Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization

JOSEPH W. CUTLER[*], University of Pennsylvania, USA
CRAIG DISSELKOEN, Amazon Web Services, USA
AARON ELINE, Amazon Web Services, USA
SHAOBO HE, Amazon Web Services, USA
KYLE HEADLEY[*], Unaffiliated, USA
MICHAEL HICKS, Amazon Web Services, USA
KESHA HIETALA, Amazon Web Services, USA
ELEFTHERIOS IOANNIDIS[*], University of Pennsylvania, USA
JOHN KASTNER, Amazon Web Services, USA
ANWAR MAMAT[*], University of Maryland, USA
DARIN MCADAMS, Amazon Web Services, USA
MATT MCCUTCHEN[*], Unaffiliated, USA
NEHA RUNGTA, Amazon Web Services, USA
EMINA TORLAK, Amazon Web Services, USA
ANDREW M. WELLS, Amazon Web Services, USA

Cedar is a new authorization policy language designed to be ergonomic, fast, safe, and analyzable. Rather than embed authorization logic in an application's code, developers can write that logic as Cedar policies and delegate access decisions to Cedar's evaluation engine. Cedar's simple and intuitive syntax supports common authorization use-cases with readable policies, naturally leveraging concepts from role-based, attribute-based, and relation-based access control models. Cedar's policy structure enables access requests to be decided quickly. Cedar's policy validator leverages optional typing to help policy writers avoid mistakes, but not get in their way. Cedar's design has been finely balanced to allow for a sound and complete logical encoding, which enables precise policy analysis, e.g., to ensure that when refactoring a set of policies, the authorized permissions do not change. We have modeled Cedar in the Lean programming language, and used Lean's proof assistant to prove important properties of Cedar's design. We have implemented Cedar in Rust, and released it open-source. Comparing Cedar to two open-source languages, OpenFGA and Rego, we find (subjectively) that Cedar has equally or more readable policies, but (objectively) performs far better.

[*]Work carried out while at Amazon Web Services

Authors' addresses: Joseph W. Cutler, University of Pennsylvania, USA, jwc@seas.upenn.edu; Craig Disselkoen, Amazon Web Services, USA, cdiss@amazon.com; Aaron Eline, Amazon Web Services, USA, aeline@amazon.com; Shaobo He, Amazon Web Services, USA, shaobohe@amazon.com; Kyle Headley, Unaffiliated, USA, kylenheadley@gmail.com; Michael Hicks, Amazon Web Services, USA, mwhicks@amazon.com; Kesha Hietala, Amazon Web Services, USA, khieta@amazon.com; Eleftherios Ioannidis, University of Pennsylvania, USA, elefthei@seas.upenn.edu; John Kastner, Amazon Web Services, USA, jkastner@amazon.com; Anwar Mamat, University of Maryland, USA, anwar@umd.edu; Darin McAdams, Amazon Web Services, USA, darinm@amazon.com; Matt McCutchen, Unaffiliated, USA, matt@mattmccutchen.net; Neha Rungta, Amazon Web Services, USA, rungta@amazon.com; Emina Torlak, Amazon Web Services, USA, torlaket@amazon.com; Andrew M. Wells, Amazon Web Services, USA, anmwells@amazon.com.

ACM 2475-1421/2024/4-ART118
https://doi.org/10.1145/3649835

CCS Concepts: • **Security and privacy** → *Formal methods and theory of security*; **Authorization**; • **Theory of computation** → *Semantics and reasoning*.

Additional Key Words and Phrases: Authorization, Formal models, Policies as code

ACM Reference Format:
Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 118 (April 2024), 28 pages. https://doi.org/10.1145/3649835

1 INTRODUCTION

Authorization is the problem of deciding who has access to what in a multi-user system. Every cloud-based application has to solve this problem, from photo sharing to online banking to health care. A common solution is to embed access control logic in the application code, as shown in the `get_list` method for TinyTodo (Figure 1), a hypothetical application for managing todo lists.

```
1  def get_list(request):
2      if not db.query(request.user).admin:
3          if db.query(request.listId).owner != request.user:
4              if not request.user in db.query(request.listId).readers:
5                  if not request.user in db.query(request.listId).editors:
6                      return 'AccessDenied'
7      list = db.query(request.listId)
8      return { 'id': list.id, 'owner': list.owner, ... }
```

Fig. 1. Embedded access control logic in the `get_list` method for TinyTodo

This approach has three major drawbacks. First, it is *hard to write* correct permissions. The logic in `get_list` denies access if the requester is neither an admin, nor the list's owner, nor a member of its readers or writers groups. It would be easy to mistakenly drop a `not` or improperly nest the conditions; such mistakes account for four of the top 25 security weaknesses in 2023 [34]. Second, embedded permissions are *hard to understand and audit.* An auditor needs to read the application's code to check access invariants such as *a list's editors can perform all the same actions as the list's readers.* Finally, embedded permissions are *hard to maintain.* Changing authorization logic means changing the application code, so policy versioning is tantamount to code versioning. Adding crosscutting permissions (e.g., supporting a new user role) requires updates to code in multiple places, which offers more chances for mistakes.

Policies as code. A better alternative is to externalize access control rules into policies written in a domain-specific *authorization language*, and delegate decisions to the language's *authorization engine*. This approach is called *policies as code* [54]. With policies as code, lines 2–6 of `get_list` become a single, unchanging call to the authorization engine, e.g., `if not client.is_authorized(...): return 'AccessDenied'`. The authorization engine makes a decision by consulting the application's policies, which are expressed separately from the application code, in a dedicated DSL. This makes access control rules easier to understand, audit, change, version, and share between applications.

But building a high-quality authorization language is a significant challenge that requires balancing four competing goals. In particular, the language should be **expressive**, so that application developers can grant access based on user and resource attributes, group membership, and session context [21, 26, 43]; **performant**, so that requests are decided quickly; **safe**, so that policy authoring mistakes are caught or avoided; and **analyzable**, so that policy analysis tools can reason precisely about important access invariants.

The first goal is in tension with the other three. A highly expressive language may have constructs that are slow to execute, hard to prove safe, and impossible to analyze precisely. A fast, safe, and analyzable language, in contrast, may be too weak to express common authorization use-cases. To our knowledge (Section 6), no existing language finds the ideal balance of these four goals.

Cedar: a novel authorization language. This paper presents Cedar, a new authorization language that is simultaneously expressive, performant, safe, and analyzable. Cedar is used at scale in products and services from Amazon Web Services, and is open source.

Cedar's vocabulary can naturally **express** who has access to what based on users' roles and attributes of their environment. Most Cedar operators take constant time, and looping constructs are linear, ensuring **fast** policy evaluation times. While deciding a request considers all active policies, Cedar is able to quickly *slice* the full policy set to evaluate only those relevant to the request.

Cedar's design creates a **safe** foundation for authoring policies. Policies have no side effects, and decisions are indifferent to the order that policies are considered. Cedar policies never authorize an action by default—it must be explicitly permitted. Cedar provides a *policy validator*, based on a novel type system leveraging *singleton types* [1] and *static capabilities* [12], that ensures policy evaluation will never error on a type mismatch, or a bogus attribute or role name.

Cedar's design ensures that its policies are efficiently **analyzable** by reduction to SMT, so that access invariants can be proved automatically. We define a novel *symbolic compiler* that translates Cedar policies to SMT, producing a decidable, sound, and complete encoding of their semantics.

Implementation and Evaluation. We have formalized Cedar in Lean [35] and used Lean's proof assistant to prove the properties mentioned above. We have implemented Cedar in Rust, and used extensive differential random testing to ensure that the implementation matches the model.

We conduct three sets of experiments to evaluate Cedar's expressiveness, performance, and analyzability. First, we use Cedar to encode the authorization models for two example applications previously modeled in OpenFGA [40], a recently developed language for specifying relationship-based access control (ReBAC) [43] policies. We can express both of these OpenFGA models with validated Cedar policies. Next, we compare the performance of Cedar, OpenFGA, and Rego [36]—a popular authorization language based on Datalog—on three example applications: the two OpenFGA examples and an additional example described in Section 2. Comparing the performance of these three models on randomly generated inputs, we find that the Cedar authorizer is 28.7×-35.2× faster than OpenFGA and 42.8×-80.8× faster than Rego. We also evaluate the effectiveness of Cedar's policy slicing, and find that authorization is 10.0×-18.0× faster on average, when using a Cedar templates-based encoding. Finally, we find that Cedar's type system is powerful enough to validate all policies in our example models, and Cedar's symbolic compiler reduces analysis questions about these policies into SMT-LIB formulas that take, on average, 75.1 ms to encode and solve.

In summary, this paper makes the following contributions:

- Design, implementation, and evaluation of Cedar, a new authorization language that balances expressiveness, safety, performance, and analyzability.
- A verified validator for Cedar, designed to accept safe policies that are translatable to SMT.
- A verified symbolic compiler for reducing Cedar to a decidable fragment of SMT.

Cedar's code, documentation, and example apps can be found at https://github.com/cedar-policy/.

2 OVERVIEW

We overview Cedar using TinyTodo, the application introduced in the prior section. We illustrate the structure and semantics of Cedar policies and its data model, how the Cedar validator typechecks Cedar policies to prevent run-time errors, and how the Cedar symbolic compiler allows us to reason about the meaning of Cedar policies.

```
// Policy 0: Any User can create a list
// and see what lists they own.
permit(
    principal,
    action in [Action::"CreateList",
               Action::"GetOwnedLists"],
    resource == Application::"TinyTodo");

// Policy 2: Admins can perform any action.
permit(
    principal in Team::"admin",
    action,
    resource in Application::"TinyTodo");

// Policy 4: Interns can't create task lists.
forbid(
    principal in Team::"interns",
    action == Action::"CreateList",
    resource == Application::"TinyTodo");
```

```
// Policy 1: Any User can perform any action
// on a List they own.
permit(principal, action, resource)
when {
    resource is List &&
    resource.owner == principal
};

// Policy 3: A User can see a List
// if they are either a reader or editor.
permit(
    principal,
    action == Action::"GetList",
    resource)
when {
    principal in resource.readers ||
    principal in resource.editors
};
```

Fig. 2. Cedar policies for TinyTodo

2.1 Basic TinyTodo policies: Creating and managing lists

TinyTodo allows individuals, called `Users`, and groups, called `Teams`, to organize, track, and share their todo `Lists`. We specify and enforce TinyTodo's access rules using Cedar policies.

Consider the first two of TinyTodo's policies shown in Figure 2. Policy 0 states that any principal (a TinyTodo `User`) can create a list or get a listing of their previously created lists. The resource `Application::"TinyTodo"` represents the TinyTodo application itself, which is a container for any created list. Policy 1 states that any principal can perform any action on a resource that is a `List` and whose `owner` attribute matches the requesting principal. TinyTodo always sets a `List`'s `owner` at the time it is created to the `User` that created it.

Suppose there is a TinyTodo user `andrew` who attempts to create a new todo list called `"Create demo"`, add two tasks to that list, and then complete one of the tasks. Creating the list is authorized by Policy 0: Any user is allowed to create a `List`. The other three actions are authorized by Policy 1: since `andrew` is the owner of the `List`, he may carry out any action on it.

2.2 Cedar data model: Hierarchical entities

Principals, resources, and actions are represented by Cedar *entities*. Entities are collected in an *entity store* and referenced by a unique identifier consisting of two parts: the *entity type* and *entity ID*. Each entity is associated with zero or more attributes mapped to values, and zero or more parent entities. The parent relation on entities forms a directed acyclic graph (DAG), called the *entity hierarchy*.

Figure 3 illustrates the TinyTodo entities after carrying out the actions described above. On the left is the entity for the list `andrew` created, which has entity reference `List::"0"`. We can see that the entity has five attributes, `name`, `owner`, `readers`, `editors`, and `tasks`. All entities of type `List` have these attributes. On the right of the figure we see four `User`-typed entities (along the bottom) and five `Team`-typed entities (along the top), which are arranged in a hierarchy; each arrow points to an entity's immediate parent. TinyTodo set these attributes, and created `Team::"1"` and `Team::"2"`, when it created the list, and updated `tasks` as tasks were added and completed.

An application might store its data in a SQL database or a key-value store in a format that best suits its purposes. When it goes to authorize a user request, it provides the Cedar authorizer

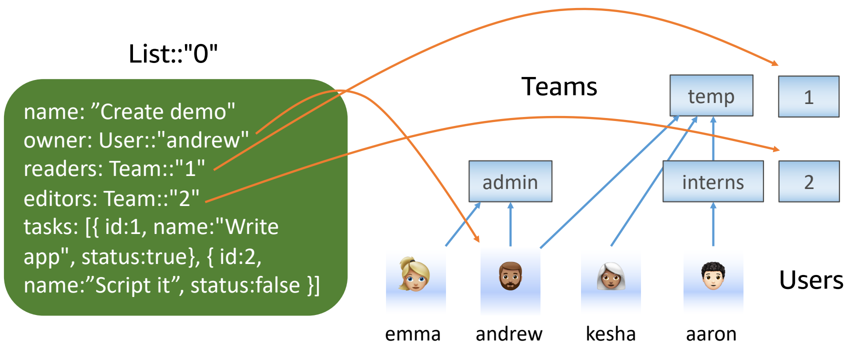


Fig. 3. Example Cedar entity hierarchy for TinyTodo

access to the necessary parts of that data, mapped to Cedar entities. It could do so by mapping its native-stored data to Cedar entities at the time of the request, or during the application's operation.

2.3 Cedar policy structure: Supporting RBAC, ABAC, ReBAC

In general, Cedar policies have three components: the *effect*, the *scope*, and the *conditions*. The effect is either `permit` or `forbid`, indicating whether the policy is authorizing access or taking it away. The scope comes after the effect, constraining the principal, action, and resource components of requests that the policy applies to. The conditions are optional expressions that come last, adding further constraints, oftentimes based on attributes of request elements. Policies access request elements using the variables `principal`, `action`, `resource`, and `context`. The `context` record stores application-specific data about the request, such as current time or source IP address.

Cedar policies support role-based, attribute-based, and relation-based access control (RBAC, ABAC, and ReBAC, respectively). RBAC is possible because of Cedar's support for a hierarchy of entities. For example, Policy 2 in Figure 2 is an RBAC-style policy, which states that any principal who is a member of the `admin` `Team` (such as `User::"emma"`) can perform any action on a TinyTodo resource.

ABAC-style policies use policy conditions to reference *attributes*. For example, Policy 1 mentions `resource.owner`, where `owner` is a attribute of `resource`. As shown in Figure 3, attributes can be primitive-typed values (like a number or string), collections, records, or entity references.

ReBAC-style policies use attributes that reference entity groups to express *relations* between two entities. For example, Policy 3 is a ReBAC-style policy, which states that any principal can read the contents of a task list (`Action::"GetList"`) if they are in either the list's `readers` or `editors` groups. Here, `principal in resource.readers` and `principal in resource.editors` are expressions that can be viewed as querying whether `principal` is in the *readers* and *editors* relations with `resource`.

For our running example, suppose `User::"andrew"` shares `List::"0"` with the team `interns` as a reader (which he is allowed to do because of Policy 1). As a result, TinyTodo will update the hierarchy in Figure 3 to add a parent edge from `Team::"interns"` to `Team::"1"`. Now if `User::"aaron"` attempts to read the contents of `List::"0"` he will be allowed to do so according to Policy 3: its condition `principal in resource.readers` is true since the `readers` attribute of `List::"0"` corresponds to `Team::"1"`, which is an ancestor of `User::"aaron"` in the entity hierarchy.

```
entity Application;
entity Team, User in [Team, Application];
entity List in [Application] { readers: Team, editors: Team, owner: User };
action CreateList appliesTo { principal: [User], resource: [Application] };
action GetList appliesTo { principal: [User], resource: [List] };
```

Fig. 4. TinyTodo schema

2.4 Safety

Cedar has a design goal of creating a *safe* foundation for writing policies. One aspect of this goal is that Cedar's authorizer *denies by default*: If no `permit` policy exists that authorizes a request, the request will be denied.

Another aspect is that `forbid` *policies always override* `permit` *policies* when making the final decision. For example, Policy 4 expresses that no intern is allowed to create a new task list (`Action::"CreateList"`). With this policy, if `aaron` tries to create a list, his request will be approved by Policy 0, but denied by Policy 4, with the final decision that the request is denied. This behavior allows `forbid` policies to act like guardrails that enforce universal non-access rules.

The third aspect of safety is that Cedar's authorizer is *deterministic*: It is guaranteed to terminate and always produce the same authorization decision for a given request, hierarchy, and set of policies. Because Cedar policies are free of side effects and general loops, policy evaluation order doesn't matter. Because Cedar's algorithm for selecting the policies *relevant* to a request is *sound*, the authorizer will produce the same outcome as if had it considered all available policies.

2.5 Validating policies

A final aspect of safety is ensuring that Cedar policy evaluation will not result in an error, e.g., if the policy attempts to access a non-existent attribute. Policy writers can use the Cedar *validator* to ensure that policies are (mostly) error-free. To do so, they provide a *schema* that lists the names and the type structure of an application's entities, as well as the application actions and the allowed shapes of requests that contain them.

A schema for TinyTodo is shown in Figure 4 (some actions and entity attributes are elided). The schema declares four entity types: `Application`, `User`, `Team`, and `List`. The `Application` entity type is the simplest possible: it has no associated attributes and has no parents in the entity hierarchy. The combined `User` and `Team` entity type declaration indicates that both can have `Team` entities as parents (as we see in Figure 3) as well as `Application` entities. A `List` entity can have `Application`-entity parents and must have the attributes `owner`, `readers`, and `writers`. The schema also declares two actions, `CreateList` and `GetList`, and specifies that they *apply to* specific principal and resource entity types—any request involving these actions must include `principal` and `resource` entities of these types. All of the policies in Figure 2 are valid with respect to this schema.

Policy validation works by typechecking the policy specialized to each of the allowed actions in the *request environments* (maps from variables like `principal` and `resource` to types) induced by those actions' schemas. A policy is valid if it typechecks in every environment. The type system uses a novel combination of *static capabilities* [12] and boolean *singleton types* [1]. Capabilities ensure that a policy always checks the presence of an optional attribute before accessing it. Singleton types *True* and *False* are ascribed to expressions sure to evaluate to `true` and `false`, respectively. The type system uses them to type expressions like **`false`** `&& (1 == "hello")`: since the first conjunct has type *False* the whole expression will—the erroring second conjunct will be short-circuited. Singleton types are also used to warn when a policy will evaluate to **`false`** (or **`true`**) in *every* environment, making it effectively useless (or over-influential).

Typechecking in each relevant environment, rather than a single generic environment, adds cost but avoids many false alarms. There are typically few `principal` types and a recommended best practice is to specialize each action to a resource type (e.g., `Action::"GetList"`), so validation time tends to be $O(na)$ where n is the size of the policies and a is the number of actions.

2.6 Analyzing policies

The validator can flag certain mistakes, but it cannot answer deeper questions about policy behavior. One such question is policy equivalence—do two (sets of) policies evaluate to **`true`** on the same set of requests? For that, we turn to Cedar's *symbolic compiler*, which works by reducing policies to SMT formulas, and discharging questions about their behavior using an SMT solver.

To illustrate, suppose that we drop Policy 2 and replace Policy 0 in Figure 2 with Policy 0.1 shown to the right. Is the resulting policy set (Policy 0.1 together with Policies 1 and 3) equivalent to the original (Policies 0, 1, 3 and 4)? The answer is no: the new policy set accepts fewer requests than the original.

```
// Policy 0.1
permit(
    principal,
    action in [Action::"CreateList",
               Action::"GetOwnedLists"],
    resource == Application::"TinyTodo")
unless {
    principal in Team::"interns"
};
```

Equivalence analysis uses the symbolic compiler to generate an SMT formula that is satisfiable if some request is allowed by one policy set but not the other. If the solver finds a model for this formula, the analysis turns it into a concrete counterexample—a request and entity store—on which the two policy sets differ. In this case, given the entity store in Figure 3, the original policy set will let `aaron` perform the `GetOwnedLists` action but the new one will not. This is because Policy 0.1 prevents interns from performing either a `CreateList` or `GetOwnedLists` action. Adding `&& action == Action::"CreateList"` to the condition of Policy 0.1 fixes the problem.

We designed Cedar specifically to support an SMT encoding that is sound, decidable, and complete; no prior authorization language enjoys such an encoding. To achieve it, we use a novel type-based translation that employs only decidable theories, and finite sets of *ground* well-formedness constraints (e.g., to ensure entity graphs are acyclic) rather than *quantified* constraints.

3 SYNTAX, SEMANTICS, AND TYPING

This section presents a partial formalization of the syntax of Cedar policies (Section 3.1), the semantics of Cedar expression evaluation (Section 3.2) and authorization (Section 3.3), and the semantics of policy validation (Section 3.4). We have proved several properties of Cedar's design (Section 3.5) using a formalization in Lean [35]. Details about full Cedar are given in Section 3.6.

3.1 Syntax: Policies and expressions

The concrete syntax of Cedar policies follows the grammar at the top of Figure 5. Nonterminals are formatted in *blue*, terminals are in `red`, and grammatical elements are in black. We write { *a* } to indicate zero or more occurrences of *a*, and [*b*] to indicate zero or one occurrence of *b*.

The grammar references *expressions* e and entity references E::s, whose abstract syntax is given at the bottom of Figure 5. The semantics of a policy c is defined by conversion to an expression e, written as toexp(c). This function conjoins the *Principal*, *Action*, and *Resource* components of c, using `true` is used if the component is just a variable, along with the conditions *Cond*: `when` expressions are used as given, and `unless` expressions are negated. For example, the conversion of Policy 3 from Figure 2 is e_s && (`principal in resource.readers` || `principal in resource.writers`), where e_s is the converted scope `true` && `action` == `Action`::"`GetList`" && `true`.

$$
\begin{array}{llll@{\qquad}llll}
\mathit{Policies}\ (C) & ::= & \{\ \mathit{Policy}\ \} & & \mathit{Policy}\ (c) & ::= & \mathit{Effect}\ (\ \mathit{Scope}\ )\ \{\ \mathit{Cond}\ \}\ ; \\
\mathit{Effect} & ::= & \texttt{permit} \mid \texttt{forbid} & & \mathit{Scope} & ::= & \mathit{Principal}\ ,\ \mathit{Action}\ ,\ \mathit{Resource} \\
\mathit{Principal} & ::= & \texttt{principal}\ [(\texttt{in} \mid \texttt{==})\ E{::}s] & & \mathit{Action} & ::= & \texttt{action}\ [((\texttt{in} \mid \texttt{==})\ E{::}s) \mid (\texttt{in}\ [E{::}s, \ldots])] \\
\mathit{Resource} & ::= & \texttt{resource}\ [(\texttt{in} \mid \texttt{==})\ E{::}s] & & \mathit{Cond} & ::= & (\texttt{when} \mid \texttt{unless})\ \{\ e\ \}
\end{array}
$$

$$
\begin{array}{lcll}
\text{Entity type} & E & \in & \textbf{ID} \qquad \text{Attribute } f \in \textbf{ID} \qquad \text{String } s \qquad \text{Integer } i \qquad \text{Nat } n \\
\text{Variable} & x & ::= & \texttt{principal} \mid \texttt{action} \mid \texttt{resource} \mid \texttt{context} \\
\text{Value} & v & ::= & E{::}s \mid \texttt{true} \mid \texttt{false} \mid s \mid i \mid [v_1, \ldots, v_n] \mid \{f_1 : v_1, \ldots, f_n : v_n\} \\
\text{Expression} & e & ::= & v \mid x \mid [e_1, \ldots, e_n] \mid \{f_1 : e_1, \ldots, f_n : e_n\} \mid e.f \mid e_1\ \&\&\ e_2 \mid e_1 \mid\mid e_2 \mid\ !e \mid -e \\
 & & \mid & e_1\ \mathit{bop}\ e_2 \mid e\ \texttt{like}\ s \mid e\ \texttt{has}\ f \mid e\ \texttt{is}\ E \mid i * e \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
\text{Binop} & \mathit{bop} & ::= & + \mid - \mid < \mid \leq \mid \texttt{==} \mid \texttt{in} \mid \texttt{contains} \mid \texttt{containsAny} \mid \texttt{containsAll} \\
\text{Type} & \tau & ::= & E \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Long} \mid \mathit{True} \mid \mathit{False} \mid \mathit{Set}\ \tau \mid \{\omega_1 f_1 : \tau_1, \ldots, \omega_n f_n : \tau_n\} \\
\text{Optionality} & \omega & ::= & \cdot \mid ? \qquad \text{Capability}\ \alpha, \varepsilon\ ::=\ \emptyset \mid \{e.f\} \mid \varepsilon \cup \varepsilon \mid \varepsilon \cap \varepsilon
\end{array}
$$

Fig. 5. Cedar policies, expressions, and types: Syntax

3.2 Expression semantics

We formalize Cedar expression evaluation as a small-step operational semantics with the judgment $\mu, \sigma \vdash e \longrightarrow e'$, which states that under *entity store* μ and *authorization request* σ the expression e reduces to expression e'. The entity store μ is a map from entity references $E{::}s$ to pairs (v, h), where v is a record value and h is the set of the entity's ancestors in the hierarchy. An authorization request σ is a map from variables x to values v, where v is an entity reference when x is either `principal`, `action`, or `resource`, and v is a record when x is `context`. Stuck states in the semantics correspond to raised errors in the implementation.

Figure 6 shows a selection of expression evaluation rules. We do not show any congruence rules as they are straightforward—evaluation is call-by-value and proceeds left to right. (While Cedar supports neither I/O nor mutable state, evaluation order affects the occurrence of run-time errors.)

The first line of rules shows that projecting an attribute f from an entity reference requires looking up the entity reference in the store and projecting from the corresponding record. This will fail if the entity is not present in the store or if its record lacks the requested attribute. Rules for records are similar.

The first rule on the second line shows that values are considered equal if and only if they are syntactically identical. For entity references this amounts to *nominal*, rather than *structural*, equality since $\mu, \sigma \vdash E_1{::}s_1 \texttt{ == } E_2{::}s_2 \longrightarrow \texttt{false}$ even when $\mu(E_1{::}s_1) = \mu(E_2{::}s_2)$. The second rule on that line shows that v_1 `is` E amounts to a dynamic check of whether v_1 has the type E. The last rule handles request variable lookup.

The third line shows that membership in the entity hierarchy, $E_1{::}s_1$ `in` $E_2{::}s_2$, evaluates to `true` if $E_2{::}s_2$ is $E_1{::}s_1$ or is a member of the ancestors of $E_1{::}s_1$. Otherwise, the result is `false`.

Per the fourth line, the `in` operator with its RHS as a *set* of entities evaluates to `true` if the LHS is `in` at least one entity in the set (`false` otherwise). As usual, conditionals short-circuit; note they are not well-defined for non-boolean guards. Expressions $e_1 \mid\mid e_2$ and $e_1\ \&\&\ e_2$ evaluate equivalently to `if` e_1 `then true else` (`if` e_2 `then true else false`) and `if` e_1 `then` (`if` e_2 `then true else false`) `else false`, respectively.

$$(1) \qquad \frac{\mu(E{::}s) = (\{\ldots, f{:}v, \ldots\}, \_)}{\begin{array}{c}\mu, \sigma \vdash E{::}s\ \texttt{has}\ f \longrightarrow \texttt{true} \\ \mu, \sigma \vdash E{::}s.f \longrightarrow v\end{array}} \qquad \frac{\begin{array}{c}\mu(E{::}s)\ \text{undef},\ \textit{or} \\ \mu(E{::}s) = \{g_1{:}v_1, \ldots, g_n{:}v_n\} \qquad f \notin \{g_1, \ldots, g_n\}\end{array}}{\mu, \sigma \vdash E{::}s\ \texttt{has}\ f \longrightarrow \texttt{false}}$$

$$(2) \qquad \frac{\begin{array}{c}v_1 = v_2 \Rightarrow v = \texttt{true} \\ v_1 \neq v_2 \Rightarrow v = \texttt{false}\end{array}}{\mu, \sigma \vdash v_1 == v_2 \longrightarrow v} \qquad \frac{\begin{array}{c}(v_1 = E_1{::}s \wedge E = E_1) \Rightarrow v_2 = \texttt{true} \\ (\textit{otherwise}) \Rightarrow v_2 = \texttt{false}\end{array}}{\mu, \sigma \vdash v_1\ \texttt{is}\ E \longrightarrow v_2} \qquad \frac{\sigma(x) = v}{\mu, \sigma \vdash x \longrightarrow v}$$

$$(3) \qquad \frac{\begin{array}{c}E_2{::}s_2 = E_1{::}s_1,\ \textit{or} \\ \mu(E_1{::}s_1) = (\_, h_1) \qquad E_2{::}s_2 \in h_1\end{array}}{\mu, \sigma \vdash E_1{::}s_1\ \texttt{in}\ E_2{::}s_2 \longrightarrow \texttt{true}} \qquad \frac{\begin{array}{c}E_2{::}s_2 \neq E_1{::}s_1 \\ \mu(E_1{::}s_1)\ \text{undef}\ \ \textit{or}\ \mu(E_1{::}s_1) = (\_, h_1)\ \textit{and}\ E_2{::}s_2 \notin h_1\end{array}}{\mu, \sigma \vdash E_1{::}s_1\ \texttt{in}\ E_2{::}s_2 \longrightarrow \texttt{false}}$$

$$(4) \qquad \frac{\begin{array}{c}(\exists i.\ \mu, \sigma \vdash E{::}s\ \texttt{in}\ E_i{::}s_i \longrightarrow \texttt{true}) \Rightarrow v = \texttt{true} \\ (\forall i.\ \mu, \sigma \vdash E{::}s\ \texttt{in}\ E_i{::}s_i \longrightarrow \texttt{false}) \Rightarrow v = \texttt{false}\end{array}}{\mu, \sigma \vdash E{::}s\ \texttt{in}\ [E_1{::}s_1, \ldots, E_n{::}s_n] \longrightarrow v} \qquad \begin{array}{c}\mu, \sigma \vdash \texttt{if true then}\ e\ \texttt{else}\ e' \longrightarrow e \\ \mu, \sigma \vdash \texttt{if false then}\ e'\ \texttt{else}\ e \longrightarrow e\end{array}$$

Fig. 6. Cedar expression evaluation semantics: Selected rules

Boolean operations on sets include `contains`, which checks element membership, `containsAny`, which checks overlap, and `containsAll`, which checks containment. The `like` operator matches strings, interpreting $*$ in the style of the Unix shell. Integers can be compared, added, subtracted, and multiplied by a constant.

Cedar's semantics is generally forgiving for operations on entity references that do not exist in μ, to minimize the data required to decide a request. For example, == happily compares non-existent references, so operations like `action` == `Action`::"`foo`" can succeed even when μ(`Action`::"`foo`") is undefined. Evaluation *will* get stuck on a projection $E{::}s.f$ when $E{::}s$ does not exist, since it is unclear what value to return. We considered creating a *null* value, but decided to avoid repeating that "billion dollar mistake" [25]!

3.3 Authorization and slicing

Cedar's authorization algorithm is given in the following pseudocode:

```
1  def authorize($\mu, C, \sigma$) =
2      def evaluate($e$) = $v$ where $\mu, \sigma \vdash e \longrightarrow^* v$
3      let $C_S = \{c \mid c \in \text{slice}(\sigma, C) \wedge \text{evaluate}(\text{toexp}(c)) = \texttt{true}\}$
4      if forbids($C_S$) = $\emptyset$ $\wedge$ permits($C_S$) $\neq \emptyset$ then Allow else Deny
```

Procedure authorize() takes policies C, a request σ, and an entity store μ, and returns a decision *Allow* or *Deny*. Line 2 defines partial function evaluate(e) to return v if e evaluates to it via the transitive closure of the semantics $\longrightarrow^*$. Line 3 uses this function to evaluate the request. First, it *slices* out the policies in C that are relevant to deciding σ (more below). Then, each policy c in the slice is converted to an expression e_c and evaluated; those policies c which *satisfy* the request (i.e., for which e_c evaluates to `true`) are collected in C_S. Finally, per line 4, the request is allowed if there are no satisfying `forbid` policies and at least one satisfying `permit` policy, else it is denied.[1]

Rather than evaluate σ under every policy $c \in C$, authorize uses subroutine slice(C, σ) to quickly select only the policies relevant to σ. Recall that the *Principal* and *Resource* parts of the policy scope optionally constrain the `principal` and `resource`, respectively, to be == or `in` a particular entity. Let $pof(c)$ be the entity E::s named in the *Principal* portion of policy c, or a special identifier `Any` if no entity is named. Let $rof(c)$ behave likewise for the *Resource* portion. Define the key for c to be $\langle pof(c), rof(c)\rangle$. We can store the policies C as a map Σ from keys to sets of policies.

For a request σ we construct a set of keys from the cross-product of the ancestors of P and R in μ, where $P = \sigma(\texttt{principal})$ and $R = \sigma(\texttt{resource})$. That is, let $\mu(P) = (\_, h_P)$ and $\mu(R) = (\_, h_R)$, and let $K = (h_P \cup \{P, \texttt{Any}\}) \times (h_R \cup \{R, \texttt{Any}\})$. K is the set of keys (each of which is a pair) arising from the cross product of P, R, and their ancestors or `Any`. The slice of relevant policies is then $\bigcup_{k \in K} \Sigma(k)$. Other schemes are possible too, e.g., which take into account the policy conditions. This scheme performs well when the policy store is large, but P and R have relatively few ancestors.

3.4 Policy validation

Users can optionally *validate* their Cedar policies prior to use, to ensure that certain run-time errors will not occur. Policies are validated against a *schema*, which is a pair (M, S) where M is the *entity schema* and S is the *action schema*.

- M maps entity types E to pairs (τ, P) where τ is the (record) type of E's attributes, and H contains the types of entities that can be ancestors of entities of type E.
- S maps action entities A (of form `Action`::s) to triples (A_p, A_r, A_x) that describe the allowable shape of requests involving A. In particular, A_p is the set of principal entity types that can accompany A; A_r is the set of resource entity types that can accompany A; and A_x the type of `context` records that can accompany A.

Using the concrete syntax for schemas shown in Figure 4, M is defined by the `entity` declarations, and S is defined by the `action` declarations.

Validating a policy c for a schema (M, S) proceeds in three steps. First, for each action $A \in dom(S)$, we produce a set of *request environments* A_Γ, where request environment $\Gamma \in A_\Gamma$ is a map from variables x to types τ. A_Γ is defined as $\{\ \texttt{principal}{:}E_p,\ \texttt{resource}{:}E_r,\ \texttt{context}{:}A_x \mid E_p \in A_p, E_r \in A_r\ \}$. Second, we convert c into an expression $e = \text{toexp}(c)$ (see Section 3.1), and replace all occurrences of variable `action` in e with A; call the result e_A. Finally, we typecheck e_A for each request environment $\Gamma \in A_\Gamma$. The expression typing judgment has form $\alpha; \Gamma \vdash e : \tau; \varepsilon$, and is described next. Typechecking e_A requires that there exists some ε such that $\emptyset; \Gamma \vdash e_A : \mathit{Bool}; \varepsilon$.

The judgment $\alpha; \Gamma \vdash e : \tau; \varepsilon$ states that under *capability* α and request environment Γ, expression e has type τ and produces capability ε conditioned on e evaluating to `true`. Selected type rules are given in Figure 7a. Types are defined at the bottom of Figure 5, and are notable for the use of *singleton types True* and *False*, and record types with *optional attributes*: a record attribute $\omega f : \tau$ is optional when ω is ?, otherwise it is required; we elide an optionality $\cdot$ when clear from context.

Boolean singleton types. Expressions that definitely evaluate to `true` and `false` may be given singleton types [1] *True* and *False*, respectively, which are subtypes (<:) of *Bool*. To make the SMT encoding (Section 4) tractable, the subtyping rules, shown in Figure 7b, support *depth* subtyping but not *width* subtyping, and there is no subtyping among entity types.

[1]In the Cedar implementation, the *Allow*/*Deny* decision is coupled with extra diagnostics, which include the IDs of the *determining* policies—the `permit` policies that evaluated to `true` for `Allow`, or the `forbid` policies that did so for `Deny`—and any policies that exhibited errors when evaluated.

$$(1)\quad \begin{array}{c}\alpha;\Gamma \vdash \texttt{true} : \mathit{True}; \emptyset \\ \alpha;\Gamma \vdash \texttt{false} : \mathit{False}; \varepsilon \\ \alpha;\Gamma \vdash e == e : \mathit{True}; \emptyset\end{array} \qquad \frac{\begin{array}{c}\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \\ \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2 \qquad E_1 \neq E_2\end{array}}{\alpha;\Gamma \vdash e_1 == e_2 : \mathit{False}; \varepsilon} \qquad \frac{s_1 \neq s_2}{\alpha;\Gamma \vdash E{::}s_1 == E{::}s_2 : \mathit{False}; \varepsilon}$$

$$(2)\quad \frac{\begin{array}{cc}\alpha;\Gamma \vdash e_1 : \tau_1; \varepsilon_1 & \tau_1 <: \tau \quad \tau_2 <: \tau \\ \alpha;\Gamma \vdash e_2 : \tau_2; \varepsilon_2 & \text{for some } \tau\end{array}}{\alpha;\Gamma \vdash e_1 == e_2 : \mathit{Bool}; \emptyset} \qquad \frac{\begin{array}{c}\alpha;\Gamma \vdash e : E_1; \varepsilon \\ (E_1 = E) \Rightarrow \tau = \mathit{True} \qquad (E_1 \neq E) \Rightarrow \tau = \mathit{False}\end{array}}{\alpha;\Gamma \vdash e \ \texttt{is}\ E : \tau; \emptyset}$$

$$(3)\quad \frac{\begin{array}{c}\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2 \\ E_1 \neq E_2 \qquad M(E_1) = (\_, H) \qquad E_2 \notin H\end{array}}{\alpha;\Gamma \vdash e_1 \ \texttt{in}\ e_2 : \mathit{False}; \varepsilon} \qquad \frac{\alpha;\Gamma \vdash e_1 : E_1; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_2 : E_2; \varepsilon_2}{\alpha;\Gamma \vdash e_1 \ \texttt{in}\ e_2 : \mathit{Bool}; \emptyset}$$

$$(4)\quad \frac{\alpha;\Gamma \vdash e_1 : \mathit{True}; \varepsilon_1 \qquad \alpha \cup \varepsilon_1;\Gamma \vdash e_2 : \tau; \varepsilon_2}{\alpha;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau; \varepsilon_1 \cup \varepsilon_2} \qquad \frac{\alpha;\Gamma \vdash e_1 : \mathit{False}; \varepsilon_1 \qquad \alpha;\Gamma \vdash e_3 : \tau; \varepsilon_3}{\alpha;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau; \varepsilon_3}$$

$$(5)\quad \frac{\alpha;\Gamma \vdash e_1 : \mathit{Bool}; \varepsilon_1 \qquad \alpha \cup \varepsilon_1;\Gamma \vdash e_2 : \tau; \varepsilon_2 \qquad \alpha;\Gamma \vdash e_3 : \tau; \varepsilon_3}{\alpha;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau, (\varepsilon_1 \cup \varepsilon_2) \cap \varepsilon_3} \qquad \frac{\alpha;\Gamma \vdash e : \tau; \varepsilon \qquad \tau <: \tau'}{\alpha;\Gamma \vdash e : \tau'; \varepsilon}$$

$$(6)\quad \frac{\begin{array}{c}\alpha;\Gamma \vdash e : \tau; \varepsilon \\ \mathit{attribute}(f,\tau) = \ (?\ f : \tau_f)\end{array}}{\alpha;\Gamma \vdash e\ \mathit{has}\ f : \mathit{Bool}, \{e.f\}} \qquad \frac{\begin{array}{c}\alpha;\Gamma \vdash e : \tau; \varepsilon \\ \mathit{attribute}(f,\tau) = (\omega f : \tau_f) \\ \omega = ? \Rightarrow e.f \in \alpha\end{array}}{\alpha;\Gamma \vdash e.f : \tau_f; \emptyset} \qquad \begin{array}{l}\mathit{attribute}(f,\tau) = \omega f : \tau_f \text{ when} \\ \quad \tau = \{\ldots, \omega f : \tau_f, \ldots\}, \text{ or} \\ \quad \tau = E \text{ and} \\ \qquad\quad M(E) = (\{\ldots, \omega f : \tau_f, \ldots\}, \_)\end{array}$$

(a) Typing

$$\tau <: \tau \qquad \frac{\tau <: u \qquad u <: v}{\tau <: v} \qquad \begin{array}{c}\mathit{True} <: \mathit{Bool} \\ \\ \mathit{False} <: \mathit{Bool}\end{array} \qquad \frac{\tau <: u}{\mathit{Set}\ \tau <: \mathit{Set}\ u}$$

$$\frac{\tau <: \tau' \qquad \omega <: \omega'}{\{\omega_1 f_1 : \tau_1, \ldots, \omega f : \tau, \ldots, \omega_n f_n : \tau_n\} <: \{\omega_1 f_1 : \tau_1, \ldots, \omega' f : \tau', \ldots, \omega_n f_n : \tau_n\}} \qquad \begin{array}{c}\omega <: \omega \\ \\ \omega <: ?\end{array}$$

(b) Subtyping

Fig. 7. Cedar expression typing and subtyping: Selected rules

For ==, rules on lines (1)-(2) of Figure 7a require that to be type-correct the equated expressions must be *comparable*, in the sense that their types have a shared supertype. The rules also leverage that two syntactically identical expressions always evaluate to equal values, whereas two entities with different types or identifiers are surely non-equal.

The last rule on line (2) typechecks `e is E` with singleton types exclusively, since we know expression e's precise type statically. Thus `is` is (only) useful when some request environments $\Gamma \in A_\Gamma$ typecheck e such that $E = E_1$, while others do not. For example, for TinyTodo Policy 1 (Figure 2), the use of `is` is important because `resource` has type `List` in some request environments and type `Application` in others.

For `in`, the first rule on line (3) indicates that e_1 `in` e_2 will have type *False* if the entity type of e_2 is not the same as that of e_1, and is not among e_1's type's ancestors. The rules for expressions `if` e_1 `then` e_2 `else` e_3 on line (4) leverage singleton types to ignore unreachable expressions. The first rule shows that for a *True* guard, the conditional will surely evaluate to e_2, so e_3 can be ignored; the second rule reasons similarly for a *False* guard.

Singleton types, and the rules for conditionals in particular, are important for avoiding false alarms when validating a policy. For example, consider Policy 3 from Figure 2, which converts to the following expression

```
true && (action == Action::"GetList" && (true &&
(principal in resource.readers || principal in resource.writers)))
```

To validate the policy against a schema (S, M) that corresponds to Figure 4, we first consider action `Action::"CreateList"`. We substitute it for `action` in the above expression yielding

```
true && (Action::"CreateList" == Action::"GetList" && (...))
```

We then typecheck this expression under request environment Γ = `principal` : *User*, `resource` : *Application*, `context` : {}. The subexpression (`Action::"CreateList"` == `Action::"GetList"` && (...)) is equivalent to `if Action::"CreateList" == Action::"GetList" then` (...) `else false`. When typing this expression, the guard will have type *False* per the first rule on line (2) of Figure 7a. So, the second rule on line (4) ignores the expression (...), written e_2 in the rule, giving the whole conditional the type of `else` expression, which is *False*. Doing so is critical: The `resource.readers` part of (...) fails to typecheck, since $\Gamma($`resource`$)$ = *Application*, which has no `readers` attribute.

Capabilities for accessing optional attributes. The expression e `has` f checks whether attribute f is present in a record/entity e, and the type system records the effect of this check using *capabilities* [12]. In judgment $\alpha; \Gamma \vdash e : \tau; \varepsilon$, the α represents the set of optional attributes definitely available when e is evaluated, and ε is the set of attributes e proves to be available if it evaluates to `true`.

The first rule on line (6) of Figure 7a types e `has` f, producing capability $\varepsilon = \{e.f\}$ when e's type has an optional attribute f.[2] The second rule on line (6) types expression $e.f$: If f is an optional attribute (i.e., $\omega = ?$) then $e.f$ must be in the capability α. The way it gets there is via the rules for conditionals on lines (4)-(5). The first rules on lines (4) and (5) take capability ε_1 proved for the guard e_1 and add it to α when typing e_2. This makes sense because e_2 is only evaluated when e_1 evaluates to `true`. Putting it all together, when typing expression `if` e `has` f `then` $e.f$ `else false` the guard has type *Bool* and capability $\{e.f\}$, so the rule on line (5) types `then`-expression $e.f$ under capability $\alpha \cup \{e.f\}$, ensuring the second rule on line (6) applies.

The rules for conditionals on lines (4) and (5) prove the available capability ε by leveraging the presence of singleton types. The first rule on line (4) says that when e_1 has type *True* we know that e_2 will be evaluated, so the capability of the entire conditional is $\varepsilon_1 \cup \varepsilon_2$. The second rule says that when e_1 has type *False* we know that e_3 will be evaluated. The final capability in this case is ε_3 and not $\varepsilon_1 \cup \varepsilon_3$ because attributes in ε_1 are only available if e_1 evaluates to `true`, which we know it does not. The first rule on line (5) combines the reasoning of these two. Since the guard has type *Bool* we cannot be sure which branch will be evaluated, so we merge the capabilities of *True* and *False* cases using intersection, i.e., $(\varepsilon_1 \cup \varepsilon_2) \cap \varepsilon_3$.

In general, per rules on lines (1)–(3), we can give an expression e of type *False* an arbitrary capability set ε because doing so is vacuously sound: ε contains the available attributes *only if* e *evaluates to* `true`, which we know it does not. Such rules add very useful precision, especially when handling && and || expressions.

[2]Two other rules, not shown, give this expression type *True* when e's type definitely has attribute f, and type *False* when it definitely does not, both producing an empty capability $\emptyset$.

3.5 Properties

Since the authorizer is part of an application's security *trusted computing base* (TCB), we take extra steps to confirm its design and implementation are correct. We formalize Cedar in Lean [35] and use it to prove the following properties:

Forbid trumps permit: If any `forbid` policy evaluates to `true`, the request is denied.

Default deny: If no `permit` policy evaluates to `true`, the request is denied.

Explicit allow: If a request is allowed, some `permit` policy evaluated to `true`.

Sound slicing: $\forall c \in C.\ c \notin \text{slice}(C, \sigma)$ implies c cannot satisfy request σ.

Validation soundness: Given a policy c and request σ, let $e_A = \text{toexp}(c)[\texttt{action} \mapsto A]$, where $A = \sigma(\texttt{action})$, and let Γ_A be a request environment conforming to σ. If $\emptyset, \Gamma_A \vdash e_A : \mathit{Bool}, \varepsilon$ under entity type map M, then $\mu, \sigma \vdash e_A \longrightarrow^* v$ for all entity stores μ conforming to M, and for which entity references in c and σ are defined.

Carrying out these proofs, and in particular the proof of validation soundness, revealed several bugs. Moreover, writing down a formal spec forced us to think deeply about corner cases, and the process revealed other subtle bugs in our semantics. We also use the Lean spec for testing our implementation, written in Rust, as described below in Section 3.6.

3.6 Full Cedar

For simplicity, this section has omitted discussion of three of Cedar's features: *templates*, *extension types*, and *action groups*; all three are modeled in the full Lean formalization.

Templates are policies with one or more named *slots*. A template is similar in spirit to a SQL *prepared statement*, and can be *linked* into a complete Cedar policy by filling each slot with a Cedar value. At present, Cedar supports only two named slots, `?principal` and `?resource`, which may appear only in the *Principal* and *Resource* parts of the grammar in lieu of specific entities E::s. This restriction makes it easy to typecheck unlinked templates, and to index template-linked policies.

Extension types provide a uniform mechanism for extending the language. Cedar currently supports IP addresses and decimal numbers with this mechanism. Extension-typed values are created with function calls, e.g., `ip("1.2.3.4")` and `decimal("138.22")`, and operated on with method calls, e.g., `resource.amt.lessThan(context.amt)` and `context.addr.isInRange(principal.net)`. The Cedar evaluator implementation provides a plugin mechanism for new extension types.

Cedar actions are entities, so they can also be arranged hierarchically into groups. Action group memberships are specified in the schema so they can be leveraged during validation.

The Lean model is executable by compilation to native code, so we use extensive differential random testing [32, 55] to confirm that our Rust code and Lean model agree. We write input generators (for policies, entity stores, and requests) in the style of Pałka et al. [42], and use the cargo fuzz framework [9] to test that equal inputs map to equal outputs. We also use cargo fuzz to test properties directly on the Rust code, e.g., the "round trip" property that a pretty-printed Cedar abstract syntax tree parses back to itself. This testing regime has been fruitful, uncovering nearly two dozen bugs since the project's inception, and forcing us (via CI) to keep the Lean spec and proofs up to date.

4 SYMBOLIC COMPILATION

This section presents the Cedar symbolic compiler. The compiler works in three stages. First, it encodes Cedar types, entity stores, and requests as SMT types and variables (Section 4.1). Next, it uses this mapping to reduce well-typed Cedar expressions to well-typed SMT terms (Section 4.2). Finally, it constrains the SMT representation of the entity hierarchy to be *sufficiently well-formed* (Section 4.3), using only ground constraints (i.e., those without quantifiers). The compiler's encoding

$$\mathcal{T}(Bool) := \texttt{Bool} \quad \mathcal{T}(Long) := \texttt{(\_ BitVec 64)} \quad \mathcal{T}(String) := \texttt{String} \quad \mathcal{T}(Set\ \tau) := \texttt{(Set } \mathcal{T}(\tau)\texttt{)}$$

$$\mathcal{T}(E) := \iota(E) \text{ where } E \text{ is an entity type}$$

$$\mathcal{T}(R) := \iota(R) \text{ where } R = \{\omega_1 f_1 : \tau_1, \ldots, \omega_n f_n : \tau_n\}$$

$$\mathcal{T}_D(E) := \texttt{(declare-datatype } \iota(E) \texttt{ ((}\iota(E) \texttt{ (eid String))))}$$

$$\mathcal{T}_D(R) := \texttt{(declare-datatype } \iota(R) \texttt{ ((}\iota(R)\, \mathcal{T}_D(R, f_1) \ldots \mathcal{T}_D(R, f_n)\texttt{)))}$$

$$\mathcal{T}_D(R, f_i) := \texttt{(}\iota(R, f_i)\ \mathcal{T}(\tau_i)\texttt{)} \text{ where } R = \{\ldots, \cdot f_i : \tau_i, \ldots\}$$

$$\mathcal{T}_D(R, f_i) := \texttt{(}\iota(R, f_i) \texttt{ (Option } \mathcal{T}(\tau_i)\texttt{))} \text{ where } R = \{\ldots, ?f_i : \tau_i, \ldots\}$$

Fig. 8. Translating a Cedar type to an SMT type. The function $\mathcal{T}$ takes as input a Cedar type and returns the corresponding SMT type expression. The function $\mathcal{T}_D$ generates SMT type declarations. SMT syntax is rendered in code font, and ι maps its arguments to unique SMT identifiers.

is decidable, sound, and complete (Section 4.4). This result, the first of its kind for a non-trivial policy language, is made possible by Cedar's controls on expressiveness, and by leveraging invariants ensured by Cedar's policy validator.

4.1 Encoding types, entity stores, and requests

Cedar's types are designed to enable a direct translation to SMT types, as shown in Figure 8. Primitives and sets are encoded as built-in SMT types (as well as CVC5's theory of sets which is not standardized [6]), and entities and records as SMT algebraic datatypes. An entity type becomes a datatype with a string field `euid` that represents the entity ID. A record type becomes a datatype that encodes a required attribute of type τ as a field of type $\mathcal{T}(\tau)$, and an optional attribute as a field of type `(Option` $\mathcal{T}(\tau)$`)`. The translation uses the function ι to ensure that all occurrences of a given Cedar type map to the same SMT type.

The symbolic compiler leverages this type mapping to represent the request and entity store as a set of *uninterpreted functions and constants.* Figure 9 shows how to generate this representation for a request environment Γ and entity schema M (see Section 3.4). The function $\mathcal{V}_D(\Gamma)$ encodes each variable $x \in \Gamma$ as an uninterpreted constant of type $\mathcal{T}(\Gamma(x))$. The function $\mathcal{F}_D(M)$ encodes the attributes and ancestors of each entity type $E \in M$ as uninterpreted functions: $\mathcal{F}_D(E, R)$ maps entities of type $\mathcal{T}(E)$ to their attributes, while $\mathcal{F}_D(E, E')$ maps entities to their ancestors of type $\mathcal{T}(E')$. Together, these functions and constants represent the set of all possible concrete stores and requests that conform to M and Γ. We refer to them respectively as the *symbolic store* for M and *symbolic request* for Γ.

Example 4.1. *Consider the entity schema M for the TinyTodo schema in Figure 4, and the request environment Γ for the action* `Action::"GetList"`. *The symbolic store and request for M and Γ are defined as follows, with parts of the encoding omitted for brevity, and names prettified for readability:*

```
(declare-datatype User ((User (eid String))))             ; Entity types
(declare-datatype Team ((Team (eid String))))
(declare-datatype List ((List (eid String))))             ; Record types
(declare-datatype ListRecord ((ListRecord (readers Team) (editors Team) (owner User))))
(declare-fun listAttrs (List) ListRecord)                 ; Attribute functions
(declare-fun userInTeam (User) (Set Team))                ; Ancestor functions
(declare-fun teamInTeam (Team) (Set Team))
(declare-const principal User)                            ; principal constant
(declare-const resource List)                             ; resource constant
```

$$\mathcal{V}_D(\Gamma) := \{\mathcal{V}_D(x,\tau) \mid \Gamma(x) = \tau\} \qquad \mathcal{V}(x,\tau) := \iota(x,\tau)$$
$$\mathcal{V}_D(x,\tau) := \texttt{(declare-const } \iota(x,\tau)\ \mathcal{T}(\tau)\texttt{)}$$
$$\mathcal{F}_D(M) := \bigcup \{\mathcal{F}_D(E,R,P) \mid M(E) = (R,P)\} \qquad \mathcal{F}(E,R) := \iota(E,R)$$
$$\mathcal{F}_D(E,R,P) := \{\mathcal{F}_D(E,R)\} \cup \{\mathcal{F}_D(E,E') \mid E' \in P\} \qquad \mathcal{F}(E,E') := \iota(E,E')$$
$$\mathcal{F}_D(E,R) := \texttt{(declare-fun } \iota(E,R)\ \texttt{(}\mathcal{T}(E)\texttt{)}\ \mathcal{T}(R)\texttt{)}$$
$$\mathcal{F}_D(E,E') := \texttt{(declare-fun } \iota(E,E')\ \texttt{(}\mathcal{T}(E)\texttt{)}\ \mathcal{T}(Set\ E')\texttt{)}$$

Fig. 9. Representing entity stores and requests as uninterpreted functions and constants. $\mathcal{V}_D(\Gamma)$ introduces an uninterpreted constant for each variable in the request environment; $\mathcal{F}_D(M)$ introduces a set of uninterpreted functions for each entity type E. $M(E) = (R,P)$ gives the record and set of ancestor entity types for E.

4.2 Encoding expressions

The compiler uses the rules in Figure 10 to reduce a Cedar expression e to an SMT term $\hat{e}$ under a given symbolic store and request. This reduction is formalized using a big-step operational semantics: the judgement $M, \Gamma \vdash e \downarrow \hat{e}$ states that under the symbolic entity store for M and symbolic request for Γ, e reduces to $\hat{e}$. The rules assume e is well-typed, and define a total function from well-typed expressions to well-typed terms. If e has type τ under M and Γ, then $M, \Gamma \vdash e \downarrow \hat{e}$ holds, and $\hat{e}$ is a well-typed term of type `(Option` $\mathcal{T}(\tau)$`)` (Theorem 4.1). We use the `Option` type to encode errors: the term $\hat{e}$ evaluates to `none` on an error not ruled out by the validator (Section 3.5).

Theorem 4.1. *Let e, Γ, and M be an expression, environment, and entity schema such that $\alpha; \Gamma \vdash e : \tau, \varepsilon$ for some α. Then, there is a well-typed SMT term $\hat{e}$ of type* `(Option` $\mathcal{T}(\tau)$`)` *such that $M, \Gamma \vdash e \downarrow \hat{e}$.*

Figure 10 shows a selection of symbolic compilation rules. Omitted rules are defined straightforwardly in terms of the corresponding SMT operators. For example, Cedar equality == reduces to SMT equality = (not shown).

The rules on line 1 handle variables and entity references, translating them to terms of the form `(some` $\hat{v}$`)`. A variable x translates to the uninterpreted constant $\mathcal{V}(x, \Gamma(x))$, which represents the value of x in the symbolic request. The entity reference E::s translates to an application of the constructor for the SMT datatype $\mathcal{T}(E)$ to the string s. This constructs a term representing the entity of type E with entity ID s (Figure 8).

The rules on line 2 translate attribute operations to terms of the form $ifOk(\hat{e}, \hat{v})$. The resulting terms encode error propagation by evaluating to `none` if e errors. For an optional attribute f, the expression $e.f$ translates to $\hat{f}$, which retrieves the optional value of f from the record or entity $val(\hat{e})$. The expression e `has` f translates to a term $\hat{b}$ that evaluates to `true` only if $\hat{f}$ is a `some` value. Required attributes are handled analogously.

The rule on line 3 translates the hierarchy membership test e_1 `in` e_2 on entities. The generated term evaluates to `true` when $\hat{e}_1$ and $\hat{e}_2$ evaluate to entity terms $\hat{v}_1$ and $\hat{v}_2$ satisfying one of two conditions: $\hat{v}_1$ equals $\hat{v}_2$, or $\hat{v}_1$ has ancestors of type E_2 that include $\hat{v}_2$. This encodes a strongly typed version of the dynamic `in` semantics given on line 4 of Figure 6.

Finally, the rules on lines 4 and 5 handle conditional expressions `if` e_1 `then` e_2 `else` e_3. This follows a standard symbolic semantics with merging [44]. If $\hat{e}_1$ reduces to `(some true)` or `(some false)`, the conditional translates to $\hat{e}_2$ or $\hat{e}_3$, respectively. Otherwise, it translates to an `ite` term that selects $\hat{e}_2$ or $\hat{e}_3$ based on the value of $\hat{e}_1$, or errors if $\hat{e}_1$ is `none`.

$$(1)\qquad \frac{\Gamma(x)=\tau}{M,\Gamma\vdash x\downarrow \texttt{(some }\mathcal{V}(x,\tau)\texttt{)}}\qquad \frac{M(E)=(R,\_)}{M,\Gamma\vdash E\text{::}s\downarrow \texttt{(some (}\iota(E)\ s\texttt{))}}$$

$$(2)\qquad \frac{\begin{array}{c}M,\Gamma\vdash e\downarrow\hat{e}\quad \hat{v}=\mathit{val}(\hat{e})\\ \mathit{field}(f,\hat{v})=\langle\cdot,\hat{f}\rangle\end{array}}{\begin{array}{c}M,\Gamma\vdash e\ \texttt{has}\ f\downarrow \mathit{ifOk}(\hat{e},\texttt{(some true)})\\ M,\Gamma\vdash e.f\downarrow \mathit{ifOk}(\hat{e},\texttt{(some }\hat{f}\texttt{)})\end{array}}\qquad \frac{\begin{array}{c}M,\Gamma\vdash e\downarrow\hat{e}\quad \hat{v}=\mathit{val}(\hat{e})\\ \mathit{field}(f,\hat{v})=\langle ?,\hat{f}\rangle\quad \hat{b}=\mathit{isSome}(\hat{f})\end{array}}{\begin{array}{c}M,\Gamma\vdash e\ \texttt{has}\ f\downarrow \mathit{ifOk}(\hat{e},\texttt{(some }\hat{b}\texttt{)})\\ M,\Gamma\vdash e.f\downarrow \mathit{ifOk}(\hat{e},\hat{f})\end{array}}$$

$$(3)\qquad \frac{\begin{array}{c}M,\Gamma\vdash e_1\downarrow\hat{e}_1\quad \hat{v}_1=\mathit{val}(\hat{e}_1)\quad M,\Gamma\vdash e_2\downarrow\hat{e}_2\quad \hat{v}_2=\mathit{val}(\hat{e}_2)\\ \mathit{lift}(\mathit{type}(\hat{v}_1))=E_1\quad \mathit{lift}(\mathit{type}(\hat{v}_2))=E_2\quad M(E_1)=(\_,P)\\ \hat{b}_1=\text{if } E_1=E_2 \text{ then } \texttt{(= }\hat{v}_1\ \hat{v}_2\texttt{)} \text{ else } \texttt{false}\\ \hat{b}_2=\text{if } E_2\in P \text{ then } \texttt{(set.member }\hat{v}_2\ \texttt{(}\mathcal{F}(E_1,E_2)\ \hat{v}_1\texttt{))} \text{ else } \texttt{false}\end{array}}{M,\Gamma\vdash e_1\ \texttt{in}\ e_2\downarrow \mathit{ifOk}(\hat{e}_1,\mathit{ifOk}(\hat{e}_2,\texttt{(some (or }\hat{b}_1\ \hat{b}_2\texttt{))}))}$$

$$(4)\qquad \frac{\begin{array}{c}M,\Gamma\vdash e_1\downarrow\texttt{(some true)}\\ M,\Gamma\vdash e_2\downarrow\hat{e}_2\end{array}}{M,\Gamma\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\downarrow\hat{e}_2}\qquad \frac{\begin{array}{c}M,\Gamma\vdash e_1\downarrow\texttt{(some false)}\\ M,\Gamma\vdash e_3\downarrow\hat{e}_3\end{array}}{M,\Gamma\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\downarrow\hat{e}_3}$$

$$(5)\qquad \frac{\begin{array}{c}M,\Gamma\vdash e_1\downarrow\hat{e}_1\quad \hat{e}_1\neq\texttt{(some true)}\quad \hat{e}_1\neq\texttt{(some false)}\\ \hat{v}_1=\mathit{val}(\hat{e}_1)\quad M,\Gamma\vdash e_2\downarrow\hat{e}_2\quad M,\Gamma\vdash e_3\downarrow\hat{e}_3\end{array}}{M,\Gamma\vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\downarrow \mathit{ifOk}(\hat{e}_1,\texttt{(ite }\hat{v}_1\ \hat{e}_2\ \hat{e}_3\texttt{)})}$$

$$\begin{array}{rcl}\mathit{val}(\hat{e}) & := & \text{if } \hat{e}=\texttt{(some }\hat{v}\texttt{)} \text{ then } \hat{v} \text{ else } \texttt{(val }\hat{e}\texttt{)}\\ \mathit{isSome}(\hat{e}) & := & \text{if } \hat{e}=\texttt{(some }\hat{v}\texttt{)} \text{ then } \texttt{true} \text{ else if } \hat{e}=\texttt{(as none \_)} \text{ then } \texttt{false} \text{ else } \texttt{((\_ is some) }\hat{e}\texttt{)}\\ \mathit{ifOk}(\hat{e}_1,\hat{e}_2) & := & \text{let } \hat{b}=\mathit{isSome}(\hat{e}_1), \hat{e}_3=\texttt{(as none }\mathit{type}(\hat{e}_2)\texttt{)} \text{ in}\\ & & \text{if } \hat{b}=\texttt{true} \text{ then } \hat{e}_2 \text{ else if } \hat{b}=\texttt{false} \text{ then } \hat{e}_3 \text{ else } \texttt{(ite }\hat{b}\ \hat{e}_2\ \hat{e}_3\texttt{)}\\ \mathit{field}(f,\hat{v}) & := & \text{if } \mathit{lift}(\mathit{type}(\hat{v}))=R=\{\ldots,\omega f:\_,\ldots\} \text{ then } \langle\omega,\texttt{(}\iota(R,f)\ \hat{v}\texttt{)}\rangle\\ & & \text{if } \mathit{lift}(\mathit{type}(\hat{v}))=E \text{ and } M(E)=(R,\_) \text{ then } \mathit{field}(f,\texttt{(}\mathcal{F}(E,R)\ \hat{v}\texttt{)})\end{array}$$

Fig. 10. Compiling Cedar expressions to SMT terms: selected rules. We use $\mathit{lift}(\mathit{type}(\hat{v}))$ to denote the Cedar type that corresponds to the SMT type of the term $\hat{v}$.

Example 4.2. *Consider the symbolic store and request from Example 4.1, and the expression* `principal in resource.readers` *from Policy 3 in Figure 4. The symbolic compiler translates this expression to the term* `(some (set.member (readers (listAttrs resource)) (userInTeam principal)))`.

4.3 Sufficiently well-formed hierarchies

The rules in Figure 10 encode expression semantics for an arbitrary entity store that conforms to the entity schema M. But not all such stores are well-formed under Cedar's semantics, which expects the entity hierarchy to be a DAG. This requires the ancestor functions $\mathcal{F}(E,E')$ to collectively define an acyclic, transitive relation on entities. Leaving these functions unconstrained leads to incompleteness, as shown in Example 4.3.

Example 4.3. *Consider the expression* `Team::"A" in Team::"B" && Team::"B" in Team::"A"`. *This expression can never evaluate to true given a well-formed entity hierarchy (a DAG). But its encoding with respect to the symbolic store from Example 4.1 has a model:*

```
  (define-fun teamInTeam ((t Team)) (Set Team) ; maps A -> B and B -> A
    (ite (= t (Team "A")) (set.singleton (Team "B")) (set.singleton (Team "A")))
```

This model does not correspond to any well-formed hierarchy because it contains a cycle between `Team::"A"` *and* `Team::"B"`. *As such, it represents a spurious counterexample (a false alarm) to the assertion that the example expression is always false.*

One solution is to encode acyclicity and transitivity constraints on the ancestor functions directly using quantified formulas. For example, the following formulas assert well-formedness for the function `teamInTeam` from Example 4.1:

```
(assert ; acyclicity: ∀t. t ∉ teamInTeam(t)
  (forall ((t Team)) (not (set.member t (teamInTeam t)))))
(assert ; transitivity: ∀t1, t2. t2 ∈ teamInTeam(t1) → teamInTeam(t2) ⊆ teamInTeam(t1)
  (forall ((t1 Team) (t2 Team))
    (=> (set.member t2 (teamInTeam t1)) (set.subset (teamInTeam t2) (teamInTeam t1)))))
```

However, the quantifiers make the encoding undecidable, causing timeouts or unknown results.

We address this by *grounding* the well-formedness constraints based on the following observation: a Cedar expression e accesses only a finite set of entities from the store. It is sufficient for the store to be well-formed on just this set. We can over-approximate this set, called the *footprint* of e, by collecting all subexpressions of e with an entity type. For example, the expression in Example 4.3 has footprint `{Team::"A", Team::"B"}`, and the expression in Example 4.2 has footprint `{principal, resource, resource.readers}`. Grounding the constraints relative to the footprint maintains decidability while ensuring completeness (Section 4.4).

Figure 11 shows the grounding function $wf(e, \Gamma, M)$. The function emits a transitivity constraint for every pair of footprint expressions of type E_1 and E_2 that may be transitively connected via an entity of type E_3. It emits an acyclicity constraint for every footprint expression of type E_1 that may have an ancestor of the same type. Together, these two constraints force the relation defined by the ancestor functions to be the transitive closure of an underlying DAG.

Example 4.4. *To illustrate grounding, consider again the expression e from Example 4.3, along with* Γ *and* M *from Example 4.1. The compiler uses* $wf(e, \Gamma, M)$ *to generate the following ground well-formedness assertions:*

```
  (assert (not (set.member (Team "A") (teamInTeam (Team "A")))))
  (assert (not (set.member (Team "B") (teamInTeam (Team "B")))))
  (assert (=> (set.member (Team "B") (teamInTeam (Team "A")))
              (set.subset (teamInTeam (Team "B")) (teamInTeam (Team "A")))))
  (assert (=> (set.member (Team "A") (teamInTeam (Team "B")))
              (set.subset (teamInTeam (Team "A")) (teamInTeam (Team "B")))))
```

4.4 Soundness and completeness

The Cedar symbolic encoding is sound and complete for standard policy analyses, such as equivalence or subsumption [5]. Theorem 4.2 states the soundness and completeness result for the equivalence analysis. The proof of the theorem relies on a more general lemma, which we omit for brevity. The general lemma states that the encoding itself is sound and complete with respect to the concrete semantics of Cedar, following prior work [44]. This lemma can be used to show that other analyses based on the encoding are also sound and complete. We have proved soundness and completeness of the encoding on paper. We have also implemented the symbolic encoder in Lean, and proofs of soundness and completeness using the Lean encoder are underway, as of this writing.

$$\begin{aligned}
wf(e,\Gamma,M) &:= \text{let } \epsilon = \{\langle \hat{e}_i, E_i, P_i\rangle \mid subExpr(e_i, E_i, e) \land (\_, P_i) = M(E_i) \land M, \Gamma \vdash e_i \downarrow \hat{e}_i\} \text{ in} \\
&\quad\ \{wfa(\hat{e}_1, E_1) \mid \langle \hat{e}_1, E_1, P_1\rangle \in \epsilon \land E_1 \in P_1\} \cup \\
&\quad\ \{wft(\hat{e}_1, \hat{e}_2, E_1, E_2, E_3) \mid \langle \hat{e}_1, E_1, P_1\rangle \in \epsilon \land \langle \hat{e}_2, E_2, P_2\rangle \in \epsilon \land \\
&\qquad\qquad\qquad\qquad\qquad E_2 \in P_1 \land E_3 \in P_1 \cap P_2\} \\
wfa(\hat{e}_1, E_1) &:= \text{let } \hat{v}_1 = val(\hat{e}_1) \text{ in} \\
&\quad\ implies(isSome(\hat{e}_1), \texttt{(not (set.member } \hat{v}_1 \texttt{ (}\mathcal{F}(E_1,E_1)\ \hat{v}_1\texttt{))))} \\
wft(\hat{e}_1, \hat{e}_2, E_1, E_2, E_3) &:= \text{let } \hat{v}_1 = val(\hat{e}_1), \hat{v}_2 = val(\hat{e}_2), \hat{b} = and(isSome(\hat{e}_1), isSome(\hat{e}_2)) \text{ in} \\
&\quad\ implies(and(\hat{b}, \texttt{(set.member } \hat{v}_2 \texttt{ (}\mathcal{F}(E_1,E_2)\ \hat{v}_1\texttt{))}), \\
&\qquad\quad \texttt{(set.subset (}\mathcal{F}(E_2,E_3)\ \hat{v}_2\texttt{) (}\mathcal{F}(E_1,E_3)\ \hat{v}_1\texttt{)))} \\
implies(\hat{b}_1, \hat{b}_2) &:= \text{if } \hat{b}_1 = \texttt{true} \text{ then } \hat{b}_2 \text{ else if } \hat{b}_1 = \texttt{false} \text{ then } \texttt{true} \text{ else } \texttt{(=> } \hat{b}_1\ \hat{b}_2\texttt{)} \\
and(\hat{b}_1, \hat{b}_2) &:= \text{if } \hat{b}_1 = \texttt{true} \text{ then } \hat{b}_2 \text{ else if } \hat{b}_1 = \texttt{false} \text{ then } \texttt{false} \text{ else } \texttt{(and } \hat{b}_1\ \hat{b}_2\texttt{)}
\end{aligned}$$

Fig. 11. Generating ground well-formedness constraints for e, Γ, and M. We use $subExpr(e_i, E_i, e)$ to denote that e_i is a subexpression of e with type E_i, i.e., $\alpha; \Gamma \vdash e_i : E_i, \varepsilon$ for some α and ε.

Theorem 4.2. *Let e_1 and e_2 be well-typed boolean expressions under environment Γ and entity schema M. Let $\hat{e}_1$ and $\hat{e}_2$ be their encodings where $M, \Gamma \vdash e_1 \downarrow \hat{e}_1$ and $M, \Gamma \vdash e_2 \downarrow \hat{e}_2$. Let ϕ be the term $\texttt{(= } isTrue(\hat{e}_1)\ isTrue(\hat{e}_2)\texttt{)}$, where $isTrue(\hat{e}) := \texttt{(= (some true) } \hat{e}\texttt{)}$. Then, the assertions $wf(\phi, \Gamma, M) \cup \{\texttt{(not } \phi\texttt{)}\}$ are unsatisfiable iff either e_1 and e_2 are both true or both untrue on all well-formed inputs conforming to Γ and M.*

5 EVALUATION

We evaluate Cedar by comparing it against two prominent open-source, general-purpose authorization languages, OpenFGA [40] and Rego [36], on three example sets of policies.[3] We briefly discuss how these examples are encoded in each language (Section 5.1), and we compare the performance in terms of evaluation time for authorization requests (Section 5.2). We also examine the performance impact of policy slicing (Section 5.3) and the performance of our SMT encoding (Section 5.4). The code and data used for evaluation is available at https://github.com/cedar-policy/.

5.1 Example applications

We use the following examples to compare Cedar, OpenFGA, and Rego. Full authorization models for each of these examples are given in the appendix of an extended version of this paper [13].

5.1.1 `gdrive` *example.* The `gdrive` example [39] comes from OpenFGA; we reimplemented it in Cedar and Rego. In this application, Users are organized into Groups, and Documents are organized into Folders. Users may own Documents and Folders, and Users and Groups can be granted view access to Documents and Folders. View access is transitive for both Folders and Groups: view access to a Folder entails view access to any sub-Folders and contained Documents; and likewise, Users inherit view access from their Group parents. Other permissions are not transitive: a Folder's owner(s) can create new Documents inside the Folder, but owner(s) of the Folder's parent Folder(s) do not inherit this permission. Ownership also comes with implied permissions: a Document's

[3]We report the results of some small experiments comparing Cedar's expressiveness and performance against cloud service providers' purpose-built policy languages in Section 6.

```
type doc
  relations
  define owner: [user]
  define parent: [folder]
  define viewer: [user, user:*, group#member]
  define can_change_owner: owner
  define can_read: viewer or owner or viewer from parent
  define can_write: owner or owner from parent
  define can_share: owner or owner from parent
```

(a) OpenFGA gdrive declaration of the doc type

```
permit (
  principal,
  action in [Action::"readDocument",
             Action::"writeDocument",
            Action::"shareDocument"],
  resource)
when {
  resource in
     principal.ownedDocuments ||
  resource in
     principal.ownedFolders };
```

(b) Cedar policy controlling `write` and `share` permissions in `gdrive`

Fig. 12. Selected `gdrive` permission rules in OpenFGA and Cedar

owner or owners of the Document's parent Folder(s) can view, share, or write to the Document.[4] Finally, Documents may be marked as "public" in which case any User has view access.

OpenFGA encodes these permissions partly in the *authorization model* and partly in the *tuples* representing relationships between specific entities. This is somewhat analogous to how Cedar has both *policies* and *entity data* respectively. For example, the OpenFGA declaration of the `doc` type in Figure 12(a) indicates that owners of a document, or owners of the document's parent folder, can write to and share the document. In Cedar, we express this permission using the policy in Figure 12(b).

There are many ways to encode the `gdrive` authorization model in Cedar; we implemented two. The first encoding uses static policies over entity data that embody specific permissions, such as a User being granted view access to a Document. The second one uses Cedar templates to express these permissions; the application would grant view access to a new User or Group by linking a Cedar template. These encodings represent different tradeoffs in the design space. We discuss these tradeoffs, including the impact on performance, in Section 5.3.

The encodings of the `gdrive` and `github` examples in Rego are similar to the one for TinyTodo, discussed below.

5.1.2 `github` *example.* OpenFGA also provides a `github` example [38] which we reimplemented in Cedar and Rego. In this application, Users and Repositories are organized into Teams and Organizations. Users and Teams may be granted read/triage/write/maintain/admin privileges on Repositories, and Users or Organizations may be granted read/write/admin privileges on Organizations. Repositories inherit the permissions of their (Organization) owners.

As in the `gdrive` example, these permissions are encoded partly in policies and partly in entity data. For Cedar, we make one simplification over OpenFGA's encoding: whereas OpenFGA supports multiple Repository owners of type User or Organization, we require that repositories have a single Organization owner. As we did for `gdrive`, for `github` we implemented two different Cedar encodings: one using only static policies, and another using Cedar templates.

5.1.3 TinyTodo example. We also reimplemented the TinyTodo example application (Section 2) in OpenFGA and Rego. The OpenFGA translation was straightforward; the most important part is the definition of the `list` type, shown in Figure 13a. This defines the hierarchy wherein all owners of a list are also editors and readers, and all editors are also readers.

[4]OpenFGA's provided authorization model does not consider owners of transitive parent Folder(s) for write and share, but we modified the example to consider transitive parents (in both OpenFGA and Cedar), which is in line with the semantics of Google Drive permissions. Figure 12(a) shows the original OpenFGA declaration for clarity, rather than our modification.

```
type list
  relations
  define owner: [user]
  define editor: [user,team#member] or owner
  define reader: [user,team#member,user:*]
  ↪  or owner or editor
  define get_list: reader
  define update_list: editor
  define create_task: editor
  define update_task: editor
  define delete_task: editor
  define edit_shares: owner
  define delete_list: owner
```

(a) OpenFGA TinyTodo `list` type declaration

```
public_actions :=
  ["Action::\"CreateList\"",
   "Action::\"GetLists\""]

allow = true {
    input.Request.Action in public_actions
}

allow = true {
    input.Request.Resource.Owner ==
    input.Request.Principal
}
```

(b) Rego TinyTodo policy fragment equivalent to Cedar policies 0 and 1 in Section 2

```
allow = true {
    some group in input.Request.Resource.Writers
    group in graph.reachable(input.Data.Groups, [input.Request.Principal])
}
```

(c) Rego TinyTodo policy fragment for computing membership transitively

Fig. 13. OpenFGA and Rego code for TinyTodo

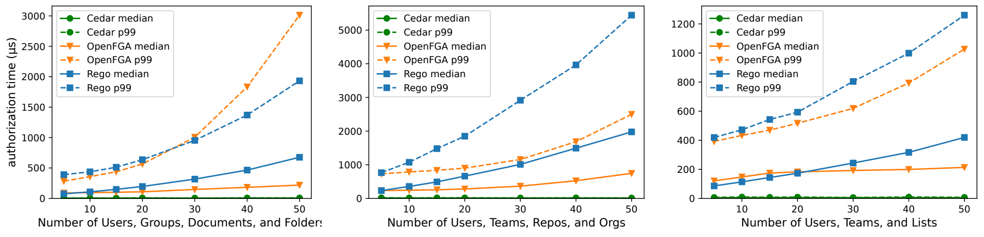
Most of the Rego translation was also straightforward, resulting in Rego code like in Figure 13b. We encoded the graph of team membership using a series of JSON objects with `super` and `users` attributes. Unlike Cedar and OpenFGA, Rego does not have any built in notion of a transitive relation. This can be encoded by either using Rego's standard library graph algorithms, or pre-computing transitive closure over all input data to Rego, moving a substantial chunk of authorization logic out of Rego and into the application. We encode the transitivity within the policy, using graph reachability as in Figure 13c.

5.2 Performance evaluation

We compare the authorization performance of Cedar, Rego, and OpenFGA on each of the application examples just presented. We consider the time it takes to authorize a request when all policies and entity data are available in memory. We thus factor out any time taken to retrieve policies and data from stable storage (if kept there) to focus our comparison on language-level evaluation time.

For each of the application examples, we implemented a random generator that produces entity and request data in the appropriate format, complying with each application's assumed invariants. Our random generators produce edges in the entity graph (e.g., user-in-group relations, folder-viewable-by-user relations, etc.) in proportion to the number of possible edges in the graph, so the number of edges increases superlinearly as more entities are added. As we conduct performance testing, we ensure that the responses (allow/deny) returned by each of the authorization engines agree for each request; this confirms that the translation is correct and the respective implementations are solving the same underlying authorization problems.

5.2.1 Experimental setup. We conducted performance tests on an Amazon EC2 `m5.4xlarge` instance running Amazon Linux 2. We used Cedar version 3.0.1, Rego version 0.61.0, and OpenFGA commit



(a) `gdrive` example (b) `github` example (c) TinyTodo example

Fig. 14. Performance results for Cedar, OpenFGA, and Rego on the `gdrive`, `github`, and TinyTodo examples

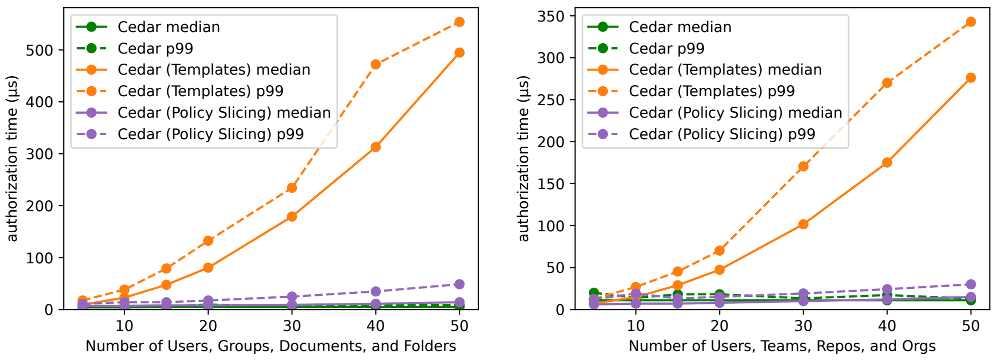
`bbb4a07`.[5] For each datapoint in the graphs in Figure 14, we generated 200 separate datastores, and 500 random `is_authorized()` requests for each, for a total of 100,000 requests. We time the execution of the core `is_authorized()` operation in Cedar, OpenFGA, and Rego, not including the time required to (for instance) initialize the entity data, parse the OpenFGA authorization model or Cedar/Rego policies, or perform HTTP-related processing. We report the median and p99 (99th percentile) execution times across these 100,000 requests.

5.2.2 Results. Averaged across all tested input sizes, and considering the median performance, Cedar is 28.7×, 34.4×, or 35.2× faster than OpenFGA (for `gdrive`, `github`, and TinyTodo respectively). By the same measure, Cedar is 60.4×, 80.8×, or 42.8× faster than Rego, respectively.

Drilling down further, we also see a marked difference in scaling behavior. In the `gdrive` case, Rego's performance deteriorates from median 76μs with 5 Users/Groups/Documents/Folders, to median 676μs with 50 Users/Groups/Documents/Folders. Similarly, at p99, Rego's performance deteriorates from 391μs to 1933μs. Rego's scaling behavior on `github` shows similar trends. Meanwhile, OpenFGA scales well in the median case, with only a modest increase from 89μs to 219μs for `gdrive` as the number of entities increases, or 235μs to 746μs for `github`. Interestingly, increasing the input size affects OpenFGA's p99 performance more strongly, with the `gdrive` p99 increasing from 283μs to 3012μs as the number of entities increases. In contrast to both Rego and OpenFGA, Cedar scales well (with our default, non-templates-based encoding): it handles `gdrive` authorization requests in a median 4.0μs with 5 Users/Groups/Documents/Folders, increasing only to 5.0μs with 50 Users/Groups/Documents/Folders. Likewise, its `github` performance is around median 11.0μs across this entire input range. Cedar's p99 performance is also strong and consistent, under 10μs (`gdrive`) or 20μs (`github`) for all tested input sizes.

The `gdrive`, `github`, and TinyTodo examples rely on transitive relations in the authorization logic. Since Cedar and OpenFGA provide primitives for reasoning about transitive relationships, they are better equipped to handle these policies. In Rego, the user has to encode the transitivity of rules within the policy, resulting in both a reduction of clarity and increased running time. A user of Rego could instead opt to maintain the transitive closure as an invariant on the inputs to Rego, but this comes at the cost of moving a chunk of the authorization logic outside of the authorization policy language. Doing this improves the scaling behavior of Rego a great deal. For these three examples, Rego policies are hard to read and do not scale as the other two solutions.

[5]Based on communication with the OpenFGA developers, the OpenFGA in-memory datastore is intended mainly for debugging and is not optimized, but obviously it is a fairer comparison than would be OpenFGA's persistent-database-backed datastore.



(a) `gdrive-templates` example (b) `github-templates` example

Fig. 15. Performance results of sound policy slicing

5.3 Policy slicing

The sound policy slicing scheme presented in Section 3.3 does not benefit our Cedar `gdrive` and `github` examples because their policies do not have scope-level constraints on `principal` and `resource`. However, policy slicing does benefit the template-based encodings of these examples discussed in Section 5.1. (We name these variants `gdrive-templates` and `github-templates`, respectively.) This is because linked templates have `principal` or `resource` constraints in the policy scope, which serve as policy indexing keys.

`gdrive-templates` contains 4 static policies and 1 template whereas `github-templates` contains 3 static policies and 5 templates. We use templates to grant specific permissions to individual principals. For instance, we write the following template to grant a certain User, Team, or Organization write access to a repository.

```
permit (principal in ?principal, action in Action::"writeRepository", resource == ?resource);
```

We implement the policy slicing algorithm in Section 3.3 and evaluate its performance by randomly generating template links and requests. For each possible pair of principal and resource entities, we generate a template link for that pair with probability 0.05. So, the expected number of template-linked policies is quadratic in the number of entities per entity type. We use the same experimental setup as described in Section 5.2.1.

We evaluate policy slicing by comparing our default Cedar encodings with no templates ("Cedar"), our template-based Cedar encodings without policy slicing ("Cedar (Templates)"), and our template-based Cedar encodings with policy slicing enabled ("Cedar (Policy Slicing)"). The results are shown in Figure 15. With templates but without policy slicing, Cedar's performance is much poorer: there are many more policies to evaluate, and this more than offsets the performance benefit from the template encodings' slightly less complex policies and slightly less entity data. With policy slicing enabled, most of the policies are excluded from the slice, and overall performance returns to comparable to our default Cedar encoding without templates—averaged across all tested input sizes, the Cedar (Policy Slicing) median is about 18% faster than the Cedar median for `github`, but about 94% slower than Cedar for `gdrive`.

5.4 SMT encoding

The Cedar symbolic compiler can be used to implement many different SMT-based analyses. One particularly useful analysis is checking that refactorings are done correctly. This requires analyzing

```
permit (
  principal,
  action == Action::"writeRepository",
  resource)
when { principal in resource.owner.writers };
permit (
  principal,
  action == Action::"triageRepository",
  resource)
when { principal in resource.owner.writers };
```

(a) Original policies

```
permit (
  principal,
  action in
    [Action::"writeRepository",
    Action::"triageRepository"],
  resource)
when {
  principal in resource.owner.writers
};
```

(b) Refactored policy

Fig. 16. Analysis example: Equivalence after a refactoring of `github` policies

the whole policy set, and is thus a good way to evaluate the scalability of our encoding. To perform this evaluation, we consider a refactoring operation on each example application.

We illustrate the refactoring using the `github` model discussed in Section 5.1.2. Suppose we started with the two policies in Figure 16a and wanted to replace them with the policy in Figure 16b. This refactoring is correct: it doesn't change the set of requests that satisfy the policy set under the given schema. The refactoring may seem obvious, but it is only correct because none of the actions listed have children in the schema. If we modify the schema to add a child action, the refactoring is no longer valid, because of the semantics of `in` on sets (see line 4 in Figure 6):

```
action "bug_inducing" in ["writeRepository"]
  appliesTo { principal: [User], resource: [Repository] }
```

We perform this same refactoring (expanding policies with lists of actions into multiple policies comparing actions using `==`) to all of the policies in each of our applications. The overall timing results and times spent in the SMT solver are given in Table 1.

Table 1. Median running times (50 trials) of SMT encoding + solving

Refactoring	`github` (ms)	`gdrive` (ms)	TinyTodo (ms)
Correct	27.1+33.8	16.4+28.9	52.2+67.4
Buggy	29.9+40.9	19.6+34.9	56.2+77.3

All experiments are run 50 times on a Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz. We use CVC5 [6] version 1.0.5 as our solver. Note that the invalid refactorings generally take longer to solve because we invoke the solver once for each action and the invalid cases have the additional action, `bug_inducing`.

6 RELATED WORK

Authorization policy languages have been studied extensively over the last few decades. Table 2 presents a qualitative comparison of Cedar to closely related prior languages. The comparison is along six axes: (1) *Expressiveness* considers support for common access control models, `forbid` policies, and search and string matching operations; (2) *Syntax* considers the readability of the language syntax; (3) *Performance* considers support for policy indexing and the running time of a policy in terms of policy and input size, both typical and worst cases; (4) *Formal* considers whether the language has a formal semantics and metatheory for security-relevant properties; (5) *Validation* considers support for policy validation, ranging from linting to sound type checking;

Table 2. Qualitative comparison with prior authorization languages

Language	Expressiveness						Syntax	Performance		Formal	Validation	Analysis
	RBAC	ABAC	ReBAC	`forbid`	search	str		index	eval wc (typ)			
Cedar	●	●	◖	●	○	○	●	yes	poly (lin)	yes	●	●
XACML	◖	●	-	●	◖	●	○	partial	poly	partial	±◖	○
OPA/Rego	◖	●	◖	◖	●	◖	◖	partial	exp (poly)	no	●	-
Zanzibar	●	-	●	●	-	-	●	yes	lin	no	●	±◖
Ponder	●	●	●	●	●	◖	◖	yes	exp (poly)	no	◖	±○
Cassandra	●	◖	◖	-	●	*e*	◖	no	exp (poly)	yes	●	±○
SecureUML	●	●	◖	-	○	◖	○	yes	poly	partial	±●	±○

●	full, native support	-	no support
◖	partial support, or full support with encoding	*e*	support via extension
○	minimal support	±	possible support, none now, and unlikely to be full support

and (6) *Analysis* considers whether the language supports semantic policy analysis. Elements of this characterization are necessarily subjective, and complete information was not always available.

Cedar natively supports RBAC and ABAC, can encode ReBAC, and supports `forbid` policies. Cedar has rudimentary string matching, and limits searching to membership in sets and the entity hierarchy. Cedar's syntax directly expresses authorization concepts in simple terms. Cedar policies generally have linear-time performance; set containment operations can make running times quadratic in the worst case. Cedar has a formal semantics, with metatheory proved in Lean. It offers a sound schema-based policy validator, as well as a sound and complete encoding for the full language to a decidable fragment of SMT-LIB.

XACML [50] is an open source declarative policy language. XACML provides native support for ABAC and `forbid` (called *deny*) rules; RBAC can be encoded, but it is unclear how to encode ReBAC. XACML provides slightly more general search and string-matching combinators compared to Cedar. XACML's syntax is XML-based, which can be hard to read; ALFA [52] defines a more readable syntax for XACML. XACML policies are labeled with an explicit set of *target* conditions that form the basis of policy indexing, analogously to Cedar policies' scope. XACML has typed operators, but no schema-style validator (that we could find). Several prior works developed analyses for XACML policies, leading to partial formalizations of the language [10]. Fisler et al. [22] developed Margrave, which uses multi-terminal binary decision diagrams to provide sound satisfiability and change impact analysis for a limited subset of XACML. Hughes and Bultan [27] analyze a different subset of XACML by translation to SAT. Their encoding is neither sound nor complete.

Open Policy Agent (OPA) [36] is an open source authorization system that uses a Datalog-based language called Rego. Rego, like other Datalog-based languages (some are discussed below), is more expressive than Cedar, allowing users to define their own notions of evidence prioritization and combination, and data hierarchy. Rego naturally supports `permit`-style ABAC policies, and is powerful enough to encode RBAC, ReBAC, and `forbid` policies. Rego supports expressive string matching and search via recursive rules. Rego's added expressiveness is a blessing but also a curse. It provides no built-in authorization concepts such as 'allow' or 'deny', leaving policy authors to make their own choices to encode these concepts, which later policy readers may find difficult to understand (especially if they are not experts). Rego's evaluation performance can also be hard to anticipate. While typical uses for authorization should be polynomial [31], worst-case performance is exponential [16]. Datalog's resolution algorithm is behind the scenes, so small policy differences can potentially lead to performance surprises [48, 53]. The Rego documentation suggests policy writers limit themselves to a "linear fragment," though it is not precise about what that fragment is [37]. OPA supports indexing but rules must adhere to certain restrictions [47]. Rego provides a policy validator that leverages JSON schema [46], but it is neither specified carefully nor proved sound. Producing an analyzer would be difficult, as Datalog program equivalence is undecidable [49].

Zanzibar [43] is a highly scalable, relationship-based access control (ReBAC) system that Google uses to manage permissions for its cloud products. The system defines authorization in terms of relationships between users and resources. Zanzibar's implementation is proprietary, but there are several open source clones, such as Ory Keto [41], AuthZed SpiceDB [2], and Auth0 Fine Grained Authorization (FGA) [40]. Zanzibar supports RBAC and ReBAC with a natural syntax, and with policy validation. It does not support ABAC, string matching, or search. `forbid`-style policies are supported by expressing one relationship in terms of non-membership in some other relation. Zanzibar can load authorization models lazily, providing a kind of indexing [43]. Our experiments show that OpenFGA's performance is roughly linear in the number of tuples involved in relevant relationships. Zanzibar provides no deep policy analysis now, but it might be possible, e.g., by using the SMT theory of relations [33].

Ponder [14] is a declarative object-oriented policy specification language for writing access control policies, with a readable syntax. It has direct support for ABAC, RBAC, ReBAC, and `forbid` policies. Ponder policy constraints are specified in a subset of the Object Constraint Language [51], which is typed, has no side effects, supports string matching, and provides a variety of higher-order operators that enable sophisticated searches. Ponder deployments support resource-based policy indexes [17], and while Ponder's powerful operators could result in exponential worst-case performance, typical policies should be polynomial. Ponder has no formal semantics, but offers some degree of typechecking. We are not aware of automated analyses for Ponder policies, but there have been analyses developed for OCL via encoding to formal logic for use by SMT solvers [11, 15]. These encodings leave out some OCL features, and are otherwise incomplete due to the use of quantifiers.

SecureUML [30] specifies access rules via UML, in an approach called *model driven security* [7]. RBAC-style rules can be refined by conditions over attributes, which are specified in OCL. Rules are attached to UML actions, supporting a kind of indexing, and OCL conditions generally encourage polynomial-time performance. Analysis for SecureUML leverages analysis for OCL constraints, which has the limitations discussed above.

Cassandra is a Datalog-based authorization language parametrized by a choice of *constraint domain* to support arithmetic operations, set operations, and others. Cassandra can express a variety of access control models and support powerful search and string matching operations, but at the cost of exponential worst-case performance and even greater challenges in building analysis tools. Cassandra focuses on `permit` policies; it is unclear how `forbid` policies could be supported. Cassandra is a *trust management* [8] system, which handles *decentralized* authorization by potentially consulting (or receiving signed certificates from) multiple policy stores instead of a single, centralized policy store. Other similar works (also based on Datalog) include **RT***x* [29], the Query Certificate Manager (**QCM**) [24], and Secure Dynamically Distributed Datalog (**SD3**) [28]. It would be interesting to explore extensions to Cedar to support the decentralized approach.

Cloud service authorization policy languages are purpose-built to protect resources managed by cloud services, including those from Google Cloud [23], Microsoft Azure [4], and Amazon Web Services (AWS) [3]. Cedar is primarily different in that it is general-purpose: policies can be customized to meet an application's needs via Cedar's user-defined principal, resource, and action types; custom entity hierarchies; and arbitrary entity relationships. Nevertheless, Cedar has features that are similar to those in cloud-service authorization languages, and supports some of the same patterns. For example, AWS IAM and Azure policies are also structured around principals/actions/resources, and they support service-specific attributes and roles. While a full translator is outside of the scope of this work, as a simple exercise we closely examined a subset of the AWS IAM policies in the benchmark collected by Eiers et al. [20] (specifically, the policies under the "samples/iam" folder in their repository [45]). We found that we could hand-translate each of these policies to Cedar fairly directly. Authorization times on the translated policies were, averaged over 300 runs with 950 entities, 14μs with random queries and 15μs with well-formed queries. These times are in line with those of our experiments in Section 5.2.

Cedar's symbolic compiler (described in Section 4) likewise bears similarities to previous work on symbolic analysis of cloud-based authorization languages. Backes et al. [5] present the first symbolic analysis of AWS IAM policies, with the aim of detecting policy misconfigurations. Eiers et al. perform symbolic analysis of AWS IAM and Microsoft Azure policies, using model counting to quantify permissiveness [19], compare policies' permissiveness [20], or perform policy repair [18]. Our symbolic compiler for Cedar differs from these in that its encoding is both sound and complete, which is made possible by Cedar's careful language design. This encoding can be a building block for follow-on analyses; for instance, it would be interesting to build analogues of the policy-misconfiguration or model-counting analyses from these previous works on top of our Cedar encoding.

7 CONCLUSION

This paper has presented Cedar, a new authorization policy language whose design carefully balances expressiveness, safety, performance, and analyzability. Cedar is used at scale in products and services from Amazon Web Services, and is open source. All code, documentation, and example applications can be found at https://github.com/cedar-policy/.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their helpful feedback on a draft of this paper; Sarah Cecchetti, Julian Lovelock, Abhishek Panday, and Mark Stalzer for their contributions to the development of Cedar; Bhakti Shah for contributions to Cedar's Lean development; and Andrew Gacek for his help in constructing our TinyTodo sample application.

REFERENCES

[1] David Aspinall. 1994. Subtyping with singleton types. In *CSL: International Workshop on Computer Science Logic*. https://doi.org/10.1007/BFb0022243

[2] authzed-spicedb 2024. spicedb. https://github.com/authzed/spicedb. Open Source, Google Zanzibar-inspired permissions database to enable fine-grained access control for customer applications.

[3] aws-iam 2024. Access Management – AWS Identity and Access Management (IAM). https://aws.amazon.com/iam/.

[4] azure-policy 2024. Azure policy documentation. https://learn.microsoft.com/en-us/azure/governance/policy/.

[5] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. https://doi.org/10.23919/FMCAD.2018.8602994

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[7] David Basin, Manuel Clavel, and Marina Egea. 2011. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies (SACMAT)*. https://doi.org/10.1145/1998441.1998443

[8] Matt Blaze, Joan Feigenbaum, and Martin Strauss. 1998. Compliance checking in the PolicyMaker trust management system. In *Financial Cryptography*. https://doi.org/10.1007/BFb0055488

[9] cargo-fuzz 2023. Rust Fuzz Book. https://rust-fuzz.github.io/book/cargo-fuzz.html.

[10] Carmine Caserio, Francesca Lonetti, and Eda Marchetti. 2022. A Formal Validation Approach for XACML 3.0 Access Control Policy. *Sensors* 22, 8 (2022). https://doi.org/10.3390/s22082984

[11] Manuel Clavel, Marina Egea, and Miguel Angel García de Dios. 2010. Checking unsatisfiability for OCL constraints. *Electronic Communications of the EASST* 24 (2010).

[12] 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.* https://doi.org/10.1145/292540.292564

[13] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. 2024. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization (Extended version). arXiv:2403.04651 [cs.LO]

[14] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. 2001. The Ponder Policy Specification Language. In *Policies for Distributed Systems and Networks*, Morris Sloman, Emil C. Lupu, and Jorge Lobo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–38. https://doi.org/10.1007/3-540-44569-2_2

[15] Carolina Dania and Manuel Clavel. 2016. OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems.* 65–75. https://doi.org/10.1145/2976767.2976774

[16] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)* 33, 3 (2001), 374–425. https://doi.org/10.1145/502807.502810

[17] Naranker Dulay, Emil Lupu, Morris Sloman, and Nicodemos Damianou. 2001. A policy deployment model for the Ponder language. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No. 01EX470).* IEEE, 529–543. https://doi.org/10.1109/INM.2001.918064

[18] William Eiers, Ganesh Sankaran, and Tevfik Bultan. 2023. Quantitative Policy Repair for Access Control on the Cloud. In *International Conference on Software Testing and Analysis (ISSTA).* https://doi.org/10.1145/3597926.3598078

[19] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying permissiveness of access control policies. In *Proceedings of the 44th International Conference on Software Engineering.* 1805–1817.

[20] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2023. Quacky: Quantitative Access Control Permissivness Analyzer. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22).* Association for Computing Machinery, New York, NY, USA, Article 163, 5 pages. https://doi.org/10.1145/3551349.3559530

[21] David F. Ferraiolo and D. Richard Kuhn. 1992. Role-Based Access Control. In *15th National Computer Security Conference.*

[22] K. Fisler, S. Krishnamurthi, L.A. Meyerovich, and M.C. Tschantz. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* https://doi.org/10.1109/ICSE.2005.1553562

[23] gcp-iam 2024. Identity and Access Management | IAM | Google Cloud. https://cloud.google.com/iam/.

[24] Carl A. Gunter and Trevor Jim. 2000. Policy-directed certificate retrieval. *Software - Practice and Experience* 30, 15 (Dec. 2000). https://doi.org/10.1002/1097-024X(200012)30:15%3C1609::AID-SPE334%3E3.0.CO;2-5

[25] C.A.R. Hoare. 2009. Null References: The Billion Dollar Mistake. Presentation at the QCon conference. https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[26] Vincent C. Hu, D. Richard Kuhn, David F. Ferraiolo, and Jeffrey Voas. 2015. Attribute-Based Access Control. *Computer* 48, 2 (2015). https://doi.org/10.1109/MC.2015.33

[27] Graham Hughes and Tevfik Bultan. 2008. Automated verification of access control policies using a SAT solver. *International journal on software tools for technology transfer* 10, 6 (2008), 503–520. https://doi.org/10.1007/s10009-008-0087-9

[28] Trevor Jim. 2001. SD3: A Trust Management System with Certified Evaluation. In *IEEE Symposium on Security and Privacy.* https://doi.org/10.1109/SECPRI.2001.924291

[29] Ninghui Li, John C. Mitchell, and William H. Winsborough. 2005. Beyond proof-of-compliance: security analysis in trust management. *J. ACM* 52 (2005). https://doi.org/10.1145/1066100.1066103

[30] Torsten Lodderstedt, David Basin, and Jürgen Doser. 2002. SecureUML: A UML-based modeling language for model-driven security. In *International Conference on the Unified Modeling Language.* 426–441. https://doi.org/10.1007/3-540-45800-X_33

[31] Neil Madden. 2022. Is Datalog a good language for authorization? https://neilmadden.blog/2022/02/19/is-datalog-a-good-language-for-authorization/.

[32] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[33] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational Constraint Solving in SMT. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 148–165. https://doi.org/10.1007/978-3-319-63046-5_10

[34] MITRE. 2023. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[35] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[36] OPA 2023. Policy-based control for cloud native environments. https://www.openpolicyagent.org/.

[37] OPA-linear 2023. Open Policy Agent documentation: Linear fragment. https://www.openpolicyagent.org/docs/latest/policy-performance/linear-fragment.

[38] OpenFGA 2023. OpenFGA GitHub sample store. https://github.com/openfga/sample-stores/tree/main/stores/github.

[39] OpenFGA 2023. OpenFGA Google Drive sample store. https://github.com/openfga/sample-stores/tree/main/stores/gdrive.

[40] OpenFGA 2023. OpenFGA: Relationship-based access control made fast, scalable, and easy to use. https://openfga.dev/.

[41] ory-keto 2024. keto. https://github.com/ory/keto. Open Source (Go) implementation of "Zanzibar: Google's Consistent, Global Authorization System".

[42] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*. https://doi.org/10.1145/1982595.1982615

[43] 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 (USENIX ATC)*.

[44] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A Formal Foundation for Symbolic Evaluation with Merging. *Proc. ACM Program. Lang.* 6, POPL, Article 47 (jan 2022), 28 pages. https://doi.org/10.1145/3498709

[45] Quacky 2022. Quacky. https://github.com/vlab-cs-ucsb/quacky/tree/master/samples.

[46] Rego Policy Language: Schema 2023. Policy Language: Schema. https://www.openpolicyagent.org/docs/latest/policy-language/#schema.

[47] Torin Sandall. 2017. Optimizing OPA: Rule indexing. https://blog.openpolicyagent.org/optimizing-opa-rule-indexing-59f03f17caf3.

[48] Torin Sandall. 2020. [OPA issue] Implement loop-invariant code motion optimization. https://github.com/open-policy-agent/opa/issues/2094.

[49] Oded Shmueli. 1993. Equivalence of Datalog queries is undecidable. *The Journal of Logic Programming* 15, 3 (1993), 231–241. https://doi.org/10.1016/0743-1066(93)90040-N

[50] OASIS Standard. 2013. Extensible Access Control Markup Language (XACML) version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[51] OMG Standard. 2014. Object Constraint Language (OCL), version 2.4. http://www.omg.org/spec/OCL/2.4.

[52] OASIS Standard. 2024. ALFA - the Abbreviated Language for Authorization. https://alfa.guide/.

[53] K Tuncay Tekle and Yanhong A Liu. 2010. Precise complexity analysis for efficient Datalog queries. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 35–44. https://doi.org/10.1145/1836089.1836094

[54] Or Weis. 2022. What is Policy as Code? https://www.permit.io/blog/what-is-policy-as-code.

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993498.1993532

Received 20-OCT-2023; accepted 2024-02-24