

AppSec Core: A Normalized Ontology for Security Requirements Across Heterogeneous Frameworks

Pedro Farinha

Independent Researcher

pedro.farinha@shiftleft.pt

github.com/pedrofarinhaatshiftleftpt

Supported by Shiftleft - Secure Software Engineering, lda.

Abstract

The application security (AppSec) landscape is served by an expanding set of frameworks, standards, and regulations, each using its own vocabulary, granularity, and structural conventions. Organizations that must demonstrate alignment with multiple sources simultaneously face a quadratic mapping problem with no shared semantic model to mediate it.

We propose AppSec Core, a normalized ontology for application security comprising 10 domain slices and 234 typed instances. The ontology defines four entity types — ControlObjective, Practice, Mechanism, and Artifact — connected by a stable set of relations (implements, realizes, evidences) that are structurally invariant across all slices. Each slice is governed by a formal contract specifying scope, boundaries, and non-goals. The ontology was derived from practitioner semantics accumulated over a decade of AppSec engineering, not from any single external framework.

We present evidence that AppSec Core can function as a semantic normalization layer through canonical reduction: requirements from a first-wave set of five sources (SSDF, ASVS, SLSA, CIS Controls, and CAPEC) — expressed in incompatible vocabularies — are reduced to shared ControlObjectives and their associated operational chains within the ontology, after which downstream coverage analysis operates on the normalized objective layer independently of framework vocabulary. Mapping exercises across three presented slices show that 79% of analyzed ControlObjectives are mapped by at least two frameworks, and the 10-slice model accommodates every requirement analyzed under the stated inclusion protocol without structural extension.

We contribute: (1) the AppSec Core ontology as a reusable domain model; (2) the four-type instance schema with explicit traceability relations; (3) empirical evidence of semantic normalization through canonical reduction — heterogeneous framework requirements are reduced to shared ControlObjectives, after which downstream coverage analysis is independent of framework vocabulary; and (4) a contract-based governance model that supports principled extension.

Keywords: ontology, application security, normalization, security frameworks, SSDF, ASVS, SLSA, knowledge representation

1. Introduction

Organizations developing software in regulated or security-sensitive environments must align their practices with multiple external frameworks. NIST’s Secure Software Development Framework (SSDF) [1] prescribes 20 practices across the secure development lifecycle. OWASP’s Application Security Verification Standard (ASVS) v5.0 [2] defines 443 verification requirements organized in 36 thematic clus-

ters. Supply-chain Levels for Software Artifacts (SLSA) [3] specifies 14 build integrity requirements across three maturity levels. CIS Controls v8.1 [4] addresses 18 enterprise security controls. Regulatory instruments such as the EU’s DORA [5] and NIS2 [6] add further obligations; their specific mapping to AppSec Core is outside the evidentiary scope of this paper (see Scope, below).

Each of these sources uses its own terminology, structural conventions, and level of granularity. SSDF speaks of *practices* organized in four families (Prepare, Protect, Produce, Respond). ASVS speaks of *verification requirements* grouped by thematic cluster. SLSA speaks of *build levels* with specific provenance and isolation properties. These are not interchangeable vocabularies: they partition the same underlying domain differently.

For any single framework, alignment is manageable. The problem emerges at the intersection: when an organization must assess coverage against multiple frameworks simultaneously, the mapping effort scales as N internal practices \times M external requirements, terminology conflicts accumulate, and there is no principled way to determine whether two requirements from different frameworks address the same underlying security concern.

The Missing Layer

What is missing is a normalization layer — a shared semantic model into which requirements from heterogeneous frameworks can be mapped, enabling:

- Semantic deduplication: recognizing that SSDF PW.9 (“secure settings by default”), ASVS `secure_configuration_baseline`, and CIS-4 (“secure configuration”) address the same concern
- Unified coverage analysis: determining whether an organization’s practices cover a requirement *once*, regardless of how many frameworks express it
- Framework-version resilience: when ASVS v6.0 is published, the mapping to the normalization layer is updated — downstream analysis does not change
- Cross-framework gap detection: identifying requirements covered by one framework but not another, through the shared model

This paper proposes AppSec Core as this normalization layer. AppSec Core is an ontology of 10 domain slices with 234 typed instances, derived from practitioner semantics rather than from any single external framework.

Contributions

1. The AppSec Core ontology: 10 slices covering the application security domain, each with typed instances (ControlObjective, Practice, Mechanism, Artifact) connected by explicit relations.
2. A four-type instance model with structural invariance across all slices, providing a reusable schema for security knowledge representation.
3. Empirical evidence of semantic normalization through canonical reduction: we show through concrete mapping examples that requirements from SSDF, ASVS, and SLSA are reduced to shared AppSec Core ControlObjectives, producing a canonical representation in which framework-specific vocabulary is no longer required for downstream coverage analysis.
4. A contract-based governance model for ontology extension, where each slice is governed by a formal contract specifying scope, non-goals, and boundary conditions.

Scope

This paper is about representation. It does not address retrieval pipelines, LLM grounding, or experimental evaluation — those are treated in companion work. The empirical mapping evidence reported here is restricted to the first-wave framework set described above; later maturity and regulatory waves are outside the evidentiary scope of this paper. The SbD-ToE manual, from which the ontology was empirically derived, is referenced as context for the derivation process but is not the subject of the paper.

2. Related Work

2.1 Application Security Frameworks

SSDF [1], ASVS [2], SLSA [3], and CIS Controls [4] each provide comprehensive guidance within their respective scopes. They are designed as consumption endpoints — organizations implement against them — not as normalization layers. Each defines its own vocabulary and structure. Cross-framework alignment is left to the consumer.

2.2 Security Ontologies

Several ontologies have been proposed for information security. Herzog et al. [7] proposed a general security ontology covering assets, threats, and countermeasures. Blanco et al. [8] carried out a systematic review and formal comparison of security ontologies, deriving the basis for a more integrated security ontology. These works operate at the broad information security level; AppSec Core is scoped specifically to application security practices — what development teams do to build secure software — a narrower but more operationally precise domain.

CWE [9] and CAPEC [10] model the problem space (what can go wrong). AppSec Core models the solution space (what practitioners do to prevent it). This distinction is relevant: coverage analysis requires a solution-space model. CAPEC is included in the normalization mapping (Section 5) as a problem-space cross-reference: its attack patterns validate AppSec Core solution-space objectives rather than prescribe practices, and it is treated as a complementary adversarial source rather than a normative framework equivalent to SSDF or ASVS.

2.3 Maturity Models

OWASP SAMM [11] and BSIMM [12] assess how well an organization performs security activities at varying maturity levels. They are complementary to AppSec Core: SAMM/BSIMM assess maturity; AppSec Core models what the activities are. An organization could use AppSec Core for coverage analysis and SAMM for maturity assessment over the same underlying practices.

2.4 Requirements Normalization

The NIST SP 800-53 overlay methodology [13] provides a mechanism for customizing security control baselines. Fabian et al. [14] compare security requirements engineering methods including SQUARE and SREP. Goknil et al. [15] formalize trace relation semantics for requirements models. AppSec Core differs from these in providing a semantic normalization layer with typed instances and explicit relations, rather than a methodology or a single framework’s customization mechanism.

2.5 Gap

To our knowledge, no existing model provides a typed, structurally invariant, framework-independent semantic layer at the application security practice level that supports cross-framework normalization.

AppSec Core is designed to address this gap.

3. The AppSec Core Ontology

3.1 Design Principles

AppSec Core was designed to satisfy four principles:

P1 — Practitioner fidelity. The ontology reflects how practitioners organize and reason about application security. It was derived bottom-up from 10+ years of hands-on AppSec engineering, not from the categories of any specific framework.

P2 — Framework independence. The ontology is designed to be stable as external frameworks evolve. Adding a new framework requires mapping its requirements to existing slices, not restructuring the ontology.

P3 — Structural invariance. Every slice uses the same entity types, the same relation model, and the same contract structure. This enables uniform tooling and analysis without per-slice customization.

P4 — Contract-based governance. Each slice is governed by a formal contract specifying scope, non-goals, and boundary conditions, enabling principled extension without scope creep.

3.2 Derivation Process

The ontology was derived through an iterative, manual-first, qualitative normalization process over the SbD-ToE manual corpus. The derivation was not a top-down design exercise, nor was it driven by unsupervised clustering or topic modelling. No statistical clustering algorithm was used to define ontology boundaries or slice membership. Ontology boundaries, slice contracts, and objective atomicity were established by expert qualitative analysis of requirements and broad control surfaces across chapters.

The derivation proceeded as follows:

1. Corpus analysis: 4,139 structural units from the manual were indexed with document role and normative weight annotations, providing a structured surface for qualitative reading
2. Concern clustering: recurring security concerns were identified by expert reading of requirements and broad control surfaces across chapters, then grouped into candidate subdomains by semantic judgment — not by co-occurrence analysis or embedding-based clustering
3. Objective extraction: each candidate subdomain was opened serially as a slice; within each slice, ControlObjectives were formulated atomic-first — each objective as a single, normatively coherent statement before practices or mechanisms were added
4. Practice/mechanism/artifact identification: operational, technical, and evidentiary instances were extracted from the corpus and linked to objectives only after the objective layer was coherent
5. Contract formalization: each slice advanced only when it reached an explicit contract state specifying scope, non-goals, and boundary conditions

This method was chosen because the derivation goal was normative, not descriptive: AppSec Core is not a statistical reflection of the corpus but a normalization layer with principled boundaries, atomic objectives, and explicit contracts. Unsupervised methods capture lexical proximity and semantic similarity, but do not determine what constitutes an atomic objective, where a legitimate slice boundary lies, or what should remain outside the core. These are judgment-dependent decisions that require interpretive expertise, not statistical inference. Future work may use embedding-based clustering or topic modelling as a post-hoc triangulation layer for coherence and coverage verification — but not as the

primary derivation mechanism.

The manual served as the empirical corpus; the ontology is the abstraction. The ontology is not a description of the manual — it is a domain model that the manual’s content instantiates. Evidence for this independence is provided in Section 7.4, where we show that several ontology concepts were documented by practitioners before the corresponding framework requirements were published.

3.3 The Ten Domain Slices

AppSec Core organizes the application security domain into 10 slices. Table 1 lists each slice with its scope and instance counts.

Table 1. AppSec Core v0 slices.

ID	Slice Name	Scope	CO	Pr	Me	Ar
ASC-01	Supply Chain & Build Integrity	Dependencies, provenance, build security, artifact signing	7	7	5	10
ASC-02	Identity, Access & Session Trust	Authentication, authorization, session management, credentials	7	6	6	6
ASC-03	Architecture & Trust Boundaries	Secure design, trust zones, boundary controls, design review	7	7	5	4
ASC-04	Testing, Security Validation & Empirical Assurance	Testing strategy, risk-proportional criteria, evidence	7	7	5	6
ASC-05	Threat Modeling, Risk Disposition & Mitigation Traceability	Threat analysis, risk treatment, mitigation tracking	7	7	6	4
ASC-06	Secret Handling, Protected Configuration & Operational Identities	Secrets, secure config baselines, machine identity, drift	7	6	4	2
ASC-07	Input Validation, Safe Parsing & Controlled Failure	Input handling, injection prevention, fail-safe, error management	7	6	4	6
ASC-08	Integration Trust & Service-to-Service Security	API security, service mesh, inter-service authentication, contracts	7	6	5	5
ASC-09	Release Promotion, Controlled Rollout & Rollback Readiness	Release gates, progressive deployment, canary, rollback	7	6	4	8
ASC-10	Security Event Logging, Audit Trail & Centralized Logging	Security logging, audit trails, log integrity, observability	7	5	4	2
	Total		70	63	48	53

CO = ControlObjective, Pr = Practice, Me = Mechanism, Ar = Artifact. Total: 234 instances.

Each slice contains exactly 7 control objectives (6 atomic + 1 composite). This regularity is a methodological choice for uniformity in v0, explicitly declared as non-ontological: future versions may use different cardinalities if the domain requires it.

3.4 Entity Types and Relations

The ontology defines four core entity types and one supporting type (EvidencePattern, not core-driving in v0), connected by a stable set of relations.

Entity Types **ControlObjective**. A reusable domain constraint or assurance goal. Required fields: `objective_id`, `name`, `objective_kind` (atomic | composite), `objective_type` (governance | preventive | detective | corrective), `domain_key`, `statement`, `expected_outcome`, `verification_posture`. Control objectives are the normative anchors of the ontology — the “what must be achieved.”

Practice. A reusable human or process discipline that realizes objectives, independent of specific tools. Required fields: `practice_id`, `name`, `practice_family`, `local_practice_type`. Practice families are cross-slice: `governance_and_review`, `policy_and_gate_enforcement`, `identity_and_trust_control`, `validation_and_analysis`, `integrity_traceability_and_records`, `rollout_recovery_and_runtime_records`.

Mechanism. A concrete technical or enforceable means that implements practices or directly realizes objectives. Required fields: `mechanism_id`, `name`, `mechanism_family`, `local_mechanism_type`.

Artifact. A control-relevant or evidence-bearing output used for review, traceability, and assurance. Required fields: `artifact_id`, `name`, `canonical_role`. Canonical roles are cross-slice: `configuration`, `governance_record`, `review_record`, `approval_record`, `gate_record`, `inventory`, `attestation`, `report`, `evidence_package`, `operational_record`, `build_definition`, `release_artifact`, `model`, `validation_output`, `binding_record`.

EvidencePattern. An expected evidence shape for deterministic review support. Required fields: `evidence_pattern_id`, `canonical_evidence_kind`, `expectation`, `validation_method`. EvidencePattern is defined and participates in the artifact traceability chain (`artifact_supports_evidence_pattern`) but is not instantiated as a first-class retrieval type in AppSec Core v0. It is instantiated in the runtime layer described in companion work.

Relations Table 2. Relation types in AppSec Core v0, stable across all slices and referenced consistently across all three companion papers. Prose shorthand (*realizes*, *implements*, *evidences*) appears in narrative descriptions as aliases for these identifiers; the canonical form is the relation name in the table below.

Relation	From	To	Meaning
<code>objective_realized_by_practice</code>	ControlObjective	Practice	The practice operationalizes the objective
<code>objective_implemented_by_mechanism</code>	ControlObjective	Mechanism	The mechanism technically implements the objective
<code>objective_expects_artifact</code>	ControlObjective	Artifact	The artifact evidences achievement of the objective
<code>objective_verified_by_evidence_pattern</code>	ControlObjective	EvidencePattern	The pattern defines how to verify the objective
<code>artifact_supports_evidence_pattern</code>	Artifact	EvidencePattern	The artifact feeds the evidence pattern
<code>manual_requirement_maps_to_objective</code>	ExternalRequirement	ControlObjective	Cross-framework mapping relation

Traceability Chain The relations form a traceability chain from normative to evidentiary (using active voice; the relation table above uses the equivalent passive form). A ControlObjective is realized

through Practice, implemented through Mechanism, and evidenced through Artifact; the associated EvidencePattern specifies how that evidence is to be verified.

This chain ensures that each security concern is represented from normative objective, through operational realization and technical implementation, to evidentiary verification.

Instance Identifiers Instance IDs follow a consistent prefix convention: ACO- (ControlObjective), ACP- (Practice), ACM- (Mechanism), ACA- (Artifact), followed by the slice abbreviation and a sequence number. For example, ACO-IVF-003 is the third control objective in the Input Validation slice; ACP-IVF-003 is the corresponding practice.

Instance Example The following excerpt from slice ACO-IVF (Input Validation) illustrates the four types and their relations:

ControlObjective:

```
id: ACO-IVF-003
name: Schema Validation Before Deserialization
objective_kind: atomic
objective_type: preventive
statement: "All structured input must be validated against an explicit schema
           before deserialization or internal processing"
verification_posture: "Verify typed model or JSON Schema at API entry point"
```

Practice:

```
id: ACP-IVF-003
name: Typed Model Validation At Entry Point
practice_family: validation_and_analysis
supports_objectives: [ACO-IVF-003]
```

Mechanism:

```
id: ACM-IVF-003
name: Strict Typed Model Enforcement
mechanism_family: validation_and_analysis
supports_practices: [ACP-IVF-003]
note: "e.g., pydantic BaseModel with strict=True"
```

Artifact:

```
id: ACA-IVF-003
name: Schema Validation Test Suite
canonical_role: validation_output
supports_objectives: [ACO-IVF-003]
note: "Test suite with malformed payload cases and schema coverage"
```

3.5 Slice Contracts

Each slice is governed by a formal contract. A contract specifies:

- Scope: what security concerns the slice covers
- Non-goals: what is explicitly excluded (e.g., ASC-06 covers secret handling but not full cryptogra-

phy; ASC-07 covers input handling but not runtime observability)

- Objective set: the 7 control objectives with their `objective_kind` and composite rules
- Module position: which entity types are first-class (CO, Practice, Mechanism, Artifact, EvidencePattern), which are deferred (Threat, Signal)
- External alignment status: explicitly deferred in v0

Contracts serve two functions: they prevent scope creep within existing slices, and they enable principled extension — adding a new slice requires defining a contract, not modifying existing ones.

3.6 Cross-Slice Vocabulary

A shared vocabulary applies uniformly across all slices:

- 6 practice families (`governance_and_review`, `policy_and_gate_enforcement`, `identity_and_trust_control`, `validation_and_analysis`, `integrity_traceability_and_records`, `rollout_recovery_and_runtime_records`)
- 15 canonical artifact roles (`configuration`, `governance_record`, `review_record`, `approval_record`, `gate_record`, `inventory`, `attestation`, `report`, `evidence_package`, `operational_record`, `build_definition`, `release_artifact`, `model`, `validation_output`, `binding_record`)
- 6 stable relations connecting entity types

This shared vocabulary enables cross-slice queries and uniform analysis. A tool that queries practices in one slice uses the same vocabulary and structure in all slices.

4. Normalization Approach

4.1 The Normalization Problem

Given two external requirements $r_1 \in \text{Framework A}$ and $r_2 \in \text{Framework B}$, the normalization question is: do r_1 and r_2 address the same security concern? Without a shared semantic model, this question can only be answered by reading both requirements and making a human judgment — a process that does not scale.

AppSec Core provides a principled answer: r_1 and r_2 address the same concern if they map to the same AppSec Core control objective(s). The ontology is the *tertium comparationis* — the shared reference point that makes cross-framework comparison possible.

4.2 Mapping Protocol

For each external framework, the mapping proceeds as:

1. Slice assignment: each external requirement is assigned to one or more AppSec Core slices based on semantic scope
2. Objective mapping: within each slice, the requirement is mapped to specific control objectives with an explicit strength indicator:
 - primary: the objective is the main semantic anchor for this requirement
 - secondary: the objective is materially relevant but not the best primary anchor
3. Rationale recording: each mapping includes a human-authored rationale explaining the semantic link

Mappings are many-to-one and one-to-many: a single framework requirement may map to multiple objectives, and a single objective may be the target of multiple framework requirements. This is expected — normalization is not a bijection.

In this study, mappings were performed by the ontology’s authors (a limitation we acknowledge in Section 8). The unit of analysis varied by framework: SSDF practices (N=20), ASVS thematic clusters (N=36), SLSA requirements (N=14), CIS controls (N=8 of 18), CAPEC patterns (N=13 of View 683). For CIS and CAPEC, whose scope extends beyond application security, we applied an explicit inclusion rule: an item was included if it directly affects how software is specified, developed, built, tested, deployed, or monitored. Items addressing enterprise IT administration, physical security, network infrastructure, or non-software-operational concerns were excluded. This rule yielded 8 CIS controls (e.g., CIS-4 Secure Configuration, CIS-16 Application Software Security) and 13 CAPEC patterns (the supply chain attack view). SSDF, ASVS, and SLSA were included in full, as their scope is inherently within the AppSec boundary.

A requirement was considered to match an objective if it addressed the same security concern at a comparable level of abstraction (scope overlap) and the mapping was operationally meaningful — not merely topically adjacent.

4.3 Semantic Convergence

When requirements from different frameworks map to the same objective, we observe semantic convergence: distinct vocabularies describing the same underlying concern. This convergence is the empirical signature of normalization.

4.4 The Normalization Property

We define normalization precisely:

Two external requirements $r_1 \in F_A$ and $r_2 \in F_B$ are normalized-equivalent with respect to a concern if they share at least one primary `ControlObjective` mapping within the same AppSec Core slice. They may additionally overlap on secondary objectives; the primary mapping anchors the normalized concern.

Once mapped, downstream reasoning operates on the normalized objective layer, not on framework-specific vocabulary:

- coverage analysis — coverage is assessed against the objective set; a requirement is covered when its primary objective(s) are satisfied
- implementation reasoning — the practice and mechanism chains associated with the objective apply regardless of which framework motivated the requirement
- verification reasoning — the evidence patterns and artifacts are determined by the objective, not by the originating framework

The canonical traceability structure is as follows: an `ExternalRequirement` maps to a `ControlObjective`; that objective is operationalized through `Practice` and `Mechanism`, and it is evidenced through `Artifact` and `EvidencePattern`.

This is the property that distinguishes normalization from classification. A taxonomy assigns items to categories but preserves framework-specific identity. Normalization performs canonical reduction: requirements addressing the same security concern are reduced to a shared operational representation anchored on common objectives, and subsequent coverage analysis operates on that representation. The framework origin becomes provenance metadata, not an input to reasoning.

The reduction is not strict set equality — different frameworks may cover different subsets of a slice’s objectives (as shown in Section 5.1). What is shared is the normalized concern: the primary objective(s) and their associated operational chains. This is sufficient for unified coverage analysis while honestly

reflecting the granularity differences between frameworks.

5. Mapping Examples

5.1 Configuration Security: ASVS, SSDF, and CIS

Three frameworks address configuration and secret security:

- ASVS v5.0: cluster `secure_configuration_baseline` — verification of secure defaults, secret externalization
- SSDF PW.9: “*Configure Software to Have Secure Settings by Default*”
- CIS-4: “*Secure Configuration of Enterprise Assets and Software*”

All three converge on slice ACO-SPC (Secret Handling, Protected Configuration & Operational Identities):

Objective	Description	ASVS	SSDF	CIS
ACO-SPC-001	Secret leak prevention (anti-hardcoding, scanning)	✓	✓	✓
ACO-SPC-002	Vault-backed protected storage and controlled retrieval	✓	✓	—
ACO-SPC-003	Rotation and renewal discipline	✓	—	—
ACO-SPC-004	Operational identity binding (OIDC, short-lived credentials)	—	—	✓

The convergence is not uniform: ASVS provides the most granular decomposition; SSDF operates at practice level; CIS includes enterprise concerns beyond AppSec scope. Yet all three reduce to the same normalized concern within ACO-SPC, anchored by overlapping ControlObjectives.

The operational representation is shared:

ACP-SPC-001: # *Practice*

name: Secret Leak Prevention In Source And Pipeline

supports_objectives: [ACO-SPC-001]

ACP-SPC-002: # *Practice*

name: Vault-Backed Secret Storage

supports_objectives: [ACO-SPC-002]

Requirements from the manual’s `requirements_catalog` (CFG-003: no hardcoded parameters; CFG-004: externalized configuration; ENC-006: secret exposure detection; ENC-007: key rotation) also map to these same objectives — demonstrating that practitioner-derived requirements and framework requirements converge on the same representation.

This example illustrates the normalization property (Section 4.4): ASVS, SSDF, and CIS requirements share primary objectives within ACO-SPC (notably ACO-SPC-001), providing a shared operational basis for the normalized concern of configuration security. Coverage is assessed once at the objective level: an organization that satisfies the mapped objectives addresses the underlying concern expressed by all three frameworks — subject to the correctness and granularity of the mapping. After the mapping step, downstream coverage analysis operates on the normalized objective layer, not on three separate framework-specific vocabularies.

5.2 Input Validation: ASVS, SSDF, and CAPEC

- ASVS v5.0: clusters injection_and_sanitization, input_contract_validation, validation_before_internal_use
- SSDF PW.5: “Create Source Code by Adhering to Secure Coding Practices”
- CAPEC-536: “Data Injected During Configuration”

All three converge on slice ACO-IVF (Input Validation, Safe Parsing & Controlled Failure):

Objective	Description	ASVS	SSDF	CAPEC
ACO-IVF-001	Entry-point input contract validation	✓	✓	—
ACO-IVF-002	Schema, type, and allowlist discipline	✓	✓	—
ACO-IVF-003	Dangerous-pattern exclusion	✓	✓	✓
ACO-IVF-004	Validation before internal use	✓	—	✓

ASVS provides the most granular decomposition. SSDF provides a broader practice-level mapping. CAPEC provides an attack-pattern perspective on the same concerns. AppSec Core absorbs all three into the same normalized concern, represented by overlapping objectives within the slice.

5.3 Supply Chain: SLSA, SSDF, and CAPEC

- SLSA Build Track: provenance, build integrity, platform isolation
- SSDF PW.3 / PS.2: verify third-party software, verify release integrity
- CAPEC-185/446/206: malicious download, malicious component, signing attack

All three converge on slice ACO-SCBI (Supply Chain & Build Integrity):

Objective	Description	SLSA	SSDF	CAPEC
ACO-SCBI-001	Dependency inventory and SBOM traceability	✓	✓	—
ACO-SCBI-002	Dependency risk analysis and policy gating	—	✓	✓
ACO-SCBI-003	Approved source and registry governance	✓	—	✓
ACO-SCBI-004	Trusted build definition and execution integrity	✓	—	—
ACO-SCBI-005	Promotion gates and approval governance	✓	✓	—
ACO-SCBI-006	Artifact provenance and signature integrity	✓	✓	✓

This is the most convergent slice: all three frameworks contribute to 4+ of 6 atomic objectives, despite originating from different domains (build integrity, development practices, attack patterns).

5.4 Convergence Metrics

Table 3 aggregates the convergence data from the three presented slices.

Table 3. Cross-framework convergence per slice (based on presented mapping tables).

Slice	Objectives shown	≥ 2 frameworks	≥ 3 frameworks	Max overlap
ACO-SPC	4	2 (50%)	1 (25%)	3
ACO-IVF	4	4 (100%)	1 (25%)	3
ACO-SCBI	6	5 (83%)	1 (17%)	3
Total	14	11 (79%)	3 (21%)	3

Across the 14 objectives analyzed, 79% are mapped by at least two frameworks, and each slice contains at least one objective mapped by all three frameworks in its group. The triple-convergence points — ACO-SPC-001 (secret leak prevention), ACO-IVF-003 (dangerous-pattern exclusion), ACO-SCBI-006 (artifact provenance) — represent core concerns where framework agreement is strongest and where canonical reduction provides the greatest value.

Single-framework objectives (3 of 14, 21%) occur at specialization edges: rotation discipline (ASVS only), operational identity binding (CIS only), trusted build definition (SLSA only). The ontology accommodates these framework-specific extensions alongside the shared core — it does not force convergence where the domain is genuinely specialized.

We note that these three slices were selected for highest cross-framework overlap; convergence rates for the remaining seven slices may be lower. The metrics above should be read as evidence of convergence on core concerns, not as a claim about the full ontology.

5.5 Coverage Summary

We present mapping examples for three slices (ACO-SPC, ACO-IVF, ACO-SCBI). The remaining seven slices have been validated through internal mapping exercises but are not presented here for space reasons.

Table 4 summarizes framework coverage across the full mapping exercise.

Table 4. Framework coverage summary.

Framework	Unit of analysis	Items mapped	Slices used	Items outside model
SSDF v1.1	Practices	20	8	0
ASVS v5.0	Thematic clusters	36	10	0
SLSA v1.0	Requirements	14	3	0
CIS v8.1 (AppSec scope)	Controls	8	6	0
CAPEC v3.9 (View 683)	Attack patterns	13	4	0
Total		91	10	0

Every external requirement mapped to at least one AppSec Core slice. No requirement fell outside the 10-slice model under the stated inclusion rules and mapping protocol (Section 4.2). This provides evidence — not proof — of *framework coverage*: the model accommodates these frameworks’ requirements.

We distinguish this from *domain coverage* — whether AppSec Core captures all relevant AppSec concerns. The latter would require validation against a broader set of practitioners, organizations, and domains (e.g., embedded systems, ML pipeline security), which is future work. The evidence here supports framework coverage for the analyzed sources; domain coverage remains an open question.

6. Structural Properties

6.1 Structural Invariance

Every slice exhibits the same structure: 7 objectives, practices linked via `objective_realized_by_practice`, mechanisms linked via `objective_implemented_by_mechanism`, artifacts linked via `objec-`

tive_expects_artifact, and a governing contract. This invariance has practical consequences:

- Tooling reuse: a tool that analyzes one slice works on all slices unchanged
- Uniform coverage analysis: the same algorithm determines coverage for any domain
- Predictable extension: new slices follow the same template

6.2 Framework Independence

AppSec Core was derived from practitioner semantics. This means:

- Adding a new framework (e.g., ASVS v6.0) requires mapping its requirements to existing objectives, not restructuring slices
- The 10-slice model reflects what practitioners do; framework vocabulary is absorbed, not adopted
- Security concerns evolve slowly; framework vocabularies evolve rapidly. The ontology models the former.

6.3 Extensibility

The contract-based model enables principled extension. New slices can be defined when the domain requires it — the current count of 10 is methodological, not ontological. Existing contracts prevent scope creep. The cross-slice vocabulary ensures new slices integrate cleanly.

6.4 What AppSec Core is Not

- Not a replacement for frameworks: organizations still use SSDF, ASVS, etc. for their specific purposes. AppSec Core provides the normalization layer for cross-framework analysis.
- Not a maturity model: it does not assess how well practices are performed.
- Not a vulnerability taxonomy: it models the solution space, not the problem space.
- Not a formal ontology in the OWL/description-logic sense [16, 17, 19]: it is a typed semantic model with explicit relations, optimized for practitioner use and tooling integration. The informal use of *ontology* follows Gruber’s sense of “a specification of a conceptualization” [16].

7. Discussion

7.1 Normalization vs. Classification

The distinction is precise:

- Classification assigns requirements to categories. Each requirement retains its framework-specific identity; the categories provide grouping but not reduction. Analysis still requires reading each framework’s vocabulary.
- Normalization performs canonical reduction: requirements from different frameworks that address the same security concern are reduced to shared ControlObjectives and their associated operational chains — shared practices, shared mechanisms, shared evidence — such that they become operationally equivalent for coverage analysis. The framework of origin becomes provenance metadata, not an input to downstream reasoning.

The evidence in Section 5 supports that AppSec Core performs normalization, not merely classification. SSDF PW.9, ASVS secure_configuration_base_line, and CIS-4 are three distinct vocabulary entries that, after mapping, resolve to the same normalized concern within ACO-SPC, anchored by overlapping ControlObjectives and shared operational chains. Downstream coverage analysis operates on the canonical chain — not on three separate framework-specific assessments. This is canonical reduction,

not grouping.

7.2 Towards Standardization

AppSec Core is proposed as a candidate for standardization, not as a finished standard. The path would require:

1. Multi-organization derivation: the current ontology reflects one team's experience. Cross-organizational validation would test whether the 10-slice model generalizes.
2. Requirement-level alignment: v0 maps at the practice/cluster level. A standard would require finer-grained alignment.
3. Community governance: slice contracts are designed to support community-driven extension, but this has not been tested.
4. Formal specification: the current YAML representation may need formalization (e.g., SHACL [18], OWL [17]) for interoperability, though accessibility must be weighed against formality.

7.3 Practical Benefits

If AppSec Core or a similar model were adopted:

- Unified compliance surface: one coverage analysis for N frameworks
- Version resilience: framework updates require re-mapping, not restructuring
- Tool interoperability: security tools could align on shared identifiers
- Cross-organization comparison: organizations using the same normalization layer could compare coverage without exposing proprietary documentation

7.4 The Practitioner-Derived Argument

A normalization ontology must be independent of the frameworks it normalizes; otherwise it is a biased projection. AppSec Core avoids this by being derived from what practitioners actually do — documented before many of the frameworks it now normalizes.

The SbD-ToE corpus draws on a prescriptive Security by Design framework first documented in practitioner deployment in 2018 [19], predating the publication of SSDF v1.0 (2021), SLSA v1.0 (2021), and CIS Controls v8 (2021). Several concerns now formalised in those frameworks — including risk-proportional requirement selection and multi-framework normalisation into a unified prescriptive activity model — were already operationalised and documented in client-facing practice from that period. This does not predate the underlying security domain knowledge; it predates the formal framework publications. The frameworks codified concerns that practitioners had already identified and implemented.

More recent examples corroborate the same pattern at the individual-concept level: secure configuration baselines (CFG-001→007) were documented in the corpus in 2023; ASVS v5.0 formalised `secure_configuration_baseline` in 2025. Container admission policies were implemented in practice in 2022; SLSA formalised build platform isolation in 2023.

This temporal precedence supports the independence claim: AppSec Core was not reverse-engineered from frameworks to fit their structure. It was abstracted from operational practice, and the frameworks independently converged on the same domain concerns.

8. Limitations

Single-team derivation. The ontology reflects one team’s 10+ years of experience. The 10-slice structure may not capture concerns unique to domains not encountered by this team (e.g., embedded systems, ML pipeline security, mobile-native development). We argue for structural transferability — the typed schema is domain-independent; the slices may need adjustment — but this has not been validated.

V0 granularity. External alignment operates at the practice/cluster level, not at individual requirement level. The convergence evidence in Section 5 uses this granularity.

No formal specification. The ontology is represented in YAML with explicit types and relations. This is a deliberate trade-off between practitioner accessibility and formal interoperability. A formal OWL or SHACL representation is future work.

Evaluator subjectivity. The mapping exercise was performed by the ontology’s authors. Inter-rater reliability with independent evaluators was not measured.

Temporal specificity. The framework landscape is captured as of early 2026.

9. Conclusion

We have presented AppSec Core, a normalized ontology for application security with 10 domain slices and 234 typed instances. The ontology provides:

- Measured cross-framework convergence: in the presented slices, 79% of analyzed ControlObjectives are mapped by at least two frameworks, supporting the claim that core AppSec concerns can be reduced into shared objectives despite vocabulary differences
- Structural invariance: every slice uses the same four-type schema and relation model
- Framework independence: derived from practitioner semantics, stable as frameworks evolve
- Contract-based governance: each slice governed by a formal contract supporting extension

The evidence supports that AppSec Core functions as a normalization layer, not merely a taxonomy. It provides a canonical reduction layer: heterogeneous framework requirements addressing the same security concerns are reduced to shared ControlObjectives and their associated operational chains. Downstream coverage analysis operates on this normalized objective layer independently of framework vocabulary. This is canonical reduction — the defining property of normalization — not grouping or alignment. The reduction does not claim full normative equivalence between frameworks; it claims a shared operational basis sufficient for unified coverage analysis.

To our knowledge, the application security domain lacks a shared semantic model for cross-framework normalization. AppSec Core is our proposal for this model. It is not a replacement for existing frameworks — it is a candidate for the canonical layer between them.

10. Artifact Availability

Curated supporting artifacts for this paper are available in the companion public repository at <https://github.com/sbd-ai-runtime/appsec-core-ontology-research>. For this paper, the relevant materials are organized under `papers/01-appsec-core-normalized-ontology/artifacts/`, notably `papers/01-appsec-core-normalized-ontology/artifacts/ontology/`, `papers/01-appsec-core-normalized-ontology/artifacts/schema/`, and `papers/01-appsec-core-normalized-`

ontology/artifacts/slice_contracts/. The same repository also contains this paper’s curated source under papers/01-appsec-core-normalized-ontology/source/ and its public PDF under papers/01-appsec-core-normalized-ontology/pdf/.

References

- [1] NIST. Secure Software Development Framework (SSDF) Version 1.1. SP 800-218. 2022.
- [2] OWASP. Application Security Verification Standard (ASVS) v5.0.0. 2025.
- [3] SLSA. Supply-chain Levels for Software Artifacts. Build Track v1.0. 2023.
- [4] CIS. Center for Internet Security Controls v8.1. 2024.
- [5] European Union. Digital Operational Resilience Act (DORA). Regulation (EU) 2022/2554. 2022.
- [6] European Union. Network and Information Security Directive (NIS2). Directive (EU) 2022/2555. 2022.
- [7] A. Herzog, N. Shahmehri, and C. Duma, “An ontology of information security,” *Int. J. Inf. Security and Privacy*, vol. 1, no. 4, pp. 1–23, 2007.
- [8] C. Blanco, J. Lasheras, E. Fernández-Medina, R. Valencia-García, and A. Toval, “Basis for an integrated security ontology according to a systematic review of existing proposals,” *Computer Standards & Interfaces*, vol. 33, no. 4, pp. 372–388, 2011, doi: 10.1016/j.csi.2010.12.002.
- [9] MITRE. Common Weakness Enumeration (CWE). Available: <https://cwe.mitre.org> (accessed: 2026-04-06).
- [10] MITRE. Common Attack Pattern Enumeration and Classification (CAPEC). Available: <https://capec.mitre.org> (accessed: 2026-04-06).
- [11] OWASP. Software Assurance Maturity Model (SAMM) v2.1. 2024.
- [12] Synopsys. Building Security In Maturity Model (BSIMM) 13. 2023.
- [13] NIST. Security and Privacy Controls for Information Systems and Organizations. SP 800-53 Rev. 5. 2020.
- [14] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt, “A comparison of security requirements engineering methods,” *Requirements Engineering*, vol. 15, no. 1, pp. 7–40, 2010, doi: 10.1007/s00766-009-0092-x.
- [15] A. Goknil, I. Kurtev, K. van den Berg, and J.-W. Veldhuis, “Semantics of trace relations in requirements models for consistency checking and inferencing,” *Software & Systems Modeling*, vol. 10, no. 1, pp. 31–54, 2011, doi: 10.1007/s10270-009-0142-3.
- [16] T. R. Gruber, “A translation approach to portable ontology specifications,” *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993, doi: 10.1006/knac.1993.1008.
- [17] W3C. OWL 2 Web Ontology Language: Document Overview (Second Edition). W3C Recommendation, 2012. Available: <https://www.w3.org/TR/owl2-overview/> (accessed: 2026-04-06).
- [18] W3C. Shapes Constraint Language (SHACL). W3C Recommendation, 2017. Available: <https://www.w3.org/TR/shacl/> (accessed: 2026-04-06).
- [19] P. Farinha, “Framework Shiftleft — Security by Design,” Shiftleft internal technical documentation,

2018. Multiple client deployment instances. Available from the authors upon request.