov}} = \mathcal{G}_{\text{ev}} = \perp \tag{3}$$

1.3 Scope and Contributions

This paper makes the following contributions:

C1 Formal language model. We define ZelC's abstract syntax, type system with a two-point effect lattice ($\mathsf{RO} \sqsubseteq \mathsf{KI}$), and small-step operational semantics with evidence generation as a side-product of kinetic transitions.

C2 Capability algebra. We formalize capability injection, attenuation, and the monotonic non-escalation invariant, proving that authority at any program point is bounded by initial injection.

C3 Intent contracts and blast-radius analysis. We model intents as constraint-satisfaction problems and provide a sound static blast-radius checker via abstract interpretation.

C4 Cryptographic evidence chain. We formalize the hash chain, signature blocks, Merkle-tree anchoring, and inclusion-proof verification, proving evidence completeness for well-typed programs.

C5 Agent-agnostic safety. We define the conformance predicate and prove that the same safety guarantees hold regardless of whether code is human-authored or AI-generated.

C6 Taint tracking for operational sinks. We specialize information-flow control for SOC actuation sinks (firewall rules, IAM revocations, process termination) with evidence-generating sanitization.

C7 Concrete operational examples. We present complete ZelC programs for ransomware containment, cloud key rotation, brute-force defense, CI/CD integrity, and zero-trust gating.

2 Prior Work and Positioning

This section enumerates major adjacent prior art to pre-empt overclaiming. In each case, we state the relationship precisely: ZelC *uses* the concept as an internal mechanism but does not claim its invention.

2.1 Capability-Based Security

Capability-based security was introduced by Dennis and Van Horn [1] and developed extensively in the object-capability model by Miller [2]. Authority is represented by unforgeable tokens, contrasted with ACL-driven ambient authority; capabilities implement least privilege and avoid confused-deputy attacks.

ZelC uses the capability concept as a *language invariant*: kinetic actions are only possible when the relevant capability objects are present and validated. ZelC does not claim to invent capabilities.

2.2 Effect Systems

Effect typing makes side effects explicit at the type level, supporting disciplined separation of pure computation from effectful computation. Lucassen and Gifford [3] introduced the first effect system; Koka [4] is a well-known example of row-polymorphic effect types.

ZelC specializes the effect boundary for security operations by introducing "kinetic" actuation as the *primary effect of interest*. ZelC does not claim to invent effect typing.

2.3 Policy-as-Code and Admission Control

Kubernetes admission controllers [5] evaluate create/update/delete requests and can reject unsafe changes before state is written. OPA [6] formalizes policy evaluation prior to admission. Terraform Sentinel [7] implements policy-as-code gating for infrastructure changes.

ZelC internalizes gating into the *language semantics* of kinetic blocks and applies it to multi-domain security actuation.

2.4 Information-Flow Control and Taint Tracking

Denning [10] introduced the lattice model for secure information flow. TaintDroid [16] is a canonical system-wide dynamic taint tracking system. Jif [17] is a security-typed language extension for Java supporting IFC.

ZelC applies taint/IFC to operational sinks (firewall rules, IAM revocations, deletions) and couples violations to intent rejection and evidence emission.

2.5 Tamper-Evident Logging

Schneier and Kelsey [11] describe mechanisms to detect log manipulation. Snodgrass et al. [12] address tamper detection in audit logs. RFC 5848 [13] specifies signed syslog messages.

ZelC's claim is that evidence generation is a *semantic consequence* of kinetic execution.

2.6 Transparency Logs, Merkle Trees, and Ledgers

Certificate Transparency [14] logs are append-only and publicly auditable. Sigstore Rekor [15] provides a transparency log with signed tree heads. Amazon QLDB [20] provides cryptographically verifiable immutable transaction history.

ZelC provides a standardized evidence object model and anchoring interface consistent with transparency-log properties.

2.7 Proof-Carrying Code and Static Verification

Proof-carrying code [18] attaches machine-checkable proofs to untrusted code. The eBPF verifier [8] statically analyzes programs before execution.

ZelC borrows the "verify before execute" philosophy, applying it to kinetic constraints and capability proofs.

2.8 Abstract Interpretation

Cousot and Cousot [9] introduced abstract interpretation as a theory of sound approximation of program semantics. ZelC uses abstract interpretation for compile-time blast-radius bounding.

2.9 Constrained Language Model Programming

LMQL [19] formalizes constraints over LLM output. ZelC *complements* constrained generation by enforcing constraints at the actuation layer, even if the textual output of an agent is imperfect.

3 Core Formal Model

3.1 Semantic Domains

Definition 3.1 (Semantic Domains)**.** The ZelC semantic model is parameterized over the following domains:

$$\Sigma \in \mathbf{State} \qquad \text{Machine state} \qquad (4)$$

$$\mathcal{C} \subseteq \mathbf{Cap} \qquad \text{Capability set} \qquad (5)$$

$$\mathcal{I} \in \mathbf{Intent} \qquad \text{Intent contract} \qquad (6)$$

$$P \in \mathbf{Prog} \qquad \text{ZelC program} \qquad (7)$$

$$\mathcal{A}(P) \subseteq \mathbf{Act} \qquad \text{Kinetic actions in } P \qquad (8)$$

$$\mathcal{E} \subseteq \mathbf{Ev} \qquad \text{Evidence trace} \qquad (9)$$

$$H : \{0,1\}^* \to \{0,1\}^{256} \qquad \text{Hash (SHA-256)} \qquad (10)$$

$$\text{Sign} : \mathbf{SK} \times \{0,1\}^* \to \mathbf{Sig} \qquad \text{Digital signature} \qquad (11)$$

$$\text{Verify} : \mathbf{PK} \times \{0,1\}^* \times \mathbf{Sig} \to \{\mathit{true}, \mathit{false}\} \qquad \text{Signature verify} \qquad (12)$$

We additionally define the *state transition function* $\delta : \mathbf{State} \times \mathbf{Act} \to \mathbf{State}$ such that $\delta(\Sigma, a(\vec{v}))$ yields the successor state after executing action a with arguments $\vec{v}$.

3.2 Type Universe

ZelC provides both standard types and a rich set of *semantic security types* that distinguish security-relevant values at the type level:

Definition 3.2 (Type Universe)**.** The type universe $\mathcal{T}$ is defined by the following grammar:

$$\tau ::= \mathbf{unit} \mid \mathbf{Bool} \mid \mathbf{Int} \mid \mathbf{Float} \mid \mathbf{Text} \mid \mathbf{Duration} \qquad (13)$$

$$\mid \mathbf{IP} \mid \mathbf{CIDR} \mid \mathbf{Hash} \mid \mathbf{UserId} \mid \mathbf{CVE} \mid \mathbf{ControlId} \qquad (14)$$

$$\mid \mathbf{RiskScore} \mid \mathbf{IOCSet} \mid \mathbf{Incident} \mid \mathbf{Timeline} \mid \mathbf{EvidencePack} \qquad (15)$$

$$\mid \mathbf{List}[\tau] \mid \mathbf{Map}[\tau_k, \tau_v] \mid \mathbf{Record}\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad (16)$$

$$\mid \tau_1 \to^{\varepsilon} \tau_2 \qquad (17)$$

where $\varepsilon \in \{\mathsf{RO}, \mathsf{KI}\}$ is the effect annotation on function types.

Remark 3.3 (Semantic Types Prevent Confusion)**.** Unlike GPLs where an IP address is a `String`, in ZelC the expression `firewall block ip "Hello World"` is a *compile-time type error*: the `block` action requires an argument of type **IP**, not **Text**. This eliminates an entire class of injection and parameter-confusion bugs at the type level.

3.3 Abstract Syntax

Definition 3.4 (Full Abstract Syntax)**.** The abstract syntax of ZelC is given by:

Programs:

$$P \;::=\; \textbf{check}\ \mathit{Name}\ \{\ S\ \} \;\mid\; \textbf{define}\ f(\vec{x})\ \{\ S\ \} \;\mid\; \textbf{intent}\ \mathit{Name}\ \{\ \mathcal{I}_{\text{body}}\ \} \;\mid\; P\,;P \tag{18}$$

Statements:

$$\begin{aligned} S \;::=\;& S\,;S \mid \textbf{set}\ x = E \mid \textbf{change}\ x = E \mid \textbf{keep}\ x = E && (19)\\ &\mid \textbf{when}\ E\ \{\ S\ \}\ [\textbf{otherwise}\ \{\ S\ \}] \mid \textbf{each}\ x\ \textbf{in}\ E\ \{\ S\ \} && (20)\\ &\mid \textbf{do}[\mathcal{I},\mathcal{C}]\ \{\ K\ \}\ \textbf{end} \mid \textbf{try}\ \{\ S\ \}\ \textbf{catch}\ e\ \{\ S\ \} && (21)\\ &\mid \textbf{evidence record}\ \mathit{title}\ \textbf{details}\ E \mid \textbf{audit log}\ \mathit{event}\ \textbf{details}\ E && (22)\\ &\mid \textbf{alert}\ \mathit{severity}\ \textbf{message}\ E \mid \textbf{notify}\ \mathit{channel}\ E \mid \textbf{stop} \mid \textbf{wait}\ E && (23) \end{aligned}$$

Kinetic Statements (only inside `do...end`):

$$\begin{aligned} K \;::=\;& \textbf{action}\ a(\vec{v}) \mid K\,;K \mid \textbf{each}\ x\ \textbf{in}\ E\ \{\ K\ \} && (24)\\ &\mid \textbf{when}\ E\ \{\ K\ \} \mid \textbf{evidence record}\ E \mid \textbf{audit log}\ E && (25) \end{aligned}$$

The **do** ... **end** block is the *kinetic boundary*: only kinetic statements K are permitted inside. Outside **do**, only analytic (read-only) statements are admitted. This is the structural enforcement mechanism that closes $\mathcal{G}_{\text{act}}$.

Definition 3.5 (Intent Contract Body)**.** The intent body $\mathcal{I}_{\text{body}}$ binds to a named intent declaration:

$$\begin{aligned} \mathcal{I}_{\text{body}} ::=\;& \textbf{category}\colon c \mid \textbf{risk\_level}\colon \ell \mid \textbf{constraints}\ \{\ C_{\text{list}}\ \} && (26)\\ C_{\text{list}} ::=\;& \textbf{max\_targets}\colon n \mid \textbf{time\_limit}\colon d\\ &\mid \textbf{requires\_approval}\colon a \mid \textbf{exclude\_scope}\colon S_{\text{list}} && (27) \end{aligned}$$

3.4 Effect Typing: Read-Only versus Kinetic

Definition 3.6 (Effect Lattice)**.** We define a two-point effect lattice:

$$\varepsilon \in \{\mathsf{RO},\ \mathsf{KI}\} \qquad \text{with ordering} \qquad \mathsf{RO} \sqsubseteq \mathsf{KI} \tag{28}$$

where RO denotes *read-only* (analytic) effects and KI denotes *kinetic* (state-changing) effects. The join operator is:

$$\varepsilon_1 \sqcup \varepsilon_2 \;=\; \begin{cases} \mathsf{RO} & \text{if } \varepsilon_1 = \varepsilon_2 = \mathsf{RO}\\ \mathsf{KI} & \text{otherwise} \end{cases} \tag{29}$$

Typing judgments take the form $\Gamma \vdash E : \tau \,!\, \varepsilon$, meaning expression E has type τ and effect ε under context Γ.

Definition 3.7 (Core Typing Rules)**.** The ZelC type system enforces the kinetic boundary via the following rules:

Rule (Var) — Variable lookup is read-only:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \ \vdash \ x : \tau \,!\, \mathsf{RO}} \ \textsc{Var} \tag{30}$$

Rule (Set) — Binding is read-only:

$$\frac{\Gamma \vdash E : \tau \,!\, \mathsf{RO}}{\Gamma \ \vdash \ \textbf{set}\ x = E : \textbf{unit} \,!\, \mathsf{RO}} \ \textsc{Set} \tag{31}$$

Rule (Act) — Kinetic actions carry effect KI:

$$\frac{a \in \textbf{Act}_{\mathsf{KI}} \qquad \Gamma \vdash \vec{v} : \vec{\tau} \,!\, \mathsf{RO}}{\Gamma \ \vdash \ \textbf{action}\ a(\vec{v}) : \textbf{unit} \,!\, \mathsf{KI}} \ \textsc{Act} \tag{32}$$

Rule (Do) — Kinetic blocks require authorization and intent satisfaction:

$$\frac{\Gamma \vdash K : \textbf{unit} \,!\, \mathsf{KI} \quad \mathrm{Auth}(\Sigma, \mathcal{C}, K) \quad \mathrm{Sat}(\mathcal{I}, \Sigma, K)}{\Gamma \vdash \textbf{do}[\mathcal{I}, \mathcal{C}]\ \{K\}\ \textbf{end} : \textbf{unit} \,!\, \mathsf{KI}} \ \textsc{Do} \tag{33}$$

Rule (Outside) — Non-kinetic code must remain read-only:

$$\frac{S \text{ outside any } \textbf{do} \text{ block} \quad \Gamma \vdash S : \tau \,!\, \varepsilon}{\varepsilon = \mathsf{RO}} \ \textsc{Outside} \tag{34}$$

Rule (Seq) — Sequential composition propagates effects:

$$\frac{\Gamma \vdash S_1 : \tau_1 \,!\, \varepsilon_1 \qquad \Gamma \vdash S_2 : \tau_2 \,!\, \varepsilon_2}{\Gamma \vdash S_1 \,;\, S_2 : \tau_2 \,!\, (\varepsilon_1 \sqcup \varepsilon_2)} \ \textsc{Seq} \tag{35}$$

Rule (When) — Conditional preserves effect:

$$\frac{\Gamma \vdash E : \textbf{Bool} \,!\, \mathsf{RO} \qquad \Gamma \vdash S_1 : \tau \,!\, \varepsilon_1 \qquad \Gamma \vdash S_2 : \tau \,!\, \varepsilon_2}{\Gamma \vdash \textbf{when}\ E\ \{S_1\}\ \textbf{otherwise}\ \{S_2\} : \tau \,!\, (\varepsilon_1 \sqcup \varepsilon_2)} \ \textsc{When} \tag{36}$$

Rule (Each) — Iteration propagates body effect:

$$\frac{\Gamma \vdash E : \textbf{List}[\tau'] \,!\, \mathsf{RO} \qquad \Gamma, x : \tau' \vdash S : \tau \,!\, \varepsilon}{\Gamma \vdash \textbf{each}\ x\ \textbf{in}\ E\ \{S\} : \textbf{unit} \,!\, \varepsilon} \ \textsc{Each} \tag{37}$$

Rule (Evidence) — Evidence emission is always read-only (it is a *semantic output*, not a state change):

$$\frac{\Gamma \vdash E : \textbf{Record}\{\ldots\} \,!\, \mathsf{RO}}{\Gamma \vdash \textbf{evidence record}\ \textit{title}\ \textbf{details}\ E : \textbf{unit} \,!\, \mathsf{RO}} \ \textsc{Ev} \tag{38}$$

Theorem 3.8 (Effect Soundness)**.** *If a well-typed* ZelC *program P satisfies $\Gamma \vdash P : \tau \,!\, \mathsf{RO}$, then P performs no kinetic actions. Conversely, any kinetic action $a(\vec{v})$ in P must occur inside a* **do** . . . **end** *block.*

Proof. By structural induction on the typing derivation. The only rule introducing effect KI is (Act), which requires $a \in \mathbf{Act}_{\mathsf{KI}}$. By rule (Outside), any statement outside a **do** block must have effect RO. Since $\mathsf{RO} \neq \mathsf{KI}$, rule (Act) cannot appear outside **do**. Rule (Do) is the only construct that promotes KI into the enclosing scope. Therefore, all kinetic actions are structurally confined to **do** blocks. □ □

4 Capability-Scoped Authority

4.1 Capability Algebra

Definition 4.1 (Capability)**.** A capability κ is an unforgeable token:

$$\kappa \;=\; \langle\, o,\; r,\; \mathit{ttl},\; \mathit{issuer},\; \mathit{sig} \,\rangle \;\in\; \mathbf{Cap} \tag{39}$$

where $o \in \mathbf{Obj}$ is the target object (endpoint, IAM identity, firewall rule set, S3 bucket, . . .), $r \subseteq \mathbf{Right}$ is the granted right set, $\mathit{ttl} \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ is the time-to-live, *issuer* is the granting principal, and *sig* is a digital signature for authenticity.

Definition 4.2 (Authorization Predicate)**.** The authorization predicate is:

$$\text{Auth} \;:\; \mathbf{State} \times 2^{\mathbf{Cap}} \times \mathbf{Act} \;\longrightarrow\; \{\mathit{true}, \mathit{false}\} \tag{40}$$

defined as:

$$\text{Auth}(\Sigma,\, \mathcal{C},\, a(\vec{v})) \;=\; \exists\, \kappa \in \mathcal{C} : \big[\, \text{obj}(\kappa) = \text{target}(a, \vec{v}) \;\wedge\; \text{reqRights}(a) \subseteq r(\kappa) \;\wedge\; \text{alive}(\kappa, \Sigma) \,\big] \tag{41}$$

where $\text{alive}(\kappa, \Sigma) \equiv (\Sigma.\mathit{time} \leq \kappa.\mathit{ttl}) \wedge \text{Verify}(\kappa.\mathit{issuer}.\mathit{pk},\ \text{serialize}(\kappa),\ \kappa.\mathit{sig})$.

4.2 Attenuation and Non-Escalation

Definition 4.3 (Attenuation Operator)**.** The attenuation operator produces a restricted capability:

$$\text{attenuate}(\kappa,\, r',\, \mathit{ttl}') \;=\; \kappa' \;=\; \langle\, o,\, r',\, \min(\mathit{ttl}, \mathit{ttl}'),\, \mathit{issuer},\, \text{Sign}(\mathit{sk}, \text{serialize}(\kappa'))\,\rangle \tag{42}$$

subject to: $r' \subseteq r$ and $\mathit{ttl}' \leq \mathit{ttl}$.

Axiom 4.4 (Monotonic Non-Escalation)**.** For any derived capability κ' from κ:

$$\text{rights}(\kappa') \;\subseteq\; \text{rights}(\kappa) \qquad \wedge \qquad \mathit{ttl}(\kappa') \;\leq\; \mathit{ttl}(\kappa) \tag{43}$$

This is enforced by construction in Definition 4.3.

Proposition 4.5 (Least-Privilege Bound)**.** *Under Axiom 4.4, the authority available at any program point p is bounded:*

$$\forall\, p \in P,\ \forall\, \kappa_p \in \mathcal{C}_p :\ \text{rights}(\kappa_p) \subseteq \bigcup_{\kappa_0 \in \mathcal{C}_{\text{init}}} \text{rights}(\kappa_0) \tag{44}$$

Proof. By induction on the derivation chain from $\mathcal{C}_{\text{init}}$ to $\mathcal{C}_p$. Each derivation step applies attenuate, which by Definition 4.3 can only reduce rights. Therefore the union of rights at p is bounded by the union of rights at initialization. □ □

Definition 4.6 (Capability Scope Binding)**.** Capabilities are explicitly injected into **do** blocks via lexical scoping:

$$\textbf{do}[\mathcal{I}, \mathcal{C}]\ \{K\}\ \textbf{end} \implies \mathcal{C} \text{ is immutable within } K \tag{45}$$

No capability can be fabricated, copied from ambient scope, or escalated within the kinetic block.

5 Intent Contracts and Blast-Radius Constraints

5.1 Intent as a Constrained Operational Contract

Definition 5.1 (Intent Contract)**.** An intent contract $\mathcal{I}$ is a seven-tuple:

$$\boxed{\mathcal{I} \;=\; \langle\, \textit{Allow},\ \textit{Deny},\ \textit{Limits},\ \textit{Window},\ \textit{Approvals},\ \textit{Redaction},\ \textit{ExcludeScope}\,\rangle} \tag{46}$$

where:

$$\begin{aligned}
\textit{Allow} &\subseteq \mathbf{Act} && \text{Permitted actions} && (47)\\
\textit{Deny} &\subseteq \mathbf{Act} && \text{Forbidden actions} && (48)\\
\textit{Limits} &= \langle L_{\text{tgt}}, L_{\text{rate}}, L_{\text{scope}}\rangle \in \mathbb{N}^3 && \text{Numerical bounds} && (49)\\
\textit{Window} &= [t_{\text{start}}, t_{\text{end}}] \subseteq \mathbb{R}_{\geq 0} && \text{Temporal bounds} && (50)\\
\textit{Approvals} &\in \{\mathbf{auto}, \mathbf{manual}(n), \mathbf{quorum}(k, n)\} && \text{HITL policy} && (51)\\
\textit{Redaction} &\in \mathbf{RedactionRule} && \text{Field masking} && (52)\\
\textit{ExcludeScope} &\subseteq \mathbf{Obj} && \text{Protected objects} && (53)
\end{aligned}$$

Definition 5.2 (Intent Satisfaction)**.** Intent satisfaction is a conjunction of five conditions:

$$\boxed{\text{Sat}(\mathcal{I},\ \Sigma,\ a(\vec{v})) \;\equiv\; \varphi_{\text{allow}} \;\wedge\; \varphi_{\text{deny}} \;\wedge\; \varphi_{\text{limits}} \;\wedge\; \varphi_{\text{window}} \;\wedge\; \varphi_{\text{scope}}} \tag{54}$$

where:

$$\begin{aligned}
\varphi_{\text{allow}} &\equiv\ a \in \textit{Allow} && (55)\\
\varphi_{\text{deny}} &\equiv\ a \notin \textit{Deny} && (56)\\
\varphi_{\text{limits}} &\equiv\ \text{BR}(\Sigma, a(\vec{v})) \leq L_{\text{tgt}}\ \wedge\ \text{rate}(\Sigma, a) \leq L_{\text{rate}} && (57)\\
\varphi_{\text{window}} &\equiv\ \Sigma.\textit{time} \in [t_{\text{start}}, t_{\text{end}}] && (58)\\
\varphi_{\text{scope}} &\equiv\ \text{Targets}(\Sigma, a(\vec{v})) \cap \textit{ExcludeScope} = \emptyset && (59)
\end{aligned}$$

Kinetic execution is permitted **only if** Sat $= \textit{true}$. Otherwise the runtime *fails closed* and emits a rejection evidence record.

5.2 Blast Radius Formalization

Definition 5.3 (Blast Radius Function)**.** Let Targets : $\mathbf{State} \times \mathbf{Act} \to 2^{\mathbf{Obj}}$ return the set of assets impacted by an action. The blast radius is:

$$\text{BR}(\Sigma,\ a(\vec{v})) \;=\; \big|\,\text{Targets}(\Sigma,\ a(\vec{v}))\,\big| \tag{60}$$

For composite kinetic blocks, blast radius is additive:

$$\mathrm{BR}(\Sigma,\ K_1\,;K_2) \;=\; \big|\,\mathrm{Targets}(\Sigma, K_1) \cup \mathrm{Targets}(\delta(\Sigma, K_1), K_2)\,\big| \tag{61}$$

For iteration over n elements:

$$\mathrm{BR}\Big(\Sigma,\ \textbf{each}\ x\ \textbf{in}\ [v_1, \ldots, v_n]\ \{\ \textbf{action}\ a(x)\ \}\Big) \;=\; \left|\bigcup_{i=1}^{n} \mathrm{Targets}(\Sigma_i, a(v_i))\right| \tag{62}$$

where Σ_i is the state after executing the first $i-1$ iterations.

5.3 Static Blast-Radius Bounding via Abstract Interpretation

Definition 5.4 (Abstract Domain for Blast Radius)**.** Define the abstract domain $\hat{D} = (\hat{\textbf{State}}, \sqsubseteq_{\hat{D}})$ where abstract states $\hat{\Sigma}$ map variables to intervals or finite sets. The abstraction and concretization functions form a Galois connection:

$$(\alpha, \gamma) : 2^{\textbf{State}} \rightleftharpoons \hat{\textbf{State}} \quad \text{with} \quad \alpha \circ \gamma \sqsupseteq \mathrm{id} \tag{63}$$

following Cousot and Cousot [9].

Definition 5.5 (Abstract Blast Radius)**.** The abstract blast-radius function $\hat{U}$ over-approximates BR:

$$\hat{U} : \hat{\textbf{State}} \times \textbf{Act} \to \mathbb{N}$$
$$\text{satisfying} \quad \forall\,\Sigma \in \gamma(\hat{\Sigma}) :\ \mathrm{BR}(\Sigma, a(\vec{v})) \;\le\; \hat{U}(\hat{\Sigma}, a(\vec{v})) \tag{64}$$

For the loop construct **each** x **in** *collection* $\{$ **action** $a(x)$ $\}$, if the abstract analysis determines $|collection| \le n$ (where n may be an interval upper bound), then:

$$\hat{U}(\hat{\Sigma},\ \textbf{each}\ x\ \textbf{in}\ C\ \{\textbf{action}\ a(x)\}) \le n \cdot \max_{v} \hat{U}(\hat{\Sigma}, a(v)) \tag{65}$$

Theorem 5.6 (Soundness of Static Blast-Radius Checking)**.** *If $\hat{U}$ is a sound over-approximation of* BR *(Definition 5.5), then:*

$$\hat{U}(\hat{\Sigma},\ a(\vec{v})) \le L_{\mathrm{tgt}} \quad\Longrightarrow\quad \mathrm{BR}(\Sigma,\ a(\vec{v})) \le L_{\mathrm{tgt}} \qquad \forall\,\Sigma \in \gamma(\hat{\Sigma}) \tag{66}$$

Conversely, if $\hat{U}(\hat{\Sigma}, a(\vec{v})) > L_{\mathrm{tgt}}$, compilation is conservatively rejected*.*

Proof. By the soundness property of the Galois connection (Equation 63), for every concrete state $\Sigma \in \gamma(\hat{\Sigma})$, the abstract evaluation over-approximates the concrete. Therefore $\mathrm{BR}(\Sigma, a(\vec{v})) \le \hat{U}(\hat{\Sigma}, a(\vec{v})) \le L_{\mathrm{tgt}}$. The rejection case follows by contrapositive: if the abstract bound exceeds L_{tgt}, there *may* exist a concrete state violating the limit, so conservative rejection is sound. □ □

Example 5.7 (Compile-Time Rejection)**.** Consider the ZelC fragment:

```
intent Isolate {
  constraints { max_targets: 5 }
}

check MassReboot
```

```
  set all_hosts = inventory.get_all()  -- |all_hosts| = 500
  do
    each host in all_hosts
      linux service "critical-db" restart  -- Û = 500 > 5 = L_tgt
    end
  end
end
```

Listing 1: Blast-radius violation detected at compile time.

The abstract interpreter determines $|all\_hosts| = 500$ and computes $\hat{U} = 500 > 5 = L_{\text{tgt}}$. Compilation is rejected:

$$\hat{U} = 500 \;>\; 5 = L_{\text{tgt}} \implies \textsc{Reject}\ (\text{“Blast radius exceeds limit by }99\times\text{”}) \tag{67}$$

6 Evidence-First Semantics

6.1 Evidence Object Model

Definition 6.1 (Evidence Record)**.** For each kinetic action instance i, ZelC emits a structured evidence record:

$$ev_i = \Big\langle \underbrace{id_i, ts_i, \pi_i}_{\text{provenance}},\ \underbrace{\mathcal{I}_{\text{id}}, fp(\kappa)}_{\text{auth}},\ \underbrace{a_i, H(\vec{v}_i)}_{\text{action}},\ \underbrace{H(\Sigma_{\text{pre}}), H(\Sigma_{\text{post}})}_{\text{state}},\ \underbrace{\rho_i, H(pol_i)}_{\text{result}},\ \underbrace{h_{i-1}}_{\text{chain}} \Big\rangle \tag{68}$$

where π_i is the executing principal, $fp(\kappa) = H(\text{serialize}(\kappa))$ is the capability fingerprint, $\rho_i \in \{success, failure, rejected\}$ is the action result, and h_{i-1} is the hash of the previous evidence record, forming the chain link.

Definition 6.2 (Evidence Record Schema (Formal))**.** Each field has a fixed type from the ZelC type universe:

$$ev : \textbf{Record}\left\{ \begin{array}{ll} id : \textbf{Hash}, & ts : \textbf{Int}, \\ principal : \textbf{UserId}, & intent\_id : \textbf{Text}, \\ cap\_fp : \textbf{Hash}, & action : \textbf{Text}, \\ args\_digest : \textbf{Hash}, & pre\_state : \textbf{Hash}, \\ post\_state : \textbf{Hash}, & result : \textbf{Text}, \\ policy\_digest : \textbf{Hash}, & prev\_hash : \textbf{Hash} \end{array} \right\} \tag{69}$$

6.2 Cryptographic Hash Chaining

Definition 6.3 (Hash Chain)**.** The evidence hash chain is defined recursively:

$$\boxed{h_0 = 0^{256}, \qquad h_i = H(\text{serialize}(ev_i) \parallel h_{i-1}) \quad \text{for } i \geq 1} \tag{70}$$

Any modification to ev_j for $j \leq i$ changes h_j and cascades through all subsequent hashes.

Lemma 6.4 (Tamper Detection)**.** *Under the collision-resistance assumption on H, if an adversary modifies ev_j to produce $ev'_j \neq ev_j$, then:*

$$h'_j \neq h_j \implies h'_k \neq h_k \;\; \forall\, k \geq j \tag{71}$$

with overwhelming probability in the security parameter.

Proof. By the recursive definition (Equation 70), h_j depends on serialize(ev_j) and h_{j-1}. If $ev'_j \neq ev_j$, then serialize$(ev'_j) \neq$ serialize(ev_j) (assuming injective serialization), so $H(\text{serialize}(ev'_j)\|h_{j-1}) \neq H(\text{serialize}(ev_j)\|h_{j-1})$ by collision resistance. The cascade follows by induction on k. □ □

6.3 Signature Blocks

Periodic checkpoints are signed for authenticity and sequencing:

$$sig_j \;=\; \text{Sign}(sk,\; h_{t_j} \;\|\; j \;\|\; ts_j) \tag{72}$$

Verification: Verify$(pk,\ h_{t_j}\|j\|ts_j,\ sig_j) = \mathit{true}$.

This is aligned with RFC 5848's signature blocks for syslog messages [13], providing: origin authentication, data integrity, replay resistance, sequencing, and missing-message detection.

6.4 Merkle Tree Anchoring

Definition 6.5 (Merkle Tree Construction)**.** For a batch of evidence hashes $\{h_1, \ldots, h_n\}$, the Merkle tree is constructed recursively. For a complete binary tree with 2^d leaves:

$$M(i,j) \;=\; \begin{cases} H(h_i) & \text{if } i = j \\ H(M(i,m) \;\|\; M(m{+}1,j)) & \text{where } m = \lfloor \frac{i+j}{2} \rfloor \end{cases} \tag{73}$$

The Merkle root is $R = M(1,n)$. The root R is anchored into an append-only transparency log [14, 15] or cryptographically verifiable ledger [20].

Definition 6.6 (Inclusion Proof)**.** Given leaf h_k and proof path $\pi_k = \langle s_1, s_2, \ldots, s_d \rangle$ (sibling hashes along the path from leaf to root), verification proceeds by recomputing:

$$v_0 = H(h_k), \quad v_i = \begin{cases} H(v_{i-1}\|s_i) & h_k \text{ is left child at level } i \\ H(s_i\|v_{i-1}) & \text{otherwise} \end{cases} \tag{74}$$

and checking:

$$\text{VerifyInclusion}(h_k,\ \pi_k,\ R) = \mathit{true} \quad \iff \quad v_d = R \tag{75}$$

Proposition 6.7 (Evidence Completeness)**.** *Under ZelC's evidence-first semantics, for every kinetic action a_i executed in a well-typed program:*

$$\forall\, a_i \in \mathcal{A}(P) :\ \exists\, ev_i \in \mathcal{E}(P)\ \textit{s.t.}\ \text{VerifyInclusion}\big(H(\text{serialize}(ev_i)), \pi_i, R\big) = \mathit{true} \tag{76}$$

The design target is evidence completeness $E_1 = 1.0$.

7 Taint Tracking and Information-Flow Constraints

7.1 Security Label Lattice

Definition 7.1 (Label Lattice)**.** The taint label domain is:

$$\ell \;\in\; \mathcal{L} = \{\, \bot,\ \top \,\} \qquad \text{with} \qquad \bot \;\sqsubseteq\; \top \tag{77}$$

where $\bot$ denotes *untainted* (trusted) and $\top$ denotes *tainted* (untrusted). Types are annotated: τ^{ℓ}. The join:

$$\ell_1 \sqcup \ell_2 \;=\; \begin{cases} \bot & \text{if } \ell_1 = \ell_2 = \bot \\ \top & \text{otherwise} \end{cases} \tag{78}$$

7.2 Taint Propagation Rules

Definition 7.2 (Propagation)**.** For any n-ary operator f:

$$\frac{\Gamma \vdash x_1 : \tau_1^{\ell_1} \quad \cdots \quad \Gamma \vdash x_n : \tau_n^{\ell_n}}{\Gamma \vdash f(x_1, \ldots, x_n) : \tau^{\bigsqcup_{i=1}^{n} \ell_i}} \textsc{ Taint} \tag{79}$$

External input sources introduce taint:

$$\frac{v = \text{external\_input}()}{\Gamma \vdash v : \tau^{\top}} \textsc{ Source} \tag{80}$$

7.3 Sensitive Sinks and Sanitization

Definition 7.3 (Sensitive Sinks)**.** The set of sensitive operational sinks in ZelC:

$$\mathcal{S} = \{\texttt{firewall.block}, \texttt{iam.revoke}, \texttt{iam.disable}, \\ \texttt{edr.isolate}, \texttt{edr.kill}, \texttt{linux.kill}, \texttt{delete}, \ldots\} \tag{81}$$

Definition 7.4 (Sink Rejection Rule)**.** Tainted arguments to sensitive sinks are rejected at compile time:

$$\frac{\Gamma \vdash v : \tau^{\top} \quad s \in \mathcal{S}}{\Gamma \vdash s(v) : \textbf{error}} \textsc{ Sink-Reject} \tag{82}$$

Definition 7.5 (Evidence-Generating Sanitization)**.** Explicit sanitization produces a clean value *and* an evidence trail:

$$\frac{\Gamma \vdash v : \tau^{\top} \quad f_{\text{san}} \in \text{Sanitizers}(\tau)}{\Gamma \vdash f_{\text{san}}(v) : \tau^{\perp} \ \wedge \ \textbf{evidence emit}(\mathit{san\_rec})} \textsc{ Sanitize} \tag{83}$$

The sanitization record includes: the original tainted value's hash, the sanitizer identity, timestamp, and the clean value, enabling post-hoc audit of all data-cleaning decisions.

Theorem 7.6 (Taint Safety)**.** *In a well-typed ZelC program, no tainted value reaches a sensitive sink without passing through a sanitizer that produces an evidence record.*

Proof. By the typing rules: (Sink-Reject) prevents direct use of tainted values at sinks, and (Sanitize) is the only rule that converts $\top \to \perp$. Since (Sanitize) mandates evidence emission, every path from tainted source to sensitive sink must pass through an evidence-generating sanitization step. □ □

8 Agentic AI Safety: Intent-Gated Execution

8.1 Threat Model for Agentic Generation

Agentic AI systems may propose incorrect or dangerous actions through hallucination, prompt injection, or misalignment. The threat model includes:

(a) **Hallucinated actions**: The agent proposes `process.kill` when the intent allows only `service.restart`.

(b) **Scope escalation**: The agent targets 500 hosts when the intent allows 10.
(c) **Protected-object violation**: The agent targets production databases when those are in *ExcludeScope*.
(d) **Temporal violation**: The agent acts outside maintenance windows.

ZelC's guardrails are not prompt-based; they are enforced by the runtime regardless of code provenance.

8.2 Conformance Predicate

Definition 8.1 (Program Conformance)**.** Let P be a candidate program (human-written or agent-generated). Define:

$$\boxed{\mathrm{Conforms}(P,\mathcal{I},\Sigma,\mathcal{C}) \equiv \bigwedge_{a(\vec{v})\in\mathcal{A}(P)} \Big[\mathrm{Auth}(\Sigma,\mathcal{C},a(\vec{v})) \wedge \mathrm{Sat}(\mathcal{I},\Sigma,a(\vec{v})) \wedge \mathrm{Taint\text{-}Safe}(a(\vec{v}))\Big]} \quad (84)$$

ZelC compilation and execution require Conforms to hold, or *fail closed*.

Theorem 8.2 (Agent-Agnostic Safety)**.** *For any program P—whether authored by a human operator or generated by an AI agent—if the ZelC runtime accepts P for execution, then for every kinetic action $a(\vec{v}) \in \mathcal{A}(P)$:*

$$\mathrm{Auth}(\Sigma,\mathcal{C},a(\vec{v})) \;\wedge\; \mathrm{Sat}(\mathcal{I},\Sigma,a(\vec{v})) \;\wedge\; \mathrm{Taint\text{-}Safe}(a(\vec{v})) \;\wedge\; \exists\, ev_i \in \mathcal{E}(P) \quad (85)$$

That is: every executed kinetic action is authorized, intent-compliant, taint-safe, and evidence-generating.

Proof. By Definition 8.1, the runtime checks $\mathrm{Conforms}(P,\mathcal{I},\Sigma,\mathcal{C})$ which universally quantifies over all kinetic actions. Each conjunct is verified: Auth by Definition 4.2, Sat by Definition 5.2, and Taint-Safe by Theorem 7.6. Evidence emission follows from the operational semantics (Definition 6.1 and rule Kin-Accept in Section 9), which generates ev_i for every executed kinetic action. □ □

8.3 Verify Before Execute: Operational Proof Obligations

ZelC imposes proof obligations analogous to proof-carrying code [18]:

$$\mathcal{R}(P,\,\mathcal{I},\,\mathcal{C},\,\Sigma) \;=\; \begin{cases} \textsc{Accept} & \text{if } \mathrm{Conforms}(P,\mathcal{I},\Sigma,\mathcal{C}) \\ \textsc{Reject} & \text{otherwise (fail closed)} \end{cases} \quad (86)$$

Note that even *rejected* programs produce evidence records documenting the rejection reason, timestamp, and the principal who attempted execution.

9 Operational Semantics

9.1 Runtime Configuration

Definition 9.1 (Configuration)**.** A runtime configuration is a five-tuple:

$$\sigma \;=\; \langle\, \Sigma,\; \mathcal{C},\; \mathcal{I},\; \mathcal{E},\; S \,\rangle \quad (87)$$

where Σ is the machine state, $\mathcal{C}$ is the capability set, $\mathcal{I}$ is the active intent, $\mathcal{E}$ is the accumulated evidence trace, and S is the remaining statement to execute.

9.2 Small-Step Transition Rules

The transition relation $\sigma \longrightarrow \sigma'$ is defined by the following rules.

Rule (Set) — Variable binding (read-only):

$$\frac{[\![E]\!]_\Sigma = v}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{set}\ x = E; S\rangle \longrightarrow \langle \Sigma[x \mapsto v], \mathcal{C}, \mathcal{I}, \mathcal{E}, S\rangle}\ \textsc{Set} \tag{88}$$

Rule (When-True):

$$\frac{[\![E]\!]_\Sigma = \mathit{true}}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{when}\ E\ \{S_1\}\ \textbf{otherwise}\ \{S_2\}; S\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E}, S_1; S\rangle}\ \textsc{When-T} \tag{89}$$

Rule (When-False):

$$\frac{[\![E]\!]_\Sigma = \mathit{false}}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{when}\ E\ \{S_1\}\ \textbf{otherwise}\ \{S_2\}; S\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E}, S_2; S\rangle}\ \textsc{When-F} \tag{90}$$

Rule (Kinetic-Accept) — The central rule; requires authorization *and* intent satisfaction:

$$\frac{\mathrm{Auth}(\Sigma, \mathcal{C}, a(\vec{v}))\quad \mathrm{Sat}(\mathcal{I}, \Sigma, a(\vec{v}))\quad \Sigma' = \delta(\Sigma, a(\vec{v}))\quad ev = \mathrm{GenEvidence}(\ldots)}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E}, \textbf{action}\ a(\vec{v}); K\rangle \longrightarrow \langle \Sigma', \mathcal{C}, \mathcal{I}, \mathcal{E} \| ev, K\rangle}\ \textsc{Kin-Accept} \tag{91}$$

Rule (Kinetic-Reject) — Fail closed with evidence:

$$\frac{\neg\,\mathrm{Auth}(\Sigma, \mathcal{C}, a(\vec{v}))\ \vee\ \neg\,\mathrm{Sat}(\mathcal{I}, \Sigma, a(\vec{v}))}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{action}\ a(\vec{v}); K\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E} \| ev_{\mathrm{rej}}, \textbf{halt}\rangle}\ \textsc{Kin-Reject} \tag{92}$$

Rule (Evidence-Emit) — Evidence emission within kinetic blocks:

$$\frac{ev = \mathrm{MkEvidence}(\mathit{title}, [\![E]\!]_\Sigma, \Sigma, \mathcal{I})}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{evidence record}\ \mathit{title}\ \textbf{det}\ E;\ S\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E} \| ev, S\rangle}\ \textsc{Ev-Emit} \tag{93}$$

Rule (Each-Step) — Iteration unfolds one element:

$$\frac{[\![E]\!]_\Sigma = [v_1, v_2, \ldots, v_n]\quad n \geq 1}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{each}\ x\ \textbf{in}\ E\ \{S_b\}; S\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ S_b[v_1/x];\ \textbf{each}\ x\ \textbf{in}\ [v_2 \ldots]; S\rangle}\ \textsc{Each} \tag{94}$$

Rule (Stop) — Hard halt:

$$\frac{}{\langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E},\ \textbf{stop}; S\rangle \longrightarrow \langle \Sigma, \mathcal{C}, \mathcal{I}, \mathcal{E} \| ev_{\mathrm{halt}}, \textbf{halt}\rangle}\ \textsc{Stop} \tag{95}$$

Invariant 9.2 (Evidence Monotonicity)**.** The evidence trace $\mathcal{E}$ is monotonically non-decreasing across all transitions:

$$\sigma \longrightarrow \sigma' \quad \Longrightarrow \quad \mathcal{E}' \supseteq \mathcal{E} \tag{96}$$

Evidence is *append-only*; no transition removes or modifies existing evidence records.

10 Concrete Operational Programs

We present complete ZelC programs demonstrating that the formal model maps directly to operational reality.

10.1 Ransomware Containment

```
intent RansomwareContainment {
  category: containment
  risk_level: critical
  constraints {
    max_targets: 20
    time_limit: 5m
    requires_approval: auto
    exclude_scope: [gateway, dns_primary]
  }
}

check RansomwareContainment
  keep critical_hosts = ["prod-db", "prod-web", "prod-api"]

  when ransomware_detected
    alert critical message "RANSOMWARE DETECTED"

    do
      set infected = event.hostname
      set hash = event.file_hash

      -- Isolate: cuts network, leaves mgmt tunnel
      edr isolate host infected

      -- Kill malicious processes
      each proc in event.suspicious_processes
        edr kill process proc.pid on infected
      end

      -- Quarantine the payload
      edr quarantine file event.file_path on infected

      -- Forensic snapshot before remediation
      aws ec2 snapshot instance infected

      -- Escalate if production
      when infected in critical_hosts
        pager trigger message "CRITICAL: {infected}"
      end

      -- Threat intel enrichment
      set intel = threat lookup hash hash

      -- Evidence with blockchain anchor
      evidence record "ransomware-incident" details {
        host: infected,
        malware_hash: hash,
        processes_killed: event.suspicious_processes,
        threat_intel: intel,
        containment_time: now()
      }

      rosecoin anchor evidence_pack "ransomware-{infected}"
    end
  end
end
```

Listing 2: Ransomware containment playbook with evidence anchoring.

10.2 Cloud Key Leak Response

```
check CloudKeyLeakResponse
  keep repo = "rocheston/zelfire"
  keep channel = "#soc"

  when access_key_leak or secret_leak
    alert critical message "Possible cloud key leak detected"
    notify slack channel channel message "Rotating keys"

    do
      -- Rotate compromised credentials
      aws rotate keys

      -- Scan source for additional leaks
      github scan repo repo for secret_leak

      -- Generate and anchor evidence
      noodles generate evidence_pack
      rosecoin anchor evidence_pack "latest"

      evidence record "Key rotation and repo scan" details {
        repo: repo,
        timestamp: now()
      }

      audit log "ZelC run completed" details {
        action: "rotate+scan",
        repo: repo
      }
    end
  end
end
```

Listing 3: Cloud key leak response with credential rotation.

10.3 Brute-Force Defense

```
intent BruteForceDefense {
  category: containment
  constraints {
    max_targets: 5
    time_limit: 1h
    exclude_scope: [internal_subnets, vpn_gateway]
  }
}

check BruteForceShield
  keep threshold = 12
  keep block_duration = 2 hours

  when bruteforce_login
    alert high message "Brute force attack detected"

    do
      set attacker_ip = event.source_ip
      set target_user = event.username

      -- Block at network edge
      firewall block ip attacker_ip for block_duration

      -- Lock target account
      iam lock user target_user

      -- Notify and track
      notify slack channel "#security-alerts"
        message "Brute force from {attacker_ip}"

      ticket open title "Brute Force - {attacker_ip}"
        severity "high"

      evidence record "brute-force-containment" details {
        attacker_ip: attacker_ip,
        target_user: target_user,
        attempts: event.attempt_count,
        actions_taken: ["ip_blocked", "user_locked"]
      }
    end
  end
end
```

Listing 4: Brute-force detection and containment.

10.4 CI/CD Pipeline Integrity Verification

```
check PipelineGuard(repository_path)
  set repo = git.open(repository_path)
  set head = repo.get_head()

  -- Verify commit signature (GPG)
  set sig_valid = crypto gpg verify head.signature

  when sig_valid is false
    alert critical message "UNVERIFIED AUTHOR"
    do
      github block merge reason "Unsigned Commit"
      evidence record "Bad_Actor_Attempt" details {
        author: head.author,
        email: head.email
      }
      stop
    end
  end

  -- Scan for leaked secrets
  set scan = aina code scan_secrets repo.files

  when scan.secrets_found > 0
    alert warning message "LEAK: {scan.count} secrets"
    do
      notify slack user head.author
        message "You committed an API Key"
      stop
    end
  end

  otherwise
    do
      -- Generate SBOM
      set sbom = sbom generate format "spdx"

      -- Build distroless container
      set image = docker build repo base "distroless"

      -- Vulnerability scan
      set vulns = docker image scan image
      when vulns.critical_count > 0
        alert critical message "VULNERABLE DEPENDENCY"
        stop
      end

      -- Sign with cosign
      crypto sign artifact image

      -- Anchor provenance to blockchain
      rosecoin anchor {
        type: "Supply_Chain_Release",
        commit: head.hash,
        image_hash: image.sha256,
        sbom_hash: crypto hash sha256 sbom
      }

      -- Deploy
      kube apply image.tag namespace "production"
      notify slack channel "#releases"
        message "Deployed {repo.tag} securely"
    end
  end
end
```

Listing 5: Supply-chain integrity daemon with SLSA attestation.

10.5 Zero-Trust Network Gatekeeper

```
check ZeroTrustGatekeeper
  keep MAX_ENTROPY = 7.85
  keep REQ_PER_SEC = 5000
  keep GEO_RISK = 85

  set stream = net capture interface "eth0" filter "tcp port 443"

  while true
    set pkt = stream.next()

    -- Entropy analysis for encrypted payloads
    set entropy = crypto entropy pkt.payload
    set verdict = aina net inspect pkt

    when entropy > MAX_ENTROPY or verdict.is_malicious
      alert critical message "MALWARE: {pkt.src_ip}"
      do
        linux firewall block ip pkt.src_ip duration "permanent"
        net kill_connection pkt.session_id

        evidence record "Payload_Capture" details {
          src_ip: pkt.src_ip,
          entropy: entropy,
          ai_confidence: verdict.confidence
        }

        rosecoin anchor {
          type: "Threat_Neutralized",
          threat_hash: crypto hash sha256 pkt.payload
        }
      end
    end

    -- DDoS rate limiting
    set rate = net get_flow_rate pkt.src_ip
    when rate > REQ_PER_SEC
      alert warning message "DDoS: {pkt.src_ip} @ {rate} rps"
      do
        aws waf update_ip_set action "INSERT"
```

```
          ip pkt.src_ip list "Global_Blocklist"
        end
      end

      -- Geofencing
      set geo = net geoip pkt.src_ip
      when geo.risk_score > GEO_RISK
        do
          net drop pkt
          audit log "Geo_Block" details {
            country: geo.country,
            risk: geo.risk_score
          }
        end
      end
    end
  end
```

Listing 6: Zero-trust gatekeeper with entropy analysis and geofencing.

11 Architectural Overview

11.1 Execution Pipeline

The ZelC runtime processes programs through a multi-stage pipeline, depicted in Figure 1.

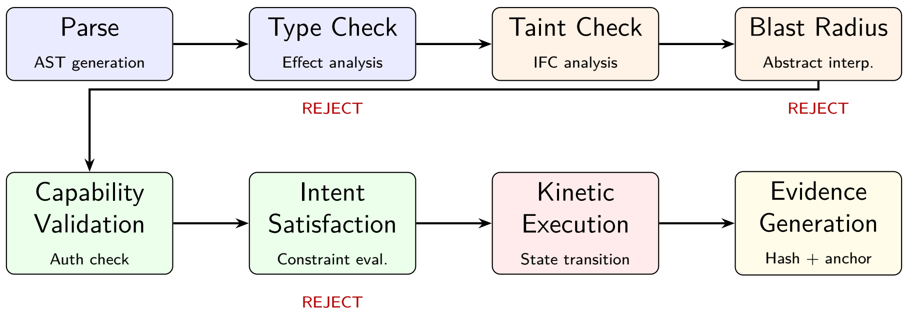


Figure 1: The ZelC execution pipeline. Static phases (top row) precede runtime phases (bottom row). Rejections at any stage produce evidence records and halt execution.

11.2 Three-Layer Safety Architecture

ZelC implements defense-in-depth through three orthogonal safety layers:

Table 1: The three layers of safety in ZelC.

Layer	**Mechanism**	**Phase**	**Prevents**
Type Safety	Effect-typed compiler	Compile	Type confusion, kinetic actions outside **do** blocks
Memory Safety	Scope-bound managed refs	Runtime	Buffer overflows, use-after-free, memory leaks
Kinetic Safety	Intent-gated guard	Runtime	Mass deletion, scope violation, unauthorized escalation

11.3 Memory Model

ZelC uses a *scope-bound* memory model without raw pointers:

Definition 11.1 (Scope-Bound Memory)**.** Memory allocations within a **check** or **do** block are automatically reclaimed when the block exits:

$$\text{exit}(\textbf{do}) \implies \forall\, \mathit{ref} \in \text{locals}(\textbf{do}) : \ \text{dealloc}(\mathit{ref}) \tag{97}$$

No pointer arithmetic, no `malloc`/`free`, no garbage-collector pauses. This eliminates buffer overflows and memory leaks for operational tasks.

11.4 Concurrency Model

ZelC employs an actor model for concurrent execution:

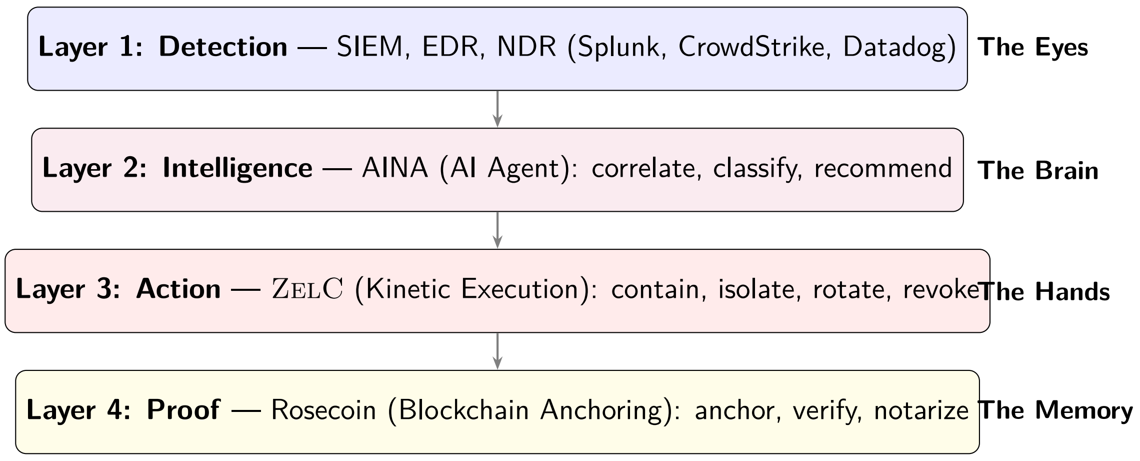
Definition 11.2 (Actor Isolation)**.** Each **check** block executes in an isolated lightweight actor with:

$$\text{Actor}(\textbf{check}\ N) \ = \ \langle\, \Sigma_{\text{local}},\ \mathcal{C}_{\text{local}},\ \mathcal{E}_{\text{local}},\ \mathit{mailbox}\,\rangle \tag{98}$$

Actors share no mutable state. Communication is via message passing through the *mailbox*. This enables 1000+ concurrent security checks without CPU thrash or race conditions.

12 Tiered Operational Model

ZelC is designed to operate within a four-layer defense stack, depicted in Figure 2.



Layer 1: Detection — SIEM, EDR, NDR (Splunk, CrowdStrike, Datadog)
The Eyes
Layer 2: Intelligence — AINA (AI Agent): correlate, classify, recommend
The Brain
Layer 3: Action — ZelC (Kinetic Execution): contain, isolate, rotate, revoke
The Hands
Layer 4: Proof — Rosecoin (Blockchain Anchoring): anchor, verify, notarize
The Memory

Figure 2: The tiered defense model. ZelC operates at Layer 3, receiving intelligence from Layer 2 and producing evidence for Layer 4.

13 Comparison with General-Purpose Languages

Table 2 provides a systematic comparison across the dimensions of the kinetic gap.

Table 2: Systematic comparison: ZelC vs. general-purpose languages along kinetic gap dimensions.

Dimension	**GPL (Python/Bash)**	**ZelC**
Execution Model	Unrestricted; any line can mutate state	Read-only by default; mutations only in **do** blocks
Safety Architecture	None; developer writes all checks	Three-layer: type safety, memory safety, kinetic guard
Permission Model	OS-level ambient authority (root = God)	Capability tokens: explicit, attenuated, time-bounded
Audit Trail	Text logs (mutable, deletable)	Evidence objects (immutable, hash-chained, signed)
AI Safety	Prompt engineering (hope and pray)	Intent validation (compiler + runtime contracts)
Blast Radius	No language-level concept	Compile-time abstract interpretation + runtime re-check
Cloud Integration	External libraries (boto3, azure-sdk)	Native primitives (`cloud.aws`, `cloud.azure`)
Compliance	Manual reports after the fact	Real-time proof generated at runtime
Data Types	Strings, integers, lists	Security-semantic: **IP**, **CIDR**, **CVE**, **RiskScore**, **IOCSet**
Taint Tracking	None (developer responsibility)	Language-level IFC with evidence-generating sanitization

14 Evaluation Methodology

We propose the following metrics organized by evaluation dimension.

Table 3: Proposed evaluation metrics for ZelC.

ID	Dimension	Description	Formula
S_1	Safety	Fraction of unsafe requests rejected	$\|\{a : \neg\,\mathrm{Sat}\}\|/\|\{a : \mathrm{unsafe}\}\|$
S_2	Safety	False-reject rate	$\|\{a : \mathrm{safe} \wedge \neg\,\mathrm{Sat}\}\|/\|\{a : \mathrm{safe}\}\|$
S_3	Safety	Time-to-containment (ms)	$t_{\mathrm{contain}} - t_{\mathrm{detect}}$
E_1	Evidence	Evidence completeness	See Eq. 99
E_2	Evidence	Tamper-detection rate	$\|\{m : \mathrm{detected}\}\|/\|\{m : \mathrm{tampered}\}\|$
E_3	Evidence	Verification time (ms)	$t_{\mathrm{verify}}(\text{inclusion proof})$
O_1	Operator	Time-to-understand playbook	$t_{\mathrm{comprehend}}$ (sec)
O_2	Operator	Review error rate	$\|\text{missed dangers}\|/\|\text{total dangers}\|$
A_1	Agentic	Hallucinated actions blocked	$\|\{a : \mathrm{hallucinated} \wedge \mathrm{blocked}\}\|/\|\{a : \mathrm{hallucinated}\}\|$
A_2	Agentic	Permitted task success rate	$\|\{t : \mathrm{success}\}\|/\|\{t : \mathrm{attempted}\}\|$

The evidence completeness metric is formally:

$$E_1 \;=\; \frac{|\,\{\, a_i \in \mathcal{A}(P) \;\mid\; \exists\, ev_i \in \mathcal{E}(P)\,\}\,|}{|\mathcal{A}(P)|} \tag{99}$$

Under ZelC's evidence-first semantics (Proposition 6.7), the design target is $E_1 = 1.0$.

The containment-time metric S_3 captures the end-to-end operational velocity. For the ransomware scenario (Listing 2), the target is:

$$S_3 \;\le\; 400 \text{ ms} \qquad (\text{detect} \to \text{isolate} \to \text{evidence anchored}) \tag{100}$$

15 Contribution Claims

To maintain academic rigor, we state ZelC's contributions as *integration-level* mechanisms. Each claim identifies the novel binding and cites the prior art it builds upon.

Claim 15.1 (C1: Kinetic Semantics as Primary Operational Effect)**.** A language/runtime model enforcing structural separation between read-only reasoning (RO) and state-changing cybersecurity actuation (KI), with actuation permitted only inside explicit kinetic blocks (**do** . . . **end**) and prohibited elsewhere, via effect checks (Theorem 3.8) and runtime gating (Rule Kin-Accept). *Prior art*: effect systems [3, 4].

Claim 15.2 (C2: Capability-Scoped Actuation with Formal Non-Escalation)**.** A language invariant binding each kinetic action to explicit capability tokens with proven monotonic non-escalation (Proposition 4.5). *Prior art*: capability security [1, 2].

Claim 15.3 (C3: Intent Contracts as Enforced Blast-Radius Constraints)**.** A compiler/runtime mechanism interpreting intent declarations as constraint-satisfaction problems (Definition 5.2) with compile-time blast-radius bounding via abstract interpretation (Theorem 5.6). *Prior art*: admission control [5–7], abstract interpretation [9], eBPF verifier [8].

Claim 15.4 (C4: Evidence-First Semantics with Cryptographic Anchoring)**.** A runtime emitting structured evidence records as a semantic consequence of kinetic execution (Rule Kin-Accept), with hash chaining (Lemma 6.4), signature blocks, and Merkle-tree anchoring (Definition 6.5). *Prior art*: tamper-evident logging [11–13], transparency logs [14, 15].

Claim 15.5 (C5: Agent-Agnostic Enforcement with Taint Safety)**.** A safety model in which identical constraints are enforced regardless of code authorship (Theorem 8.2), with taint tracking that prevents untrusted data from reaching sensitive sinks without evidence-generating sanitization (Theorem 7.6). *Prior art*: PCC [18], LMQL [19], IFC [10, 16, 17].

16 Conclusion

ZelC proposes a cybersecurity-native programming model that treats actuation, authority, governance, and evidence as *first-class semantics*. While each underlying primitive has significant prior art, ZelC's contribution is an integrated, operator-centered language/runtime that binds these mechanisms into a coherent operational semantics for the security operations center and incident response—especially in an agentic era where "prompt discipline" is not a safety strategy.

The formal model presented herein comprises: (i) an effect-typed kinetic boundary with soundness proof, (ii) a capability algebra with monotonic non-escalation, (iii) intent contracts formalized as constraint-satisfaction problems with blast-radius bounding via abstract interpretation, (iv) cryptographic evidence chains with Merkle-tree anchoring and inclusion-proof verification, (v) taint tracking with evidence-generating sanitization for operational sinks, and (vi) a conformance predicate providing agent-agnostic safety guarantees.

The concrete program listings demonstrate that these formal mechanisms are not abstract constructs but directly executable security operations: ransomware containment, credential rotation, brute-force defense, supply-chain integrity, and zero-trust network gating.

By making kinetic operations explicit, capability-scoped, intent-constrained, and evidentiary by construction, ZelC aims to reduce catastrophic operator error, improve auditability, and enable safe automation at scale—closing the kinetic gap that has separated security intent from operational execution for decades.

A ZelC Type Universe: Complete Reference

Table 4 enumerates the complete type universe.

Table 4: Complete ZelC type universe.

Category	Type	Description
Primitive	**unit**, **Bool**, **Int**, **Float**, **Text**	Standard value types
Temporal	**Duration**, **Timestamp**	Time-aware operations
Network	**IP**, **CIDR**, **Port**, **MAC**, **URL**	Network address types
Identity	**UserId**, **GroupId**, **ServiceAccount**	Identity primitives
Crypto	**Hash**, **Signature**, **PublicKey**, **Certificate**	Cryptographic types
Threat	**CVE**, **IOCSet**, **YaraRule**, **SigmaRule**	Threat intelligence
Risk	**RiskScore**, **Severity**, **Priority**	Risk quantification
Incident	**Incident**, **Ticket**, **Timeline**	Case management
Evidence	**EvidencePack**, **AuditRecord**, **ChainOfCustody**	Forensic types
Compliance	**ControlId**, **Baseline**, **GapReport**	GRC mapping
Cloud	**Instance**, **Bucket**, **SecurityGroup**, **VPC**	Cloud resource types
Container	**Pod**, **Container**, **Image**, **SBOM**	Container ecosystem
Compound	**List**$[\tau]$, **Map**$[\tau_k, \tau_v]$, **Record**$\{\ldots\}$	Generic containers
Function	$\tau_1 \rightarrow^{\varepsilon} \tau_2$	Effect-annotated functions

B ZelC Action Taxonomy

Table 5 classifies the kinetic action vocabulary by domain.

Table 5: Kinetic action taxonomy (representative subset).

Domain	Action	Signature
Firewall	`firewall.block`	**IP** $\times$ **Duration** $\rightarrow^{\mathsf{KI}}$ **unit**
EDR	`edr.isolate`	**Text** $\rightarrow^{\mathsf{KI}}$ **unit**
EDR	`edr.kill`	**Int** $\times$ **Text** $\rightarrow^{\mathsf{KI}}$ **unit**
EDR	`edr.quarantine`	**Text** $\times$ **Text** $\rightarrow^{\mathsf{KI}}$ **unit**
IAM	`iam.revoke`	**UserId** $\rightarrow^{\mathsf{KI}}$ **unit**
IAM	`iam.lock`	**UserId** $\rightarrow^{\mathsf{KI}}$ **unit**
AWS	`aws.rotate_keys`	**UserId** $\rightarrow^{\mathsf{KI}}$ **unit**
AWS	`aws.ec2.snapshot`	**Instance** $\rightarrow^{\mathsf{KI}}$ **Hash**
AWS	`aws.s3.block_public`	**Bucket** $\rightarrow^{\mathsf{KI}}$ **unit**
Azure	`azure.ad.block_user`	**UserId** $\rightarrow^{\mathsf{KI}}$ **unit**
Azure	`azure.nsg.deny`	**Text** $\times$ **IP** $\rightarrow^{\mathsf{KI}}$ **unit**
GCP	`gcloud.iam.disable`	**ServiceAccount** $\rightarrow^{\mathsf{KI}}$ **unit**
K8s	`kube.isolate_pod`	**Pod** $\rightarrow^{\mathsf{KI}}$ **unit**
Docker	`docker.stop`	**Container** $\rightarrow^{\mathsf{KI}}$ **unit**
Linux	`linux.kill`	**Int** $\rightarrow^{\mathsf{KI}}$ **unit**
Linux	`linux.service.restart`	**Text** $\rightarrow^{\mathsf{KI}}$ **unit**
Crypto	`crypto.sign`	**Text** $\rightarrow^{\mathsf{KI}}$ **Signature**
Rosecoin	`rosecoin.anchor`	**EvidencePack** $\rightarrow^{\mathsf{KI}}$ **Hash**

Note that *every* action in this table carries effect KI and therefore can only appear inside a **do** block, subject to capability authorization and intent satisfaction.

C Formal Proof: Evidence Chain Integrity

Theorem C.1 (Evidence Chain Integrity). *Let $\mathcal{E} = \langle ev_1, \ldots, ev_n \rangle$ be the evidence trace. Let $\langle h_0, \ldots, h_n \rangle$ be the hash chain (Definition 6.3), $R = \mathrm{MerkleRoot}(h_1, \ldots, h_n)$, and $\langle sig_1, \ldots, sig_m \rangle$ the signature checkpoints. The following properties hold under standard cryptographic assumptions:*

(i) ***Completeness:*** *Every kinetic action has a corresponding evidence record ($E_1 = 1.0$).*
(ii) ***Tamper evidence:*** *Modification of any ev_j is detectable (Lemma 6.4).*
(iii) ***Non-repudiation:*** *For each signed checkpoint sig_j, the signer cannot deny having seen h_{t_j}.*
(iv) ***Append-only:*** *Deletion of any ev_j is detectable via the hash chain break.*
(v) ***Verifiable inclusion:*** *For any ev_k, an auditor can verify inclusion via* VerifyInclusion *in $O(\log n)$ time.*

Proof. *(i)* follows from Proposition 6.7 and the operational semantics (rules Kin-Accept and Kin-Reject both emit evidence). *(ii)* follows from Lemma 6.4. *(iii)* follows from the unforgeability of digital signatures under the chosen scheme. *(iv)* follows because deletion of ev_j removes h_j from the chain, causing h_{j+1} verification to fail. *(v)* follows from the standard Merkle-tree inclusion proof: the proof path π_k has length $\lceil \log_2 n \rceil$, and verification requires $\lceil \log_2 n \rceil$ hash computations. □ □

References

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

[2] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. thesis, Johns Hopkins University, 2006.

[3] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 47–57, 1988.

[4] D. Leijen. Koka: Programming with row polymorphic effect types. *Electronic Proceedings in Theoretical Computer Science*, 153:100–126, 2014.

[5] Kubernetes Contributors. Admission controllers reference. `https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/`, 2024.

[6] Open Policy Agent Contributors. OPA Gatekeeper: Policy and governance for Kubernetes. `https://open-policy-agent.github.io/gatekeeper/`, 2024.

[7] HashiCorp. Sentinel: Policy as code framework. `https://www.hashicorp.com/sentinel`, 2024.

[8] Linux Kernel Contributors. eBPF verifier. `https://docs.kernel.org/bpf/verifier.html`, 2024.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[10] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[11] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 2(2):159–176, 1999.

[12] R. T. Snodgrass, S. S. Yao, and C. Collberg. Tamper detection in audit logs. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 504–515, 2004.

[13] J. Kelsey, J. Callas, and A. Clemm. Signed syslog messages. RFC 5848, Internet Engineering Task Force, 2007.

[14] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, Internet Engineering Task Force, 2013.

[15] Sigstore Contributors. Rekor: Transparency log for software supply chains. https://docs.sigstore.dev/logging/overview/, 2024.

[16] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 32(2):5:1–5:29, 2014.

[17] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[18] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[19] L. Beurer-Kellner, M. Fischer, and M. Vechev. Prompting is programming: A query language for large language models. In *Proceedings of the ACM SIGPLAN 2023 Conference on Programming Language Design and Implementation (PLDI)*, pages 1507–1520, 2023.

[20] Amazon Web Services. Amazon Quantum Ledger Database (QLDB). https://aws.amazon.com/qldb/, 2024.