Strobilus: Enriching Cedar with Stateful Policies

Massimiliano Baldo
IMT School for Advanced Studies Lucca
Lucca, Italy
massimilano.baldo@imtlucca.it

Pietro Di Gianantonio
University of Udine
Udine, Italy
pietro.digianantonio@uniud.it

Matteo Paier
IMT School for Advanced Studies Lucca
Lucca, Italy
matteo.paier@imtlucca.it

Marino Miculan
University of Udine
Udine, Italy
marino.miculan@uniud.it

Abstract

Authorization is a fundamental problem in modern distributed systems, and the "policies-as-code" paradigm has emerged as a promising solution to decouple access control logic from application code. However, most policy languages lack the ability to handle stateful policies directly. This limitation forces developers to manage policy-related state within the application code, reintroducing the very coupling that policies as code aims to eliminate and opening the door to security vulnerabilities.

To address this gap, we introduce *Strobilus*, a language designed to express effects over policy-specific data. Strobilus is built to seamlessly complement and integrate with Amazon's Cedar, allowing developers to write stateful policies without modifying Cedar's core syntax or evaluation engine. Strobilus is distinguished by its formal semantics, a strong typing system, and a guarantee of termination, which facilitates rigorous analysis and verification of policies. We have developed a prototype implementation in Rust, which demonstrates that Strobilus may lead to significant performance improvements over external methods for policy data management. This approach aims to fully realizes the promise of "policies-as-code" by providing a comprehensive, safe, and verifiable solution for both stateless and stateful authorization policies.

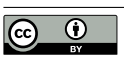
CCS Concepts

• **Security and privacy** → *Formal methods and theory of security*; **Authorization**; **Access control**; • **Theory of computation** → *Semantics and reasoning*.

Keywords

Access Control, Policy languages, Stateful Policies, Formal methods

ACM Reference Format:
Massimiliano Baldo, Pietro Di Gianantonio, Matteo Paier, and Marino Miculan. 2026. Strobilus: Enriching Cedar with Stateful Policies. In *Proceedings of the 31st ACM Symposium on Access Control Models and Technologies (SACMAT '26), July 08–10, 2026, Waterloo, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3750555.3811892



SACMAT '26, Waterloo, ON, Canada

ACM ISBN 979-8-4007-2107-6/2026/07
https://doi.org/10.1145/3750555.3811892

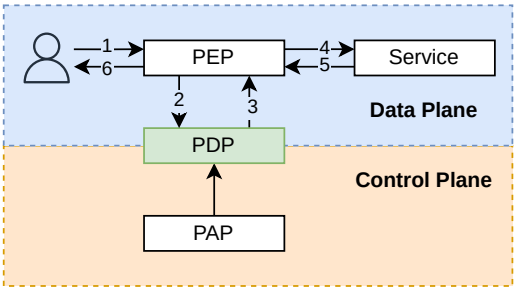


Figure 1: The PEP/PDP architecture.

1 Introduction

Access control systems serve as the primary defense for modern distributed systems by regulating data access. Traditionally, access control logic has been embedded directly into application code. This approach has significant drawbacks. First, it is highly error-prone; simple coding mistakes can easily lead to serious security vulnerabilities. This method also makes permissions difficult to understand and audit, as verifying access rules requires a deep dive into the application's code. Finally, embedded permissions are hard to maintain: any change to the authorization logic means modifying the application's codebase, often in multiple places, which increases the risk of introducing new errors.

To address these issues, the *policies-as-code* paradigm [18] has emerged as a robust alternative for managing authorization. This approach aims to decouple security logic from the application's core code by externalizing access rules into a dedicated *domain-specific language* (DSL). Such separation not only simplifies the development lifecycle but also significantly enhances the auditability, maintainability, and cross-platform consistency of security postures.

In cloud-native environments, this is typically implemented using the *Policy Enforcement Point* (PEP)/*Policy Decision Point* (PDP) architecture (Figure 1). Under this model, the PEP intercepts an incoming request and forwards it to the PDP for evaluation; the PDP analyzes the request against established policies and contextual data (e.g., user attributes) to issue a definitive *Allow* or *Deny* verdict; the PEP then enforces this outcome, granting or blocking access to the service accordingly. A *Policy Administration Point* (PAP) serves as a management hub, allowing administrators to update rules globally without modifying the underlying code.

Over the last three decades, many policy languages have been introduced, including PONDER [9], XACML [1], Rego [17], and

OpenFGA [16]. A more recent and noteworthy addition is *Cedar* [7], a high-performance, open-source authorization language developed by Amazon and used in AWS. It enables complex access rules and includes a robust typing system for safety. Most importantly, Cedar possesses a formal, clearly defined semantics. This solid foundation allows for the formal proof of language properties, enables verification of the engine's implementation (reducing the Trusted Computing Base), and supports tools for policy analysis.

However, these policy languages are almost always *side-effect free*: they operate on a "read-only" basis, consuming input data to produce a binary Allow/Deny verdict without modifying the environment or the underlying data store. This design choice creates a significant gap: the inability to express *stateful policies*, that is, rules that depend on and *must* update, historical context.

In practice, security requirements often depend on historical context and cumulative actions. Rate limiting, for instance, requires tracking access counts and timestamps; threshold-based policies may block actions only after a set number of failed attempts or a specific event sequence; history-based models may deny access if a subject has previously interacted with a conflicting entity. All such constraints require a persistent access history log that can be updated and queried during the decision-making process.

In all these cases, when developers use strictly stateless languages to enforce stateful requirements, they are forced to shift the state-management logic back into the application code. This creates a "leaky abstraction" where the application developer becomes responsible for updating policy-related data outside the formal decision loop. This shift reintroduces the very coupling that the policies-as-code paradigm aims to eliminate, blurring the line between business logic and security specifications.

Some frameworks, like XACML and its variants, attempt to bridge this gap using *obligations*, i.e, mandated actions triggered by a permission grant. However, this approach is unsatisfactory for two primary reasons. First, obligation execution is still delegated to the application layer, leaving the policy engine "blind" to whether the state was actually updated correctly. Secondly, most obligation mechanisms lack a formally defined syntax and semantics. This ambiguity makes it nearly impossible to perform rigorous formal verification or systematic security analysis, as the state transitions occur in an unverified "black box" at the application level.

The inability to manage and update policy-related state within the trusted policy domain represents a critical deficiency in current authorization models, leaving a gap between high-level security requirements and their technical enforcement.

For this reason, we advocate for *Domain Specific Languages for policy-oriented state management*. These DSLs are not intended to replace existing stateless policy languages but to complement and integrate with them. By doing so, we aim to fully achieve the "policies-as-code" ideal, thereby realizing a complete and clear separation between application logic and policy specification.

In particular, in this paper we introduce *Strobilus*, a DSL designed specifically for expressing effects over policy-specific data. Strobilus seamlessly integrates with Cedar, operating within the same entity and variable space. This tight coupling allows developers to write stateful policies directly by leveraging Strobilus alongside Cedar, all without needing to modify Cedar's syntax or evaluation engine. As a result, services are no longer burdened with managing policy-specific state in their own code, which ensures a clean separation between business logic and policy specification.

Like Cedar, Strobilus is built on a strong foundation. It is equipped with a formal, clearly defined semantics, which provides a precise reference for both the implementation of the language and for the development of specialized tools. Moreover, it features a robust and expressive typing system that guarantees sound executions, preventing type errors at runtime. The correctness of these results is guaranteed by the formal development in the Lean proof assistant.

We have developed a prototype implementation of Strobilus in Rust and evaluated its performance using, as a case study, a client-server application from the Cedar showcase. Our results demonstrate that Strobilus can achieve greater efficiency than implementations using Cedar and manually handling policy state within the application layer. By internalizing state management, Strobilus streamlines the data lifecycle and eliminates the overhead of complex, error-prone application-side logic.

The rest of this paper is structured as follows. In Section 2 we briefly recall Cedar. Section 3 gives an overview of Strobilus and shows how it can be used to implement stateful policies. Section 4 presents the formal syntax and semantics of Strobilus, along with key theoretical results. Section 5 details our prototype implementation of Strobilus. Section 6 provides an evaluation of Strobilus, and Section 7 some comparison with related work. Finally, Section 8 concludes the paper and outlines future research.

2 Background: the Cedar language

Cedar [7] is a domain-agnostic policy language which enables the concise modeling of complex authorization scenarios. Policies in Cedar are represented as declarative programs that include control flow, logical and set-based operations.

When evaluated, each Cedar policy produces a boolean expression that results in one of two effects: *permit* or *forbid*. The final decision is based on evaluating this expression against three core components: `principal` (the entity making the request), `action` (the specific operation being requested), and `resource` (the object the operation is being performed on). An optional *condition* can be included, which is an expression block introduced by a `when` or `unless` keyword, allowing for more fine-grained control over when a policy is applied.

An example of a Cedar policy is shown in Listing 1. The policy expresses the following behavior: "Permit all users of the *janeFriends* group to view photos of *janeTrips* album".

```
permit (
  principal in Group::"janeFriends",
  action == Action::"ViewPhoto",
  resource in Album::"janeTrips"
);
```

Listing 1: Example of Cedar policy.

A request is ***allowed*** only if two conditions are met:

(1) At least one policy with a *permit* effect evaluates to true.
(2) No policy with a *forbid* effect evaluates to true.

If either of these conditions fails, the request is ***denied***.

In order for evaluate policies, application data must be available. In Cedar, this data is represented as *entities*, which are elements of

the application domain used for authorization purposes, such as users, objects, processes, or other relevant resources.

Each entity has three components:

(1) An `entityUID`, a unique identifier that consists of:
   (a) an `entityType`, which defines the type of the entity (e.g., `User`, `Photo`);
   (b) an `identifier`, a string that represents a specific instance of the entity (e.g., `"Alice"`, `"2213"`);
(2) A set of *attributes*, i.e., generic key-value pairs;
(3) A set of *parents*, which are other `entityUIDs` that are in a hierarchical relationship with the entity.

Entities sets can be represented by means of JSON files, like the following.

```
[{
  "uid": {"type":"User", "id":"user2342"},
  "attrs": {"name": "Joe"},
  "parents": [{"type":"Group", "id":"janeFriends"}],
},
{
  "uid": {"type":"Group", "id":"janeFriends"},
  "attrs": {},
  "parents: {}
}]
```

Listing 2: Example of Cedar entities

Cedar introduces also the concept of *schema* for defining the valid structure of an authorization system. A schema specifies the structure of entities, including their types, permitted and required attributes, and allowed parent relationships. It also dictates which actions can be performed by which principals on which resources, thereby ensuring consistency and correctness across the entire authorization model. A schema for the policy in Listing 1 and the entities in Listing 2 is shown in Listing 3.

```
{
  "ExampleApp": {
    "entityTypes": {
      "User": {
        "shape": {
          "type": "Record",
          "attributes": {
            "name": { "type": "String" }
          }
        }
      },
      "Group": {},
      "Album": {}
    },
    "actions": {
      "ViewPhoto": {
        "appliesTo": {
          "principalTypes": ["User"],
          "resourceTypes": ["Album"]
}}}}}
```

Listing 3: Example of Cedar schema

Schemas are used to validate policies and ensure runtime safety, through a type-checking system that combines *singleton types* and *capabilities* [3, 6]. Singleton types take advantage of short-circuit evaluation in Boolean expressions: if the outcome of an expression is already known, the type checks for subsequent sub-expressions can be skipped, leading to reduced validation time. Capabilities are used to statically track the possible attributes an entity might possess, allowing for the validation of a broader class of policies.

EntTypes	$E \in \mathbf{ID}$
Attributes	$f \in \mathbf{ID}$
Strings	s
Int	i
Var	$x ::=$ principal | action | resource | context
Entities	$u ::= E::s$
Record	$r ::= \{f_1 : v_1, \ldots, f_n : v_n\}$
Val	$v ::= u \mid$ true $\mid$ false $\mid s \mid i \mid [v_1, \ldots, v_n] \mid r$
Expr	$e ::= v \mid x \mid [e_1, \ldots, e_n] \mid \{f_1 : e_1, \ldots, f_n : e_n\} \mid$ $e.f \mid e_1$ && $e_2 \mid e_1 \parallel e_2 \mid !e \mid -e \mid$ isEmpty $e \mid e_1$ *bop* $e_2 \mid e$ like $s \mid e$ has $f \mid e$ is E $\mid i * e \mid$ if e_1 then e_2 else e_3
Bin.op.	$bop ::= + \mid - \mid < \mid \leq \mid == \mid$ in $\mid$ contains $\mid$ containsAny $\mid$ containsAll

Table 1: Cedar syntax.

Policies can be decorated with *annotations*, i.e., key-value pairs of the form `@key("value")`. They have no direct impact on the authorization decision made by the Cedar engine, but they can carry additional metadata to be used by domain-specific code.

Cedar: syntax and semantics. Formally, a Cedar policy comprises an *effect*–either `permit` or `forbid`—and a *body*, which is a Boolean expression defining the conditions under which that effect is applied. This expression evaluates a request by referencing four primary variables: `principal`, `action`, `resource`, and `context`. These variables identify the requester, the target operation, the resource being accessed, and any supplementary request-time metadata, respectively. Each of these parameters is bound to an *entity*. Entities possess a defined set of *attributes*, which policy expressions manipulate using standard arithmetic, logical, and comparison operators.

The formal syntax for the various syntactic categories is in Table 1. Here, **ID**, Var, Val, and Bop denote the sets of Cedar identifiers, variables, values, and binary operators, respectively.

The evaluation of Cedar expressions is defined by a judgment[1]

$$\mu, \sigma \vdash e \Downarrow v$$

meaning that expression e evaluates to the value v in the *entity store* μ and *authorization request* σ. The *entity store* μ describes the state of the system, and it is a finite map

$$\mu : Entity \rightharpoonup Record \times \mathcal{P}(Entity)$$

associating to each entity a record value and the set of its ancestors in the hierarchy. The *authorization request* σ describes the current request, and it is a tuple

$$\sigma \in Entity \times Entity \times Entity \times Record$$

representing, respectively, the `principal`, `action`, `resource`, and `context`.

Evaluation is defined inductively over e by a set of rules, following "short-circuit" semantics, e.g.:

$$(\textsc{EvalAnd1})\ \frac{\mu, \sigma \vdash e_1 \Downarrow false}{\mu, \sigma \vdash e_1 \,\&\&\, e_2 \Downarrow false}$$

[1] In this presentation, we follow the big-step semantics corresponding to the Cedar specification in Lean, rather than the small-step semantics presented in [7].

$$(\textsc{EvalAnd2})\ \frac{\mu, \sigma \vdash e_1 \Downarrow true \quad \mu, \sigma \vdash e_2 \Downarrow v \quad v \in Bool}{\mu, \sigma \vdash e_1 \,\&\&\, e_2 \Downarrow v}$$

As another example, the rules for determining whether an entity is a descendant of another are as follows:

$$(\textsc{EvalIn}_e1)\ \frac{\mu, \sigma \vdash e_1 \Downarrow E :: s \quad \mu, \sigma \vdash e_2 \Downarrow E :: s}{\mu, \sigma \vdash e_1 \text{ in } e_2 \Downarrow true}$$

$$(\textsc{EvalIn}_e2)\ \frac{\mu, \sigma \vdash e_1 \Downarrow E_1 :: s_1 \quad \mu, \sigma \vdash e_2 \Downarrow E_2 :: s_2 \quad \mu(E_1 :: s_1) = (\_, S_1) \quad E_2 :: s_2 \in S_1}{\mu, \sigma \vdash e_1 \text{ in } e_2 \Downarrow true}$$

$$(\textsc{EvalIn}_e3)\ \frac{\mu, \sigma \vdash e_1 \Downarrow E_1 :: s_1 \quad \mu, \sigma \vdash e_2 \Downarrow E_2 :: s_2 \quad E_1 :: s_1 \neq E_2 :: s_2 \quad \mu(E_1 :: s_1) = (\_, S_1) \quad E_2 :: s_2 \notin S_1}{\mu, \sigma \vdash e_1 \text{ in } e_2 \Downarrow false}$$

For the remaining rules, we refer to [7] and the Lean formalization at https://github.com/cedar-policy/cedar-spec.

Finally, *policy sets* can be defined, without loss of generality, as follows:

$$P ::= \langle\rangle \mid P,\ \text{permit when } e \mid P,\ \text{forbid when } e$$

where e is an expression as defined in Table 1. (Actual Cedar policies use some additional syntactic structure, but they can be straightforwardly rewritten in the above syntax.)

The evaluation of a Cedar policy sets for a given authorization request σ in an entity store μ is defined by:

$$(\textsc{Policy})\ \frac{\mu, \sigma \vdash eval(P) \Downarrow v}{\text{allow}(\mu, \sigma, P) \Downarrow v} \tag{1}$$

where the *eval* function converts a policy set into a single boolean expression, extracting all bodies and joins all the constraints:

$$eval(P) = \left(\bigvee_{(\text{permit when } e) \in P} e \right) \&\&\, ! \left(\bigvee_{(\text{forbid when } e') \in P} e' \right)$$

3 Strobilus: an overview

As highlighted in the introduction, the inability of stateless languages (such as Cedar) to encode state-dependent rules compels developers to embed state-management logic directly within the application. This violates the fundamental principle of separation of concerns and undermines the very decoupling of policy and business logic that the "policies-as-code" paradigm aims to achieve.

To address these limitations, we introduce *Strobilus*, a DSL specifically designed to express state modifications in conjunction with Cedar policies. A Strobilus program consists of two *obligation clauses* structured as follows:

```
on allow { ... }
on deny { ... }
```

These clauses define the stateful operations to be executed after a Cedar policy evaluation reaches a verdict, but before that decision is enforced. This ensures that the system state is updated in direct response to the authorization outcome, maintaining a strictly synchronized and trusted decision loop.

The obligation language is a lightweight imperative DSL designed to directly manipulate the underlying data store utilized by Cedar for policy evaluation: the collection of entities, their attributes, and their hierarchical relationships.

We now provide illustrative examples of how Strobilus complements Cedar policies to enforce state-dependent requirements.

Example 3.1 (Limited free service). Consider a SaaS application that implements a usage-limited free tier. To enforce such a constraint, we must persistently track the number of requests consumed by each user and deny access once the quota is exhausted. This scenario highlights the necessity for a stateful policy capable of maintaining and updating an access counter over time.

A corresponding Cedar policy for this logic is shown in Listing 4.

```
permit ( ... )
when {
  principal.counter > 0
}
```

Listing 4: Cedar policy for limited free service

While this policy correctly evaluates the authorization decision, it remains purely reactive; it cannot modify the state it relies upon. To close the loop, the counter attribute must be decremented each time access is granted. The Strobilus obligation associated with this policy is defined in Listing 5.

```
on allow {
  updateAttribute(principal, "counter", principal.
       counter - 1);
}
```

Listing 5: Strobilus obligation for limited free service

The `updateAttribute` primitive facilitates the direct modification of an entity's state. This command accepts three arguments: the entity identifier (UID), the attribute name, and the new value. Strobilus allows these commands to reference Cedar's native variables, such as `principal`, ensuring the update is applied to the specific subject of the request. The value expression is evaluated using Cedar's engine, with Strobilus guaranteeing that the resulting state transition is applied atomically within the trusted decision domain. □

Example 3.2 (Service Access Security). In modern (micro)service architectures, services often operate at distinct security tiers, such as *secure* and *insecure*. While a secure service may interact freely with its peers, any communication with an insecure service must trigger an immediate revocation of its privileged status. This "taint" logic limits the potential attack surface: if an insecure service is compromised, any interacting service is deemed untrusted and moved to a lower privilege tier.

In Cedar we can model these permissions using *group memberships*, as shown in Listing 6.

```
permit (
  principal in Group::"secure",
  resource in Group::"secure"
);
permit (
  principal in Group::"insecure",
  resource in Group::"insecure"
);
permit (
  principal in Group::"secure",
  resource in Group::"insecure"
);
```

Listing 6: Cedar policy for service access security

However, this lacks the capacity to perform the subsequent state transition (i.e., the group migration). In a purely stateless language, this revocation must be delegated to external application logic.

To bridge this gap, we utilize Cedar's possibility to annotate policies with key-value tags. Specifically, we leverage a `strobilus_id` annotation, which Strobilus tracks using two specialized entities: `Justification::"Permits"` and `Justification::"Forbids"`. The former contains a `satisfied` attribute (a `Set String`) populated with the `strobilus_id` values of all permit policies that evaluated to true. Its `unsatisfied` counterpart tracks those that evaluated to false. Dually, `Justification::"Forbids"` tracks the `satisfied` and `unsatisfied` IDs for all `forbid` policies.

This architecture creates a formal, auditable interface between a policy verdict and its side effects. By querying these entities, a Strobilus program can determine exactly which rules triggered a decision and react accordingly. The obligation for managing these security tier updates is in Listing 7.

```
on allow {
  if (Justification::"Permits".satisfied.contains("taint
      ")) {
    removeParent(principal,Group::"secure");
    addParent(principal, Group::"insecure");
  }
}
```

Listing 7: Strobilus obligation for the secure service access scenario.

In this implementation, if an access request is granted specifically by a policy tagged with the "taint" ID, the obligation executes an atomic state change: the principal is removed from the `secure` group and added to the `insecure` group. Conversely, safe interactions (governed by policies lacking the "taint" annotation) result in an allow decision that leaves the entity store unchanged. □

Strobilus further enables "maintenance" updates on entities, which in stateless systems are typically buried within application logic. By treating the entity store as an *internal, protected* state, Strobilus ensures that all modifications must instead occur as obligations triggered by specific requests. This architecture preserves the separation of concerns and guarantees that all state changes are controlled and auditable within the policy domain.

Example 3.3 (Expired accounts clean-up). Consider the administrative task of periodically decommissioning expired accounts within an organization. Traditionally, such maintenance requires external scripts that bypass the policy engine; however, Strobilus allows this logic to be encapsulated as an internal obligation. Listing 8 illustrates an obligation that iterates through existing accounts to identify and remove those exceeding their expiration threshold.

```
on allow {
  if (action == Action::"updateaccounts") {
    for x in accounts do {
      if x.expiration_date > now {
        removeParent(x, Group::"active");
        addParent(x, Group::"expired");
      }
    }
  }
}
```

Listing 8: Strobilus obligation for account maintenance

This maintenance routine is triggered by a specific administrative request, such as `updateaccounts`, governed by a Cedar policy:

LocVar	$y \in \mathbf{ID}$
Expr	$e ::= y \mid \ldots$
BasCom	$bc ::=$ updateEntity(e, r, v) | removeEntity(e) | addParent(e_1, e_2) | removeParent(e_1, e_2) | updateAttribute(e_1, f, e_2) | removeAttribute(e, f)
Command	$c ::= bc \mid$ skip $\mid c_1; c_2 \mid$ if e then c_1 else $c_2 \mid$ for y in e do $c \mid \{c\}$
PolObl	$Po ::= P$ [on allow c_1][on deny c_2]

Table 2: Strobilus syntax.

```
permit(
  principal == Role::"Admin",
  action == Action::"updateaccounts",
  resource == Resource::"AccountStore"
);
```

Listing 9: Cedar policy rule for account maintenance

The Strobilus approach is directly applicable to any domain where authorization decisions depend on mutable, policy-managed state. As a concrete example, consider a *GDPR-compliant data access* scenario. A subject may request access to their personal records held by an organization. Under GDPR, such requests are bounded: once a subject has exercised the right of access, the controller may lawfully defer further identical requests for a reasonable period. Enforcing this rule requires tracking, per subject, the date of the last satisfied request. With Strobilus, this counter is an attribute (e.g., `last_access_date` on the `DataSubject` entity) that is atomically read and updated by a Strobilus obligation during the same authorization decision, without any application-side state management. The Cedar policy denies the request if the date is too recent; the Strobilus obligation updates it upon each *permit*. This end-to-end integration eliminates the risk of the application failing to update the attribute after authorization, a classic source of GDPR non-compliance bugs. Similarly, in a *secure document sharing* scenario, policies such as "a document may be forwarded at most k times" or "a confidential file may not be accessed after the project deadline" require mutable access-count and timestamp attributes. Both are naturally expressed and enforced within Cedar+Strobilus, with no leakage of state management into application code.

4 Strobilus: syntax, semantics and types

In this Section we introduce the formal syntax, semantics and type system of Strobilus. The syntax of Strobilus is obtained by extending that of Cedar (Table 1) with a new syntactic category *LocVar* of *local variables*, which can be used in expressions, and new syntactic categories for commands and policy sets with obligations are introduced, as defined in Table 2.

Specifically, Strobilus provides primitive commands for the dynamic instantiation of entities and the modification of their internal state, including attribute updates and parent-child reassignments. These primitives can be structured into complex logic with usual control flow constructs, such as sequential composition, conditional branching, and `for`-loops over finite sets.

4.1 Operational Semantics

The judgment for the big-step operational semantics of Strobilus has the form

$$\sigma \vdash \langle \mu, c \rangle \Downarrow \mu'$$

where c is the command to be executed, the store μ represents the initial entity store, and μ' denotes the entity store after the execution of c. Unlike in Cedar, σ in this context, is a general environment that defines both the possible local variables and the parameters in the request associated with the obligation in which c appears.

The rules for Strobilus commands are in Figure 2. The rules for basic commands show that execution proceeds by first evaluating the command arguments and then updating the entity store with the resulting values. The rules for command composition and if-then-else are standard. The rule for the for loop updates the environment to evaluate the loop body in a modified context and assumes an implicit order on the elements of a set. A cleaner approach would be to introduce a list type and iterate over its elements, but this would require more substantial changes to the Cedar part of the language, which we prefer to leave unchanged.

Then, the result of the evaluation of a policy set P with obligations c_1, c_2, for a request σ in the entity store μ, is defined by:

$$(\textsc{Obl})\ \frac{\mu, \sigma \vdash \mathit{eval}(P) \Downarrow v \quad \sigma \vdash \langle \mu, \mathsf{if}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rangle \Downarrow \mu'}{\mathsf{result}(\mu, \sigma, P\ \mathsf{on\ allow}\ c_1\ \mathsf{on\ deny}\ c_2) \to (v, \mu')}$$

4.2 Type system

In Strobilus, similarly to Cedar, if the evaluation of a request under a policy results in a runtime error, the request is denied. This mechanism ensures safe handling of runtime errors but may also lead to unintended denials of access requests.

Cedar's type system guarantees error-free policy evaluation on well-formed entity stores and requests, by validating the policy set against a *schema* (M, S) that specifies entity structure, hierarchies, and admissible request forms. In detail, a schema is a pair (M, S), where M maps each entity type E to a pair (τ, H), with τ the record type of E's attributes and H the set of possible ancestor entity types. The component S maps each action entity A (of the form $\mathsf{Action} :: s$) to a triple (A_p, A_r, A_x), where A_p and A_r are the sets of allowable principal and resource entity types for A, and A_x is the type of context records.

A policy set P, an entity store μ, and a request σ are said to *conform* to a schema (M, S) if all types and structures involved are consistent with the schema. Under this condition, and provided that μ contains all necessary entities, the evaluation of P with μ and σ is guaranteed to proceed without runtime errors.

Typing judgements for expressions take the form

$$\alpha, \Gamma \vdash e : \tau, \epsilon$$

where α is the set of *capabilities* that is, optional attributes assumed to be present in entities inside the entity store; Γ is the request environment specifying the types of the request variables; e is the expression being typed; τ its inferred type; and ϵ the attributes that are guaranteed to be present if e evaluates to `True`.

As in the original type system of Cedar [7], the type rules make use of of the schema (M, S) as an implicit extra parameter. In more detail, syntax of Cedar types and schemas is given in Table 3.

SimpleType	$\sigma ::= \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{Set}\ \tau$
RecType	$A ::= \{\omega_1 f_1 : \tau_1, \ldots, \omega_n f_n : \tau_n\}$
Type	$\tau ::= \sigma \mid E \mid A$
Qual	$\omega ::= \_ \mid ?$
EntSch	$M ::= \langle\rangle \mid M, E : (A, H)$
Ancstrs	$H ::= [E_1, \ldots, E_n]$

Table 3: Syntax of Cedar types and schemas.

The set of basic types is mostly standard; notice that, there are two singleton types, True and False, which are subtypes of Bool. Singleton types are assigned to expressions that always evaluate to true or false, respectively. Entity types are always record types, and record types can have optional fields, which are present only in some instances of the entity type.

To correctly type-check access to an optional attribute of an entity, one must use expressions of the following kind

$$\mathsf{if}\ E :: s\ \mathsf{has}\ f\ \mathsf{then}\ e.f\ \mathsf{else}\ \mathit{alt}$$

where *alt* is the value to be used if the attribute f is not present in $E :: s$. The typing rules to be used in the above case are

$$(\textsc{TypeIf3})\ \frac{\alpha; \Gamma \vdash e_1 : \mathsf{Bool}; \varepsilon_1 \quad \alpha \cup \varepsilon_1; \Gamma \vdash e_2 : \tau_2; \varepsilon_2 \quad \alpha; \Gamma \vdash e_3 : \tau_3; \varepsilon_3 \quad \tau = \tau_2 \sqcup \tau_3}{\alpha; \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau; (\varepsilon_1 \cup \varepsilon_2) \cap \varepsilon_3}$$

$$(\textsc{TypeHasAttr5})\ \frac{\alpha; \Gamma \vdash e : E; \varepsilon \quad M(E) = (\{\ldots, ?f : \tau, \ldots\}, \_) \quad (e, f) \notin \alpha}{\alpha; \Gamma \vdash e\ \mathsf{has}\ f : \mathsf{Bool}; (e, f)}$$

$$(\textsc{TypeGetAttr3})\ \frac{\alpha; \Gamma \vdash e : E; \varepsilon \quad M(E) = (\{\ldots, ?f : \tau, \ldots\}, \_) \quad (e, f) \in \alpha}{\alpha; \Gamma \vdash e.f : \tau; (e, f)}$$

As an example, the type derivation in Figure 3 shows that, by using the mechanism of capabilities, the type system can correctly type-check such expressions.

The Strobilus type system extends Cedar's formal foundations by defining a schema-conformance relation for obligations. This extension necessitates two primary modifications to the Cedar type system. First, the grammar is extended with a unit type, denoted as $*$, which is assigned to commands that execute without returning a value. Secondly, because both expressions and commands may introduce local variables, the traditional request environment Γ is generalized into a comprehensive type environment. This unified environment assigns types to both request-specific variables (viz., `principal`, `action`, `resource`, and `context`) and any local variables introduced during execution. Consequently, the typing rules for expressions is extended with the following rule:

$$(\textsc{TypeLocVar})\ \frac{\Gamma(y) = \tau}{\alpha; \Gamma \vdash y : \tau; \varnothing}$$

Then, the typing judgements for commands are as follows:

$$\alpha; \Gamma \vdash_s c : *; \alpha'$$

This states that, starting from an entity state with capabilities α, and with request and local variables conforming to the type environment Γ, the command c executes without type errors and terminates either in an entity state with capabilities α', or by producing some other kind of error (e.g., access to a non-existing entity).

$$(\textsc{EvalUpdEntity})\ \frac{\mu,\sigma\vdash e_0\Downarrow u \quad \mu,\sigma\vdash e_1\Downarrow v_1 \quad \mu,\sigma\vdash e_2\Downarrow v_2}{\sigma\vdash\langle\mu,\mathsf{updateEntity}(e_0,e_1,e_2)\rangle\Downarrow\mu[u\mapsto(v_1,v_2)]} \qquad (\textsc{EvalRemEntity})\ \frac{\mu,\sigma\vdash e\Downarrow u}{\sigma\vdash\langle\mu,\mathsf{removeEntity}(e)\rangle\Downarrow\mu|_{\mathrm{dom}(\mu)\setminus\{u\}}}$$

$$(\textsc{EvalAddParent})\ \frac{\mu,\sigma\vdash e_1\Downarrow u_1 \quad \mu,\sigma\vdash e_2\Downarrow u_2 \quad \mu(u_1)=(v,H)}{\sigma\vdash\langle\mu,\mathsf{addParent}(e_1,e_2)\rangle\Downarrow\mu[u_1\mapsto(v,H\cup\{u_2\})]} \qquad (\textsc{EvalRemParent})\ \frac{\mu,\sigma\vdash e_1\Downarrow u_1 \quad \mu,\sigma\vdash e_2\Downarrow u_2 \quad \mu(u_1)=(v,H)}{\sigma\vdash\langle\mu,\mathsf{removeParent}(e_1,e_2)\rangle\Downarrow\mu[u_1\mapsto(v,H\setminus\{u_2\})]}$$

$$(\textsc{EvalUpdAttribute})\ \frac{\mu,\sigma\vdash e_1\Downarrow u \quad \mu,\sigma\vdash e_2\Downarrow v \quad \mu(u)=(r,H)}{\sigma\vdash\langle\mu,\mathsf{updateAttribute}(e_1,f,e_2)\rangle\Downarrow\mu[u\mapsto(r[f\mapsto v],H)]} \qquad (\textsc{EvalSkip})\ \sigma\vdash\langle\mu,\mathsf{skip}\rangle\Downarrow\mu \qquad (\textsc{EvalBlock})\ \frac{\sigma\vdash\langle\mu,c\rangle\Downarrow\mu'}{\sigma\vdash\langle\mu,\{c\}\rangle\Downarrow\mu'}$$

$$(\textsc{EvalRemAttribute})\ \frac{\mu,\sigma\vdash e\Downarrow u \quad \mu(u)=(r,H)}{\sigma\vdash\langle\mu,\mathsf{removeAttribute}(e,f)\rangle\Downarrow\mu[u\mapsto(r|_{\mathrm{dom}(r)\setminus\{f\}},H)]} \qquad (\textsc{EvalSeq})\ \frac{\sigma\vdash\langle\mu,c_1\rangle\Downarrow\mu' \quad \sigma\vdash\langle\mu',c_2\rangle\Downarrow\mu''}{\sigma\vdash\langle\mu,c_1;c_2\rangle\Downarrow\mu''}$$

$$(\textsc{EvalIf1})\ \frac{\mu,\sigma\vdash e\Downarrow\mathsf{true} \quad \sigma\vdash\langle\mu,c_1\rangle\Downarrow\mu'}{\sigma\vdash\langle\mu,\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle\Downarrow\mu'} \qquad (\textsc{EvalIf2})\ \frac{\mu,\sigma\vdash e\Downarrow\mathsf{false} \quad \sigma\vdash\langle\mu,c_2\rangle\Downarrow\mu''}{\sigma\vdash\langle\mu,\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rangle\Downarrow\mu''} \qquad (\textsc{EvalFor1})\ \frac{\mu,\sigma\vdash e\Downarrow[\,]}{\sigma\vdash\langle\mu,\mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{do}\ c\rangle\Downarrow\mu}$$

$$(\textsc{EvalFor2})\ \frac{\mu,\sigma\vdash e\Downarrow[v,v_1,\ldots v_n] \quad \sigma[y\mapsto v]\vdash\langle\mu,c\rangle\Downarrow\mu' \quad \sigma\vdash\langle\mu',\mathsf{for}\ y\ \mathsf{in}\ [v_1,\ldots v_n]\ \mathsf{do}\ c\rangle\Downarrow\mu''}{\sigma\vdash\langle\mu,\mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{do}\ c\rangle\Downarrow\mu''}$$

Figure 2: Rules for Strobilus operational semantics.

$$\frac{\dfrac{\alpha;\Gamma\vdash E::s:E;\varepsilon \quad M(E)=(\{\ldots,?f:\tau,\ldots\},\_)}{\alpha;\Gamma\vdash E::s\ \mathsf{has}\ f:\mathsf{Bool};(E::s,\,f)} \qquad \dfrac{\alpha;\Gamma\vdash E::s:E;\varepsilon \quad M(E)=(\{\ldots,?f:\tau,\ldots\},\_)}{\alpha\cup(E::s,\,f);\Gamma\vdash e.f:\tau;(E::s,\,f)} \qquad \alpha;\Gamma\vdash alt:\tau;\varepsilon}{\alpha;\Gamma\vdash\mathsf{if}\ E::s\ \mathsf{has}\ f\ \mathsf{then}\ e.f\ \mathsf{else}\ alt:\tau;(E::s,\,f)\cap\varepsilon}$$

Figure 3: Example of type checking access to optional fields.

The typing rules for commands (Appendix B) differ from those for expressions in how capabilities are used: for expressions, the final capabilities ε hold only when the expression evaluates to `true`, whereas for commands, α' is valid after any execution.

The typing rules for basic commands guarantee that any modification of the entity store conforms to the schema M, and update the returned capability set accordingly.

The commands AddParent and RemoveParent modify the parent relation in the store; therefore, capabilities involving the binary operator in can be falsified. The corresponding typing rules return the set of capabilities $\mathrm{filt_p}(\alpha)$, defined as the subset of capabilities in α that do not contain the binary operator in, for example the capability $(\mathsf{if}\ (e_1\ \mathsf{in}\ e_2)\ \mathsf{then}\ e_3\ \mathsf{else}\ e_4, f)$. A similar consideration holds for the commands UpdateAttribute and RemoveAttribute; the corresponding rules use the set $\mathrm{filt_a}(f,\alpha)$, which is defined as the subset of capabilities in α not referring to the attribute f (e.g., the capability $(e.f, f')$). To deal with commands that modify entities, such as UpdateEntity and RemoveEntity, we use the filter $\mathrm{filt_t}(E,\alpha)$, which removes any capability whose expression contains references to entities of the same type E as the one being modified or removed.

The typing rules for if-then-else commands are similar to those for if-then-else expressions. The treatment of capabilities for the for loop is motivated by the fact that executing the loop body c may remove some capabilities, and it must be possible to run c without errors even in the reduced capability set. The remaining rules follow standard conventions and should be self-explaining.

4.3 Type safety

We prove that if a policy set with obligations *PO*, an initial entity store μ, and a request σ satisfy a schema (M,S), then the evaluation of *PO* is *guaranteed to be well-defined and error-free*. Furthermore, the resulting state μ' is guaranteed to preserve the invariants of the original schema (M,S).

To this end, some preliminary definitions are necessary.

Definition 4.1. *An environment* Γ *conforms to an action schema* S *if* $S(\Gamma(\mathsf{action})) = (H_p, H_r, \tau_x)$ *and* $\Gamma(\mathsf{principal}) = E_p :: s_p$, $\Gamma(\mathsf{resource}) = E_r :: s_r$, $\Gamma(\mathsf{context}) = \tau_x$, *with* $E_p \in H_p$ *and* $E_r \in H_r$.

A policy set with obligations P on allow c_1 on deny c_2 *conforms to a schema* (M,S) *if, for every type environment* Γ *conforming to the action schema* S, *the following type judgement can be derived using* (M,S) *as implicit parameters:*

$$\varnothing;\Gamma\vdash_s \mathsf{if}\ \mathit{eval}(P)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : *;\alpha_1$$

An entity store μ *conforms to an entity schema* M *if, for every entity* $E::s$ *in* $\mathit{dom}(\mu)$, $\mu(E::s)$ *has type* $M(E)$.

An authorization request σ *conforms to an action schema* S *if the authorization environment* Γ_σ *defining the types in* σ *conforms to* S.

The fact that a well-typed policy set with obligations will not generate type errors at runtime can be stated as follows.

Theorem 4.2. *Given a schema* (M,S), *a policy set with obligations* P on allow c_1 on deny c_2, *an entity store* μ, *and an authorization request* σ, *all conforming to* (M,S), *then either there exists a boolean value* v *and an entity store* μ', *conforming to* M, *such that*

$$\mathsf{result}(\mu,\sigma,P\ \mathsf{on\ allow}\ c_1\ \mathsf{on\ deny}\ c_2)\to(v,\mu').$$

or the evaluation fails due to referring to a missing entity.

The proof of this property relies on the analogous result for Cedar [7] (see Section 3.5, Validation Soundness), as well as a corresponding result for Strobilus commands, namely:

Lemma 4.3. *Given a schema* (M,S), *for any well-typed command* c, *entity store* μ, *and authorization request* σ, *conforming to* (M,S),

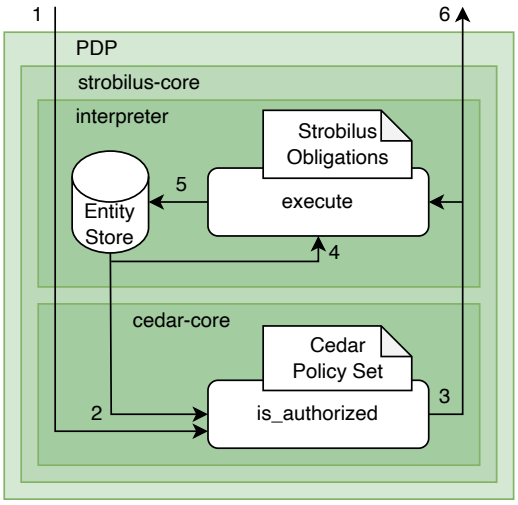


Figure 4: The architecture of a stateful PDP implemented by the Strobilus engine.

either there is an entity store μ' conforming to M such that $\sigma \vdash \langle \mu, c \rangle \Downarrow \mu'$, or the evaluation of c fails due to referring to a missing entity.

This can be established by induction on the structure of command c. These results have been formalized in the Lean proof assistant.

5 Strobilus: implementation

Strobilus is designed as a backward-compatible drop-in extension to Cedar, preserving full compatibility with existing policy sets. Its integration transforms the standard architecture in Figure 1 by creating a *stateful* PDP, responsible for internally managing the state of policy-related data and executing any associated effects upon reaching a final authorization decision.

The architecture of a Strobilus-based PDP is shown in Figure 4. When a request arrives from the PEP (1), Strobilus forwards it to the Cedar engine for evaluation, together with its internal entity store (2). After the Cedar engine computes an authorization decision (3), the request, decision, and the current entity store are passed to the Strobilus obligation engine (4). This engine then executes the obligations, which may result in changes to the entity store. The updated entity store is then persisted (5), and Cedar's original decision is returned to the PEP (6).

We have developed a prototype implementation of the interpreter and execution engine as a Rust crate named `strobilus-core`. This implementation is built on the `cedar-policy-core` crate (from the standard Cedar distribution), which provides the low-level functionality of Cedar. The crate `strobilus-core` can be used as a drop-in replacement of `cedar-policy-core`.

The Strobilus engine is initialized by parsing a Cedar policy set and obligations into their ASTs, coupled with an initial entity store that the engine maintains as persistent internal state. When a Cedar decision triggers an obligation, the interpreter traverses the corresponding AST branch and executes the mandated operations on the entity store, keeping system state in sync with authorization outcomes (see Algorithm 1).

Algorithm 1 Authorization procedure of Strobilus

```
1: procedure IS_AUTHORIZED(request)
2:     let entities = interpreter.entity_store();
3:     let decision = cedar_core.is_authorized(request, policy_set,
   entities);
4:     interpreter.execute(request, decision);
5:     return decision
6: end procedure
```

The AST for Strobilus obligations is generated using a parser built with LALRPOP. To ensure seamless integration, we extended the Cedar grammar with new syntactic categories specific to Strobilus commands. This unified grammar enables us to parse both the Cedar policy set and the Strobilus obligations, dispatching the respective ASTs to their corresponding Rust crates (Figure 4). This design ensures that the syntax and AST for expressions within Strobilus are identical to those in Cedar. Consequently, for evaluating these expressions during the execution of obligations, we directly re-use the optimized evaluation functions implemented in the `cedar-core` library. This tight integration significantly simplifies the architecture and guarantees highly efficient execution.

6 Strobilus: Performance Evaluation

In this section, we provide an evaluation of the performance of implementations using Strobilus against those that uses Cedar alone and state-related data is updated outside the authorization engine.

Our research question is: *What is the performance impact of using Strobilus compared to use only Cedar and keeping and updating the state in the application logic?*

For this evaluation, we consider *TinyTodo* [5], an official web application from the Cedar ecosystem which covers a representative set of non-trivial authorization patterns found in production systems. *TinyTodo* is a task management application that uses Cedar policies to manage access control. It enables individual *Users* to create, organize, and share their personal todo lists. Each *List* can contain multiple tasks that users can mark as completed once finished. The creator of a list (i.e., its *owner*) has the ability to share the list with other Users or Teams, granting them either *reader* or *editor* permissions. A reader can only view the list's contents, while an editor can add new tasks or marking them as done.

For the performance evaluation, we instrumented two versions of the *TinyTodo* example: the original Cedar-based implementation and a modified version that utilizes Strobilus for integrated authorization and state updates. Specifically, we modified the following CRUD (Create, Retrieve, Update, Delete) endpoints for the *List* entity to compare the performance of these stateful operations:

- `create_list`, which adds a new list for a user;
- `get_list`, which retrieves a specific list belonging to a user;
- `update_list`, which modifies the name of a user's list;
- `delete_list`, which removes a list from a user.

See Appendix A for the `create_list` listings for both versions.

Our evaluation procedure consists of 1000 experimental runs; each run executes a sequence of 400 calls: 100 calls to `create_list`, 100 calls to `get_list`, 100 calls to `update_list`, and 100 calls to `delete_list`. This entire run sequence is repeated 1000 times to

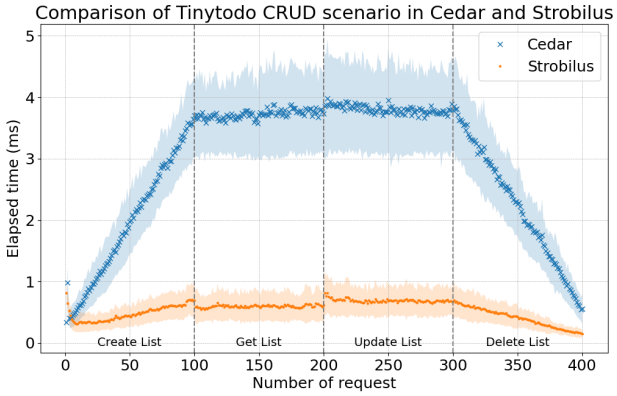


Figure 5: Time performance of Strobilus tests versus Cedar baseline implementations. Solid dots denote average values, while the shaded areas represent the standard deviation.

ensure statistical significance, with the execution time recorded for each of the 400 individual API calls. Our measurements capture the total time required for both request authorization and the associated system state update. To minimize external interference, all performance measurements are conducted using the kernel timer for CPU thread time (`CLOCK_THREAD_CPUTIME_ID`). All experiments have been run on an Intel Ultra 7 155U (14) at 4.80GHz and 16GB SDRAM DDR5 5600 MT/s, running Linux 6.14.0.

Figure 5 presents the time performance comparison between the Cedar and Strobilus versions of TinyTodo. The x-axis represents the sequential request number (1 through 400), while the y-axis displays the average execution time, and standard deviation, for each specific request number across all repetitions.

The analysis of the sequential phases reveals distinct trends. In the initial "Create List" phase, the execution time for the standard Cedar implementation quickly escalates. This is primarily due to the necessity of the application logic to convert its internal representation of the new *List* entity into Cedar's format for every authorization call. Strobilus, by contrast, integrates the data model directly within the authorization system, thereby eliminating this repeated conversion overhead. The "Retrieve List" and "Update List" phases show a clear performance disparity. While both operations only process a single element, the Strobilus version consistently achieves execution times averaging 0.7–0.9 ms, whereas the standard Cedar version averages 3.8–3.9 ms. This represents a performance improvement of approximately 4× to 5× for Strobilus. Finally, the "Delete List" phase exhibits a slight overall reduction in execution times across both systems, as the decreasing number of entities reduces the total workload. However, the performance advantage observed in the preceding phases persists: the Strobilus approach consistently outperforms Cedar due to its superior efficiency in managing and accessing integrated policy-related state.

Overall, we can observe that in the "Create List" scenario, Strobilus achieved a 67% average performance improvement over Cedar. In the remaining scenarios, the average improvement exceeded 80%, with the lowest value being 71%.

These results demonstrate that the performance overhead of using Strobilus compared to using a stateless Cedar PDP with state managed externally by the application (PEP/service logic) is generally negligible; in fact, as the entity data increases, Strobilus is substantially faster. This is because Cedar manages entities outside the authorization engine and hence the application must repeatedly convert the external data representation into Cedar's internal format for every authorization request. This conversion process introduces significant overhead. Furthermore, as the complexity or level of abstraction in the input data increases, the required conversion effort grows proportionally, leading to a compounded slowdown in the authorization phase.

We assess the statistical significance of the comparison between the Cedar and Strobilus versions using the Mann-Whitney U test [2]. Specifically, we calculated the U statistic and its corresponding p-value to determine whether the observed differences between the systems were statistically significant at a significance level of $\alpha = 0.05$. In fact, all scenarios presented a p-value below 0.05, indicating that the results were statistically significant.

7 Related work

The field of *policy languages* has seen the development of many DSL, aiming to provide a controlled and predictable method for evaluating access requests. Yet, most of these languages lack formal semantics, built-in support for state management and side effects.

A foundational language is *PONDER*, a declarative, object-oriented language for specifying security policies, mainly on distributed systems and enterprise networks [9]. Similarly to many of its contemporaries, it primarily focuses on defining permissions and prohibitions without a formal mechanism for side effects or state updates.

The *Policy Machine* (PM), developed at NIST [11], is a policy-neutral access control framework that provides a general-purpose, graph-based model for expressing and combining arbitrary access control policies. While PM is powerful for structuring multi-layered permission schemes, it does not provide a formal mechanism for executing stateful effects directly within the policy evaluation loop. State changes are realized through administrative operations that remain external to core policy evaluation. In contrast, Strobilus integrates state-mutating effects directly into the policy language itself, with full formal semantics and a type system.

The *Usage Control* model, UCON_{ABC}, introduced by Park and Sandhu [20], represents a foundational extension of traditional access control to include *usage* control. Unlike classical access control, UCON_{ABC} supports three decision factors—authorizations (A), obligations (B), and conditions (C)—and crucially introduces *attribute mutability*, allowing subject and object attributes to be updated before, during, or after access. Strobilus can be seen as realizing this attribute-mutability vision within a concrete, formally grounded, and deployable policy-as-code framework. However, UCON_{ABC} is a conceptual model rather than a deployed policy language: it does not provide a formal operational semantics, a type system, or a concrete DSL, and so cannot be directly compared to the implementation-ready approach of Cedar+Strobilus.

One of the most widespread languages for defining policies in a distributed environment is *eXtensible Access Control Markup Language* (XACML) [1]. XACML does include a concept of *obligations*, which are actions that are required to be performed by the PEP when a decision is made. However, the standard does not provide

a clear definition of obligations, leaving their actual meaning to be defined outside of the language itself. Some implementations of XACML, such as WSO2 Balana [22], support obligation definition by means of "obligation templates", analogous to a macro language: policy evaluation for a given request includes evaluating these templates, which generates the concrete obligation passed to the PEP for execution. However, this only addresses the generation of the obligation, not its execution. The actual execution semantics of these obligations remain undefined and open-ended, relying on the correct, external implementation within the PEP.

Several extensions of XACML have been proposed to address these limitations. Martinelli et al. [13] introduce *U-XACML*, which extends XACML with history-based usage control policies and an enforcement mechanism for tracking past access history. For distributed environments, Demchenko et al. [8] and El Kateb et al. [10] propose extensions to handle obligation management across distributed systems. More recently, Martinelli et al. [14] address obligation management in usage control systems more systematically. Despite this progress, all these approaches share the same fundamental limitation: the execution semantics of obligations are still delegated to external components, preventing end-to-end formal verification. Strobilus addresses this gap by making effect execution an integral, formally specified part of the policy engine itself.

Inspired by XACML, *Formal Access Control Policy Language* (FACPL) [12, 15] stands as a notable exception, offering a rigorous and formal framework for specifying and enforcing ABAC policies. Its semantics explicitly address aspects such as the management of missing attributes, errors, and the combination of obligations using various strategies. However, the actual execution semantics of obligations are not defined within the language itself, but are instead assumed as given by an external "oracle". This delegation prevents end-to-end verifiability for state-changing policies.

The *Open Policy Agent (OPA)* and its policy language, *Rego*, are very popular for implementing "policy-as-code" in cloud-native environments [17]. Rego's declarative, Datalog-inspired design is particularly effective for making assertions on structured data. However, it lacks built-in mechanisms for defining and managing obligations or state changes during a policy decision. To address this limitation, the *OPA Wrapper State Manager* (OWSM) has been introduced in [4]. OWSM functions as a "wrapper" that maintains a separate state store for policy-specific information, which the Rego engine can consult during evaluation. This allows for the definition of stateful policies in Rego without altering its core syntax or evaluation engine. Still, neither Rego nor OWSM provide a formal, clearly defined semantics for their evaluation and effects.

OpenFGA [16] is an open-source authorization system inspired by Google's Zanzibar [19]. It uses a declarative modeling language to define relationships between users, objects, and permissions). While powerful for relationship-based access control, its design is focused on a declarative model of "who can do what," and it does not provide a formal mechanism for state-changing operations or obligations as part of its core policy language.

Swamy et al. [21] introduce FINE, aiming to address state-dependent policy verification. FINE uses language-level *refinement types* to enforce stateful authorization and information flow, whereas our approach embeds state transitions directly into the policies-as-code engine, keeping policy logic isolated from application code.

8 Conclusion and Future Work

In this paper we introduced *Strobilus*, a DSL designed for defining state modifications on policy-related data by means of *obligations*, seamlessly alongside the authorization rules of Amazon's policy language Cedar. Crucially, like Cedar, Strobilus comes with a formal semantics, a guarantee of termination, a robust typing system, and a formal proof of type safety (formalized in Lean).

We developed a prototype implementation of Strobilus in Rust that is fully backward-compatible with Cedar, allowing it to function as a drop-in extension. Our experiments, using a non-trivial example from Cedar's showcase, demonstrate that the Strobilus-based implementation offers substantially greater efficiency than the original stateless Cedar version, primarily by eliminating the high overhead of external entity store reconstruction.

Ultimately, Strobilus completes the vision of "policies-as-code" by providing a comprehensive, secure, and verifiable solution for both stateless and stateful authorization.

In principle, the methodology employed in this work is generalizable to other policy languages. However, we emphasize that implementing this approach on a language that lacks a formally defined syntax and semantics would hinder the future development of rigorous verification tools grounded in formal methods.

Future Work. The formal semantics of Strobilus paves the way for the development of specialized tools for static analysis and formal verification; this would provide developers with formal guarantees about the correctness and safety of their stateful policies.

Additionally, we intend to explore the challenges of distributed state management. While Strobilus provides a solid solution for a single-source-of-truth entity store, extending the language to handle concurrent updates and consistency across multiple, distributed policy enforcement points is a crucial next step.

Multi-turn agentic AI systems, where AI agents interact with external services over a sequence of conversation steps, are a compelling application domain for Strobilus. In such settings, the authorization of each agent action (e.g., querying a database, sending an email, or invoking an API) may depend on the history of prior turns: which resources have been accessed, how many times a service has been called, or whether a previous step was successfully completed. Strobilus is well-suited for this domain: mutable attributes can model evolving agent context, obligations can atomically update state after each permitted action, and policies can enforce history-aware constraints without burdening the agent framework with policy-related state management. However, this application also exposes key limitations that motivate future work. First, concurrent multi-agent settings introduce the risk of *race conditions* and inconsistent state reads. Second, long-running conversations amplify the state-space, making formal verification and audit trail generation more challenging. Third, the non-determinism inherent in LLM-based agents complicates the specification of invariants over mutable state. Addressing these challenges is a promising direction for extending Strobilus to fully support agentic AI authorization.

Acknowledgments

This work was supported by the M4C2 I1.3 "SERICS" (PE00000014), CUP D33C22001300002, under the National Recovery and Resilience Plan (NRRP) funded by the European Union - NextGenerationEU.

References

[1] Anne Anderson, Anthony Nadalin, B Parducci, D Engovatov, H Lockhart, M Kudo, P Humenn, S Godik, S Anderson, S Crocker, et al. 2003. Extensible access control markup language (XACML) version 1.0.

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486

[3] David Aspinall. 1994. Subtyping with Singleton Types. In *Computer Science Logic, 8th International Workshop, CSL'94 (Lecture Notes in Computer Science, Vol. 933)*. Springer, 1–15. doi:10.1007/BFB0022243

[4] Massimiliano Baldo, Fabio Ionut Ion, Marino Miculan, Matteo Paier, and Vincenzo Riccio. 2025. OWSM: Empowering Rego for Stateful Access Control. In *Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2025) (CEUR Workshop Proceedings, Vol. 3962)*, Gabriele Costa, Rebecca Montanari, Michele Carminati, and Giada Sciarretta (Eds.). https://ceur-ws.org/Vol-3962/paper49.pdf

[5] Cedar-policy. 2024. TinyTodo. https://github.com/cedar-policy/cedar-examples/tree/release/4.5.x/tinytodo

[6] Karl Crary, David Walker, and Greg Morrisett. 1999. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 262–275.

[7] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John H. Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 670–697. doi:10.1145/3649835

[8] Yuri Demchenko, Oscar Koeroo, Cees de Laat, and Henrik Sagehaug. 2008. Extending XACML Authorisation Model to Support Policy Obligations Handling in Distributed Application. In *Proceedings of the 6th International Workshop on Middleware for Grid Computing*.

[9] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. 2001. A policy deployment model for the Ponder language. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings*. 529–543. doi:10.1109/INM.2001.918064

[10] Donia El Kateb, Yehia ElRakaiby, Takoua Mouelhi, Ishtiaq Rubab, and Yves Le Traon. 2014. Towards a Full Support of Obligations in XACML. In *International Conference on Risks and Security of Internet and Systems (CRiSIS)*. 213–221.

[11] David Ferraiolo and Serban Gavrila. 2019. *Policy Machine: Features, Architecture, and Specification*. Technical Report NIST SP 500-302. National Institute of Standards and Technology.

[12] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. 2017. A rigorous framework for specification, analysis and enforcement of access control policies. *IEEE Transactions on Software Engineering* 45, 1 (2017), 2–33.

[13] Fabio Martinelli, Ilaria Matteucci, Paolo Mori, and Andrea Saracino. 2016. Enforcement of U-XACML History-Based Usage Control Policy. In *Security and Trust Management (STM 2016) (Lecture Notes in Computer Science, Vol. 9871)*. Springer, 49–64.

[14] Fabio Martinelli, Paolo Mori, Andrea Saracino, and Francesco Di Cerbo. 2019. Obligation Management in Usage Control Systems. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 356–364.

[15] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. 2012. Formalisation and implementation of the XACML access control mechanism. In *International Symposium on Engineering Secure Software and Systems*. Springer, 60–74.

[16] OpenFGA. 2025. Relationship-based access control made fast, scalable, and easy to use. Available at https://openfga.dev/.

[17] OpenPolicyAgent. 2025. Rego documentation. Available at https://www.openpolicyagent.org/docs/latest/policy-language/.

[18] Samodha Pallewatta and Muhammad Ali Babar. 2024. Towards secure management of edge-cloud IoT microservices using policy as code. In *European Conference on Software Architecture*. Springer, 270–287.

[19] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (ATC '19)*.

[20] Jaehong Park and Ravi S. Sandhu. 2004. The UCON$_{ABC}$ Usage Control Model. *ACM Transactions on Information and System Security* 7, 1 (2004), 128–174.

[21] Nikhil Swamy, Juan Chen, and Ravi Chugh. 2010. Enforcing stateful authorization and information flow policies in FINE. In *European Symposium on Programming*. Springer, 529–549.

[22] WSO2. 2015. Balana Implementation. https://github.com/wso2/balana

A Example of usage of Strobilus

```rust
fn create_list(&mut self, r: CreateList) -> Result<AppResponse> {
  let timer = ThreadTime::now();
  let euid = self
    .entities
    .fresh_euid::<ListUid>(TYPE_LIST.clone())
    .unwrap();
  self.is_authorized(
    &r.uid,
    &*ACTION_CREATE_LIST,
    &*APPLICATION_TINY_TODO,
  )?;
  let l = List::new(&mut self.entities, euid.clone(), r.uid, r.name);
  self.entities.insert_list(l);
  save_measure_time(timer.elapsed(), "./results/tinytodo-cedar/create_list.csv");
  Ok(AppResponse::euid(euid))
}
```

Listing 10: `create_list` function from TinyTodo, instrumented for time measuring.

```rust
fn create_list(&mut self, r: CreateList) -> Result<AppResponse> {
  let timer = ThreadTime::now();

  let euid = self
    .entities
    .fresh_euid::<ListUid>(TYPE_LIST.clone())
    .unwrap();

  self.is_authorized(
    &r.uid,
    &*ACTION_CREATE_LIST,
    &*APPLICATION_TINY_TODO,
    Some(HashMap::from([
      ("uid", RestrictedExpression::new_entity_uid(EntityUid::from(euid.clone()).into())),
      ("owner", RestrictedExpression::new_entity_uid(EntityUid::from(r.uid.clone()).into())),
      ("tasks", RestrictedExpression::new_set(std::iter::empty())),
      ("name", RestrictedExpression::new_string(r.name.clone())),
    ])),
  )?;
  save_measure_time(timer.elapsed(), "./results/tinytodo-strobilus/create_list.csv");
  Ok(AppResponse::euid(euid))
}
```

Listing 11: `create_list` function from TinyTodo, ported to Strobilus, instrumented for time measuring.

```
on allow {
  if (action == Action::"CreateList" || action == Action::"UpdateList") then {
    updateEntity(context.uid, {
      "owner": context.owner,
      "name": context.name,
      "tasks": context.tasks
    }, [Application::"TinyTodo"], {});
  } else {
    if (action == Action::"DeleteList") then {
      removeEntity(resource);
    }
  }
}
```

Listing 12: Strobilus obligation for TinyTodo.

B Typing rules for Strobilus commands

$$(\textsc{TypeAddParent})\ \frac{\alpha;\Gamma \vdash e_1 : E_1;\varepsilon_1 \quad \alpha;\Gamma \vdash e_2 : E_2;\varepsilon_2 \quad M(E_1) = (\_, H_1) \quad E_2 \in H_1}{\alpha;\Gamma \vdash_s \mathsf{addParent}(e_1, e_2) : *; \mathrm{filt}_{\mathrm{p}}(\alpha)}$$

$$(\textsc{TypeRemoveParent})\ \frac{\alpha;\Gamma \vdash e_1 : E_1;\varepsilon_1 \quad \alpha;\Gamma \vdash e_2 : E_2;\varepsilon_2 \quad M(E_1) = (\_, H_1) \quad E_2 \in H_1}{\alpha;\Gamma \vdash_s \mathsf{removeParent}(e_1, e_2) : *; \mathrm{filt}_{\mathrm{p}}(\alpha)}$$

$$(\textsc{TypeUpdateAttribute})\ \frac{\alpha;\Gamma \vdash e_1 : E;\varepsilon_1 \quad M(E) = (\{\ldots, \omega f : \tau, \ldots\}, \_) \quad \alpha;\Gamma \vdash e_2 : \tau;\varepsilon_3}{\alpha;\Gamma \vdash_s \mathsf{updateAttribute}(e_1, f, e_2) : *; \mathrm{filt}_{\mathrm{a}}(f, \alpha \cup \{(e_1, f)\})}$$

$$(\textsc{TypeRemoveAttribute})\ \frac{\alpha;\Gamma \vdash e : E;\varepsilon_1 \quad M(E) = (\{\ldots, ?f : \tau, \ldots\}, \_)}{\alpha;\Gamma \vdash_s \mathsf{removeAttribute}(e, f) : *; \mathrm{filt}_{\mathrm{a}}(f, \alpha \backslash \{(e', f) \mid e' \in \mathit{Expr}\})}$$

$$(\textsc{TypeUpdateEntity})\ \frac{\alpha;\Gamma \vdash e_1 : E;\varepsilon_1 \quad \alpha;\Gamma \vdash e_2 : A;\varepsilon_2 \quad M(E) = (A, H) \quad \{E_1, \ldots, E_n\} \subseteq H}{\alpha;\Gamma \vdash_s \mathsf{updateEntity}(e_1, e_2, [E_1 :: s_1, \ldots, E_n :: s_n]) : *; \mathrm{filt}_{\mathrm{t}}(E, \alpha)}$$

$$(\textsc{TypeRemoveEntity})\ \frac{\alpha;\Gamma \vdash e : E;\varepsilon}{\alpha;\Gamma \vdash_s \mathsf{removeEntity}(e) : *; \mathrm{filt}_{\mathrm{t}}(E, \alpha)}$$

$$(\textsc{TypeSequence})\ \frac{\alpha;\Gamma \vdash_s c_1 : *; \alpha_1 \quad \alpha_1;\Gamma \vdash_s c_2 : *; \alpha_2}{\alpha;\Gamma \vdash_s c_1; c_2 : *; \alpha_2} \qquad (\textsc{TypeBlock})\ \frac{\alpha;\Gamma \vdash_s c : *; \alpha_1}{\alpha;\Gamma \vdash_s \{c\} : *; \alpha_1}$$

$$(\textsc{TypeIfC1})\ \frac{\alpha;\Gamma \vdash e : \mathsf{True};\varepsilon \quad \alpha \cup \varepsilon;\Gamma \vdash_s c_1 : *; \alpha_1}{\alpha;\Gamma \vdash_s \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : *; \alpha_1} \qquad (\textsc{TypeIfC2})\ \frac{\alpha;\Gamma \vdash e : \mathsf{False};\varepsilon \quad \alpha;\Gamma \vdash_s c_2 : *; \alpha_1}{\alpha;\Gamma \vdash_s \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : *; \alpha_1}$$

$$(\textsc{TypeIfC3})\ \frac{\alpha;\Gamma \vdash e : \mathsf{Bool};\varepsilon \quad \alpha \cup \varepsilon;\Gamma \vdash_s c_1 : *; \alpha_1 \quad \alpha;\Gamma \vdash_s c_2 : *; \alpha_2}{\alpha;\Gamma \vdash_s \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : *; \alpha_1 \cap \alpha_2}$$

$$(\textsc{TypeFor})\ \frac{\alpha;\Gamma \vdash e : \mathit{Set}\ \tau;\varepsilon \quad \alpha_1 = \alpha \cap \alpha_2 \quad \alpha_1;\Gamma \cup \{(y, \tau)\} \vdash_s c : *; \alpha_2}{\alpha;\Gamma \vdash_s \mathsf{for}\ y\ \mathsf{in}\ e\ \mathsf{do}\ c : *; \alpha_1}$$