SymCert: Verifying SMT-Based Policy Analyses

Emina Torlak
Amazon Web Services
Seattle, USA
torlaket@amazon.com

***Abstract*—Cedar is a popular authorization language designed to support sound and complete policy analysis by reduction to SMT. But building these analyses is error-prone: subtle encoding mistakes can silently compromise soundness or completeness, and are hard to catch through testing alone. Cedar's high-assurance development process therefore requires writing and maintaining formal proofs of correctness for each new analysis.**

This paper presents SymCert, a framework implemented in Lean for building verified SMT-based analyses of Cedar policies. SymCert provides a verified symbolic compiler and authorizer for reducing policies to SMT formulas, a hierarchy enforcer for ensuring well-formedness of counterexamples, and a counterexample extractor for converting infinite SMT models into finite Cedar inputs, which is essential for proving analysis completeness. To make verification practical, we develop a modular proof approach that decomposes symbolic compiler correctness into two general properties, reducibility and interpretability, yielding reusable lemmas that simplify proofs across all components.

We evaluate SymCert by verifying five analyses used in a Cedar analysis service at Amazon Web Services. Modeling and proving these analyses took just one work day, resulting in efficient executable models that service developers use for prototyping and differential random testing of production code. We demonstrate SymCert's maintainability through three extensions to support new Cedar features, requiring only modest effort (4 to 8 days each) thanks to our modular proof approach.

I. Introduction

Cedar [10] is a popular open-source language for policy-based authorization. Applications use Cedar policies to control access to their resources. When an access decision is needed, the Cedar authorization engine evaluates these policies to determine whether to allow or deny the request. Cedar is designed to support sound and complete reduction [12] of policy semantics to a decidable fragment of SMT-LIB [4]. This enables effective automated analysis of policies against key access invariants, such as checking that a policy never grants unintended access, or that two policy sets are equivalent.

Because Cedar's correctness is crucial for application security, its core functions are built using verification-guided development [15], which involves formal modeling and verification in Lean [22], production implementation in Rust, and extensive differential random testing between the models and implementation. Cedar already provides a verified authorization engine and type checker. But building verified SMT-based analyses poses an additional challenge: each analysis combines multiple components that reduce policies to SMT and map SMT models back to concrete inputs, all of which must be proven correct and maintained as the language evolves.

This paper presents SymCert, a framework for building verified SMT-based analyses of Cedar policies, implemented in Lean. SymCert provides four verified building blocks: a *symbolic compiler* translates policies to SMT formulas encoding their behavior, a *symbolic authorizer* encodes Cedar's authorization semantics, a *hierarchy enforcer* adds constraints ensuring SMT models correspond to well-formed Cedar inputs, and a *counterexample extractor* converts satisfying models back into concrete Cedar inputs witnessing property violations. Together, these components let developers quickly build new analyses and prove them sound and complete with respect to Cedar's semantics.

To illustrate, consider a *denies-all analysis* (Figure 3) that checks if a policy denies all requests. Figure 1 applies this analysis to a policy (Figure 1a) that permits all users of an online learning platform to take introductory computer science courses. Users and courses are *entities*, which are named objects stored in an *entity store*. Entities can have attributes (e.g., course level) and form a DAG called the *entity hierarchy*, capturing membership relationships such as courses belonging to categories. A *schema* (Figure 1b) defines the types of entities, their attributes, and their hierarchy relationships. The analysis uses the symbolic compiler and hierarchy enforcer to emit an SMT query (Figure 1c) that is unsatisfiable iff the policy evaluates to false on (i.e., denies) all inputs conforming to the schema. The query is satisfiable, producing a model (Figure 1d) from which the extractor produces a concrete counterexample (Figure 1e). To prove this analysis sound and complete, we must show that (1) the symbolic compiler preserves policy semantics, (2) the hierarchy enforcer ensures the entity hierarchy is a DAG, and (3) the counterexample extractor maps the SMT model to a well-formed Cedar input.

A key challenge in proving these properties is the mismatch between Cedar's finite entity stores and their infinite SMT representation. The symbolic compiler encodes entity stores as uninterpreted functions over infinite domains, so the resulting SMT models define ancestor and attribute mappings for all entities, not just those in a given entity store. These models may also violate hierarchy constraints such as acyclicity; for example, `(Category "All")` is its own ancestor in Figure 1d. To bridge this gap, we exploit the fact that uninterpreted functions in models of quantifier-free formulas have finite ranges, enabling SymCert to relate finite entity stores to infinite SMT models. We also show how to repair hierarchy violations in SMT models to extract well-formed Cedar inputs, which is essential for proving analysis completeness.

https://doi.org/

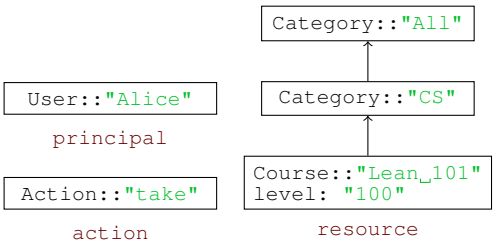
This article is licensed under a Creative Commons Attribution 4.0 International License

```
permit (
  principal,
  action == Action::"take",
  resource in Category::"CS"
) when {
  resource.level == "100"
};
```

(a) Policy

```
entity User {
  premiumPlan?: String // optional
};
entity Category in [Category];
entity Course in [Category] {
  level: String       // required
};
action take appliesTo {
  principal: User,
  resource: Course
};
```

(b) Schema



(e) Counterexample

```
(declare-datatype Category ((Category (eid String))))
(declare-fun Category.ancs (Category) (Set Category))

(declare-datatype Course ((Course (eid String))))
(declare-fun Course.ancs (Course) (Set Category))
(declare-datatype CourseAttrs ((CourseAttrs (level String))))
(declare-fun Course.attrs (Course) CourseAttrs)

(declare-const resource Course)
(define-fun CS () Category (Category "CS"))

(assert      ; Acyclicity
  (not (set.member CS (Category.ancs CS))))

(assert (=>  ; Transitivity
  (set.member CS (Course.ancs resource))
  (set.subset (Category.ancs CS) (Course.ancs resource))))

(assert      ; Verification query
  (and (set.member CS (Course.ancs resource))
       (= (level (Course.attrs resource)) "100")))
```

(c) Simplified SMT query

```
(define-fun resource () Course (Course "Lean 101"))
(define-fun Course.ancs ((c Course)) (Set Category)
  (set.union (set.singleton (Category "All"))
             (set.singleton (Category "CS"))))
(define-fun Course.attrs ((c Course)) CourseAttrs
  (CourseAttrs "100"))
(define-fun Category.ancs ((c Category)) (Set Category)
  (set.singleton (Category "All"))) ; Cycle on (Category "All")
```

(d) SMT model

Fig. 1: A simple denies-all analysis for Cedar. Given a policy (a) and schema (b), the analysis generates an SMT query (c) that is unsatisfiable when the policy is false on (i.e., denies) all well-formed inputs conforming to the schema. The example policy lets all users of an online learning platform take introductory computer science courses. The resulting query is satisfiable, producing a model (d) that corresponds to a concrete counterexample (e): a request and entity store on which the policy evaluates to true. The SMT encoding uses the functions `Course.ancs` and `Category.ancs` to represent the set of all (transitive) ancestors of a given entity in the entity hierarchy DAG, and `Course.attrs` represents the attributes of a course entity.

We prove SymCert's theorems by exploiting two general properties of the symbolic compiler: *reducibility* [23] (the compiler partially evaluates literal inputs) and a property we call *interpretability* (compilation commutes with SMT evaluation). This decomposition yields two simpler theorems whose supporting lemmas are reused to prove correctness of all SymCert components. The approach generalizes to other symbolic compilers [23] that are both reducible and interpretable.

We evaluate SymCert by verifying five analyses from a Cedar analysis service at Amazon Web Services (AWS). Modeling and proving the analyses took one work day, producing efficient executable models that service developers use for prototyping and differential random testing of production code. Applying a demo tool built on these models to example applications [9], we find that analysis queries take less than 20 milliseconds to generate and less than 40 milliseconds to solve, discovering potential authoring mistakes in 3 of 8 benchmarks. We demonstrate SymCert's maintainability through three extensions to support substantial new Cedar features [6]–[8], requiring only 4 to 8 days each thanks to our modular proof approach.

In summary, this paper contributes:

- The SymCert framework for building verified SMT-based analyses of Cedar policies, including a counterexample extractor that enables proving analysis completeness by relating finite Cedar entity stores to infinite SMT models.
- A modular, extensible, and maintainable proof approach based on reducibility and interpretability.
- An evaluation demonstrating SymCert's utility through five verified industrial analyses and its maintainability through three language extensions.

II. The Cedar Policy Language

This section briefly reviews Cedar's syntax, semantics, and typing through a running example. SymCert handles the full Cedar language, described in detail in prior work [12].

A. Example

We illustrate the core features of Cedar using LearnEd, a hypothetical online learning platform. LearnEd allows users to take courses according to a basic or premium subscription plan. Courses have levels and are organized into categories

```
forbid (
  principal,
  action == Action::"take",
  resource in Category::"Premium"
) when {
  !(principal has premiumPlan)
};
```

Fig. 2: A `forbid` policy for LearnEd.

such as subject area. Users, courses, and categories are represented as Cedar *entities* with *attributes* and membership relationships that form a DAG called the *entity hierarchy*.

To manage access control in LearnEd, we define Cedar policies and a schema (Figure 1b). We have already seen a policy (Figure 1a) that *permits* all users to take introductory computer science courses. Figure 2 shows another policy that *forbids* users without a premium subscription to take premium courses.

When a user requests to take a course, LearnEd calls the Cedar authorizer to evaluate these policies and decide if the request is allowed. The authorizer allows the request if it *satisfies* at least one `permit` policy and no `forbid` policies. For example, the request in Figure 1e is allowed because it satisfies Policy 1a but not Policy 2. It would be denied if "Lean 101" was added to the "Premium" category (satisfying Policy 2) or removed from "CS" (no longer satisfying Policy 1a).

B. Syntax, Semantics, and Typing

A Cedar policy consists of an *effect*, *scope*, and an optional *condition*. The effect is to `permit` or `forbid` access, as seen in Policy 1a and 2. The scope optionally constrains the variables `principal`, `action`, and `resource` to be equal to (`==`) or descendents of (`in`) an entity E`::`s, identified by its type name E and entity name s. The condition (`when`) is an *expression* over policy variables, supporting attribute access (`.` and `has`), hierarchy membership (`in`), equality (`==`), and boolean operators. A policy c is desugared to an expression $toExpr(c)$ by conjoining its scope constraints and condition. For example, $toExpr$(Policy 1a) produces the expression `action == Action::"take" && resource in Category::"CS" && resource.level == "100"`.

Cedar defines policy semantics in terms of expression semantics, and authorization semantics (Definition 1) in terms of policy semantics. We write $eval_c(e, \langle \rho, \sigma \rangle)$ to denote the result of evaluating the expression e in the environment $\langle \rho, \sigma \rangle$ consisting of the *request* ρ and *entity store* σ. The result r is either a value v or error. A Cedar value is a *primitive* (boolean b, integer i, or string s), an *entity* reference E`::`s, a *set* of values, or a *record* that maps attributes to values. The request ρ maps policy variables (`principal`, `action`, and `resource`) to values (entity references). The entity store σ maps entity references to pairs of values $\langle v_r, v_h \rangle$, where v_r is the entity's attribute record and v_h is the set of its ancestors in the entity hierarchy. For example, the request ρ_{1e} in Figure 1e maps the variable `resource` to the entity reference `Course::"Lean 101"`, and the store σ_{1e} maps this reference to the attribute record $\{|$level $\mapsto$ `"100"`$|\}$ and ancestor set $\{$`Category::"CS"`, `Category::"All"`$\}$.

```
def verifyDeniesAll
  (C: List Policy) (ε̂: SymEnv): Finding :=
  let t := authorize C ε̂
  let ts := enforce C ε̂
  match solve (ts ++ [t]) ε̂ with
  | .some I => Finding.cex (extract I C ε̂)
  | .none   => Finding.qed
```

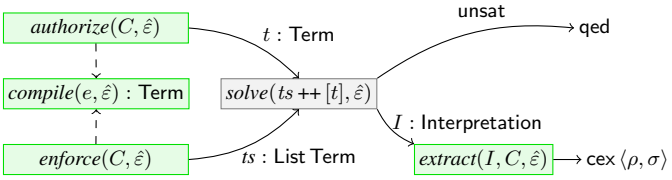


Fig. 3: Using SymCert to model the denies-all analysis from Section I. Verified components are green; trusted ones are gray.

The expression semantics is standard. Variables evaluate to their request value, attribute access (`.` and `has`) looks up the entity's attribute in its record, hierarchy membership (`in`) checks the ancestor set, and boolean operators short-circuit. In our example, $eval_c$(`resource.level`, $\langle \rho_{1e}, \sigma_{1e} \rangle$) produces the string `"100"` by looking up the value of the attribute `level` in the record $\sigma_{1e}[\rho_{1e}[$`resource`$]]$.1. Errors arise from type mismatches, unbound entities or attributes, and arithmetic overflow, and are propagated as usual.

Definition 1 (Authorization semantics)**.** Let C be Cedar policies, ρ a request, and σ an entity store. The request ρ is *authorized* by C in σ, written as $isAuthorized(C, \langle \rho, \sigma \rangle)$, iff it satisfies at least one `permit` policy and no `forbid` policies. The request ρ *satisfies* a policy c in σ iff $eval_c(toExpr(c), \langle \rho, \sigma \rangle) = \mathsf{true}$.

To prevent runtime type errors, Cedar provides a static type system [12] that validates policies against a schema. Both LearnEd policies are well-typed against the schema in Figure 1b. Expressions corresponding to well-typed policies can always be reduced to well-typed SMT formulas. SymCert operates on well-typed expressions.

III. The SymCert Framework

This section presents SymCert's components using the denies-all analysis from Section I as an example (Figure 3). The analysis checks if policies C deny all concrete requests ρ in all concrete environments σ represented by the symbolic environment $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$. The symbolic authorizer $authorize(C, \hat{\varepsilon})$ encodes the authorization semantics of C, and the hierarchy enforcer $enforce(C, \hat{\varepsilon})$ emits the hierarchy well-formedness constraints. Both use the symbolic compiler $compile(e, \hat{\varepsilon})$ to reduce Cedar expressions to terms. The resulting terms are unsatisfiable iff C denies all requests in $\hat{\varepsilon}$. An SMT solver [3] checks satisfiability via the trusted *solve* function. If unsatisfiable, the analysis returns qed. Otherwise, the counterexample extractor $extract(I, C, \hat{\varepsilon})$ converts the solver's interpretation I to a Cedar environment allowed by C.

SymEnv $\hat{\varepsilon}$ ::= $\langle \hat{\rho}, \hat{\sigma} \rangle$
SymReq $\hat{\rho}$ ::= $\{\!|\texttt{principal} \mapsto t_1, \texttt{action} \mapsto t_2, \texttt{resource} \mapsto t_3|\!\}$
SymStore $\hat{\sigma}$ ::= $\{\!|\ldots, E \mapsto \langle f, \hat{h} \rangle, \ldots|\!\}$ where $\hat{h} = \{\!|\ldots, E_i \mapsto f_i, \ldots|\!\}$

Term t ::= $\hat{x} \mid \hat{p} \mid \mathsf{none}\,\hat{\tau} \mid \mathsf{some}\,t \mid \mathsf{set}\,\{t_1, \ldots, t_n\}\,\hat{\tau} \mid \{\!|a_1 \mapsto t_1, \ldots, a_n \mapsto t_n|\!\} \mid \mathsf{app}\,\hat{o}\,[t_1, \ldots, t_n]\,\hat{\tau}$
TermVar $\hat{x}$ ::= $\mathsf{var}\,s\,\hat{\tau}$
TermPrim $\hat{p}$ ::= $b \mid \mathsf{bitvec}\,n\,k \mid s \mid E{::}s$ $\quad \mathsf{Bool}\ b \quad \mathsf{String}\ s$
TermOp $\hat{o}$ ::= $\mathsf{not} \mid \mathsf{and} \mid \mathsf{or} \mid \mathsf{eq} \mid \mathsf{ite} \mid \mathsf{set.member} \mid \mathsf{set.subset} \mid \mathsf{getVal} \mid \mathsf{getAttr}\,a \mid f_u$
UF f ::= $f_u \mid f_t$
UUF f_u ::= $\mathsf{uuf}\,s\,\hat{\tau}_{in}\,\hat{\tau}_{out}$
UTF f_t ::= $\mathsf{utf}\,\hat{\tau}_{in}\,\hat{\tau}_{out}\,\{\!|t_1 \mapsto t_1', \ldots, t_n \mapsto t_n'|\!\}\,t$

TermType $\hat{\tau}$ ::= $\mathsf{Bool} \mid \mathsf{BitVec}\,k \mid \mathsf{String} \mid E \mid \mathsf{Set}\,\hat{\tau} \mid \{\!|a_1 \mapsto \hat{\tau}_1, \ldots, a_n \mapsto \hat{\tau}_n|\!\} \mid \mathsf{Option}\,\hat{\tau}$

Fig. 4: Symbolic environments and a representative subset of terms. The operator getVal retrieves the value of an option term, while getAttr a retrieves the field a of an ADT term. These functions are defined for each ADT declaration (e.g., getAttr level corresponds to the function `level` in Figure 1c). All other operators are built into the SMT-LIB language.

Interpretation I ::= $\langle I_0, I_1 \rangle$
I_0 : TermVar $\to$ Term $\qquad$ I_1 : UUF $\to$ UTF

Evaluating terms (excerpt)
$eval_s(\hat{x}, \langle I_0, \_ \rangle)$ ::= $I_0(\hat{x})$ $\qquad$ $eval_s(\hat{p}, I)$ ::= $\hat{p}$
$eval_s(f_u, \langle \_, I_1 \rangle)$ ::= $I_1(f_u)$ $\qquad$ $eval_s(f_t, I)$ ::= f_t
$eval_s(\mathsf{app}\ f_u\ [t]\ \hat{\tau}, I)$ ::= $appUF(eval_s(f_u, I), [eval_s(t, I)])$
$eval_s(\mathsf{app\ not}\ [t]\ \hat{\tau}, I)$ ::= $not(eval_s(t, I))$

Factory functions (excerpt)
$appUF(f, t)$ ::= match f with
| $\mathsf{uuf}\ s\ \hat{\tau}_{in}\ \hat{\tau}_{out} \Rightarrow (\mathsf{app}\ (\mathsf{uuf}\ s\ \hat{\tau}_{in}\ \hat{\tau}_{out})\ [t]\ \hat{\tau}_{out})$
| $\mathsf{utf}\ \_\_\ tbl\ t_d \Rightarrow$
  if $isLit(t)$ then
    if $tbl[t] = t'$ then t' else t_d
  else let $ts = [\langle t_1, t_2 \rangle \mid tbl[t_1] = t_2]$ in
    $foldr(ts, \lambda\langle t_1, t_2 \rangle.\, t_3.\, ite(eq(t, t_1), t_2, t_3), t_d)$
$not(t)$ ::= match t with
| $b \Rightarrow\ !b$
| $\mathsf{app\ not}\ [t']\ \mathsf{Bool} \Rightarrow t'$
| $\_ \Rightarrow \mathsf{app\ not}\ [t]\ \mathsf{Bool}$

Fig. 5: Representative evaluation rules and factory functions for terms. The function $isLit$ returns true iff its argument is free of variables and application terms. Both $eval_s$ and $isLit$ are defined on all symbolic environments and terms.

The SymCert framework is implemented in Lean [22] and open-sourced [25] as part of the Cedar language specification. The symbolic compiler and hierarchy enforcer were introduced in prior work [12]. SymCert contributes a Lean formalization of both. The authorizer and counterexample extractor are new. The rest of this section describes all four components, focusing on how the extractor converts interpretations into well-formed Cedar environments.

A. Symbolic Terms and Environments

SymCert uses *symbolic terms* (Figure 4) as an intermediate representation of quantifier-free SMT-LIB [4] expressions. A term is either a variable, a primitive (boolean, bitvector, string, or entity reference), a set of terms, a record mapping attributes to terms, an *option* term ($\mathsf{none}\,\hat{\tau}$ or $\mathsf{some}\,t$), or an *operator* application. Most terms correspond to built-in SMT-LIB types; options, entity references, and records are encoded using abstract data types (e.g., `Course` and `CourseAttrs` in Figure 1c).

Term operators are either built-in SMT-LIB operators or *unary uninterpreted functions*. Like variables, uninterpreted functions are unknowns that are resolved by an *interpretation* I (Figure 5). An interpretation assigns *literal terms* to variables and *tabular functions* to uninterpreted functions. Literal terms are free of variables and applications; e.g., $(\mathsf{some}\,\mathsf{true})$ is literal but $(\mathsf{some}\,(\mathsf{app\ not}\,[\mathsf{false}]\,\mathsf{Bool}))$ is not. Tabular functions capture the insight that SMT models represent uninterpreted functions as lookup tables with a default value. A tabular function is a finite map from literal terms to literal terms, with a literal term as the default. For example, $(\mathsf{utf}\,\mathsf{String}\,\mathsf{Bool}\,\{\!|\text{“true”} \mapsto \mathsf{true}|\!\}\,\mathsf{false})$ maps “true” to true and all other strings to false.

The term evaluation function $eval_s(t, I)$ works in standard bottom-up fashion. Variables and uninterpreted functions evaluate to their interpretations in I. Primitives and $\mathsf{none}\,\hat{\tau}$ evaluate to themselves. Sets, records, and $\mathsf{some}\,t$ terms are evaluated recursively, by first evaluating their subterms. Application terms $\mathsf{app}\,\hat{o}\,ts\,\hat{\tau}$ are evaluated by applying a *factory function* for the operator $\hat{o}$ to the result of evaluating the arguments *ts*. Factory functions are a key part of SymCert. They are designed to produce literal outputs when given literal inputs. So, $eval_s(t, I)$ always returns a literal term. All SymCert components use factory functions to build terms rather than constructing them directly, which is key to our modular proof approach (Section IV-B).

We assemble terms and unary functions into *symbolic environments* $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$, which represent sets of concrete Cedar environments. A symbolic environment consists of a symbolic request $\hat{\rho}$ and a symbolic entity store $\hat{\sigma}$. The request $\hat{\rho}$ maps Cedar variables to entity terms. The store $\hat{\sigma}$ maps entity types E to pairs $\langle f, \hat{h} \rangle$ that represent the attributes and ancestors of all entities of type E. In particular, f takes as input a term t of type E and produces a term that represents t's attribute record, while $\hat{h}[E_i]$ is a function from terms of type E to sets of their ancestors of type E_i. We evaluate $\langle \hat{\rho}, \hat{\sigma} \rangle$ with respect to an interpretation I to obtain a *literal symbolic environment* comprised of literal terms and functions.

B. Symbolic Compiler, Authorizer, and Hierarchy Enforcer

The SymCert symbolic compiler $compile(e, \hat{\varepsilon})$: Term translates Cedar expressions to terms with respect to symbolic environments. The compiler uses factory functions to implement the reduction rules from prior work [12] and extends them to support the entire Cedar language. The rules use option terms to encode errors in concrete evaluation: an expression e of type τ reduces to a term of type $\mathsf{Option}\,\hat{\tau}$, where the term type $\hat{\tau}$ encodes the Cedar type τ. The resulting

$$\mathit{extract}(I, C, \hat{\varepsilon}) ::= \mathit{concretize}(C, \mathit{eval}_s(\hat{\varepsilon}, \mathit{repair}(I, C, \hat{\varepsilon})))$$

$$\begin{aligned}
&\mathit{repair}(\langle I_0, I_1\rangle, C, \langle\hat{\rho}, \hat{\sigma}\rangle) ::= \\
&\quad \text{let } \mathit{ft}_{\mathit{uids}} := \{E{::}s \mid \exists t.\, t \in \mathit{footprints}(C, \langle\hat{\rho}, \hat{\sigma}\rangle) \wedge \\
&\qquad\qquad \mathit{eval}_s(t, \langle I_0, I_1\rangle) = \mathsf{some}\, E{::}s\} \text{ in} \\
&\qquad \langle I_0, \lambda f_u.\, \text{if } \mathit{isAncsFun}(\hat{\sigma}, f_u) \\
&\qquad\qquad \text{then } \mathit{repairAncs}(\mathit{ft}_{\mathit{uids}}, I_1(f_u)) \\
&\qquad\qquad \text{else } I_1(f_u)\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathit{repairAncs}(\mathit{ft}_{\mathit{uids}}, f_t) ::= \text{let } (\mathsf{utf}\ E_1\ (\mathsf{Set}\ E_2)\ _\ _) := f_t \text{ in} \\
&\quad \text{let } \mathit{tbl} := \{\!|E_1{::}s \mapsto \mathit{appUF}(f_t, E_1{::}s) \mid E_1{::}s \in \mathit{ft}_{\mathit{uids}}|\!\} \text{ in} \\
&\qquad \mathsf{utf}\ E_1\ (\mathsf{Set}\ E_2)\ \mathit{tbl}\ (\mathsf{set}\ \{\}\ E_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{concretize}(C, \langle\hat{\rho}, \hat{\sigma}\rangle) ::= \\
&\quad \text{let } \mathit{uids} := \mathit{entities}(C) \cup \mathit{entities}(\hat{\rho}) \cup \mathit{entities}(\hat{\sigma}) \text{ in} \\
&\qquad \langle \mathit{concretizeReq}(\hat{\rho}), \mathit{concretizeStore}(\mathit{uids}, \hat{\sigma})\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathit{concretizeReq}(\hat{\rho}) ::= \langle \mathit{toEntity}(\hat{\rho}[\texttt{principal}]), \\
&\quad \mathit{toEntity}(\hat{\rho}[\texttt{action}]), \mathit{toEntity}(\hat{\rho}[\texttt{resource}])\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathit{concretizeStore}(\mathit{uids}, \hat{\sigma}) ::= \\
&\quad \{\!|E{::}s \mapsto \mathit{concretizeEntity}(E{::}s, \langle f, \hat{h}\rangle) \mid \\
&\quad\ E{::}s \in \mathit{uids} \wedge \hat{\sigma}[E] = \langle f, \hat{h}\rangle|\!\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{concretizeEntity}(E{::}s, \langle f, \hat{h}\rangle) ::= \langle \mathit{toRecord}(\mathit{appUF}(f, E{::}s)), \\
&\quad \textstyle\bigcup_{f_a \in \hat{h}} \mathit{entities}(\mathit{appUF}(f_a, E{::}s))\rangle
\end{aligned}$$

Fig. 6: Counterexample extractor. The function $\mathit{entities}(y)$ returns the set of all entity references included in the representation of y; $\mathit{toEntity}(t)$ and $\mathit{toRecord}(t)$ convert the literal term t to the corresponding concrete entity reference or record.

term evaluates to $(\mathsf{none}\,\hat{\tau})$ on inputs where e errors, and $(\mathsf{some}\,t)$ on inputs where e evaluates to a value v represented by t. The factory functions aggressively simplify the encoding, and along with error propagation, were a source of subtle bugs during development. Our proofs caught these bugs and ensure the implementation remains correct as new features are added.

The symbolic authorizer $\mathit{authorize}(C, \hat{\varepsilon})$: Term builds on the symbolic compiler to encode the authorization semantics of a list of policies with respect to a symbolic environment. This encoding is a straightforward implementation of Definition 1. The authorizer builds a term that evaluates to true exactly when at least one `permit` policy and no `forbid` policy evaluates to $(\mathsf{some}\,\mathsf{true})$.

The hierarchy enforcer $\mathit{enforce}(C, \hat{\varepsilon})$: List Term emits constraints ensuring that the ancestor relation on entities is acyclic and transitive; i.e., the underlying entity hierarchy is a DAG. In principle, expressing these constraints requires quantification over all entities, leading to solver timeouts and "unknown" results. To keep the encoding decidable, prior work [12] proposed grounding the hierarchy constraints on the finite set of entities that policies can access, called *footprints*. The enforcer computes a footprint $\mathit{footprints}(C, \hat{\varepsilon}) = \bigcup_{c \in C} \mathit{footprint}(\mathit{toExpr}(c), \hat{\varepsilon})$ that refines prior work by including only entity-typed subexpressions of C reachable from $\hat{\varepsilon}$, and emits ground acyclicity and transitivity constraints for the entities in this set.

C. Counterexample Extractor

SymCert's counterexample extractor $\mathit{extract}(I, C, \hat{\varepsilon})$: Env takes as input an interpretation I, policies C, and symbolic environment $\hat{\varepsilon}$ (Figure 6). If I satisfies the hierarchy constraints $\mathit{enforce}(C, \hat{\varepsilon})$, the extractor produces a *well-formed* Cedar environment that represents a counterexample to a verification query about the authorization semantics of C with respect to $\hat{\varepsilon}$.

Extracting a well-formed environment from I poses two challenges. First, since the enforcer ensures acyclicity and transitivity only on footprint entities, I may violate these constraints on entities outside the footprint. Second, I is defined on *all* entities, but a Cedar environment is defined on a *finite* set of entities. This finite set must be carefully chosen so that the policies C and the counterexample environment $\langle\rho, \sigma\rangle$ are free of dangling (undefined) entity references, which could cause concrete evaluation to error. All Cedar analyses are built with respect to this *closure* assumption: without it, nearly all interesting properties can be violated by ill-formed environments that are missing entities referenced in the policies.

SymCert addresses these challenges with a two-phase extraction process. The *repair* phase modifies I so that it satisfies the hierarchy constraints on all entities. The *concretization* phase extracts a closed Cedar environment from the resulting repaired interpretation by exploiting the finite representation of tabular functions.

The function $\mathit{repair}(I, C, \hat{\varepsilon})$: Interpretation implements the repair phase by first computing the set $\mathit{ft}_{\mathit{uids}}$ of entity references that result from evaluating the footprints of C under I. It then uses $\mathit{repairAncs}(\mathit{ft}_{\mathit{uids}}, I_1(f_u))$ to compute a new interpretation for every ancestor function f_u in $\hat{\sigma}$ such that (1) it returns the same ancestor set as $I_1(f_u)$ for every entity in $\mathit{ft}_{\mathit{uids}}$, and (2) it returns the empty set for every entity outside of $\mathit{ft}_{\mathit{uids}}$. In other words, the repair preserves the solver's solution on the footprint entities, and it supplies a safe default solution for all other entities (the empty ancestor set trivially satisfies both acyclicity and transitivity constraints). So when I satisfies the hierarchy constraints on the footprint, the repaired interpretation satisfies these constraints for all entities whose types are defined in $\hat{\varepsilon}$.

The function $\mathit{concretize}(C, \hat{\varepsilon})$: Env implements the concretization phase by using the functions $\mathit{concretizeReq}$ and $\mathit{concretizeStore}$ to concretize the literal symbolic request $\hat{\rho}$ and store $\hat{\sigma}$. Request concretization $\mathit{concretizeReq}(\hat{\rho})$ simply converts each literal term $\hat{\rho}[x]$ to the corresponding concrete entity reference. Store concretization $\mathit{concretizeStore}(\mathit{uids}, \hat{\sigma})$ takes as input a set of entities *uids* and a literal symbolic store $\hat{\sigma}$. It returns a concrete store that binds each entity $E{::}s$ in *uids* to attributes and ancestors given by $\mathit{concretizeEntity}(E{::}s, \hat{\sigma}[E])$. The key aspect is the choice of *uids*, which is the union of all entities in C, $\hat{\rho}$, and $\hat{\sigma}$. This choice, together with the tabular structure of function interpretations, ensures that all references in the policies and the concrete environment are well-defined. As a result, policy evaluation can never fail due to unbound references. The next section formalizes these well-formedness properties and shows how they enable correctness proofs for analyses built on SymCert.

IV. Proving Correctness of SymCert

This section shows how to prove SymCert-based analyses sound and complete. We define well-formedness and equiva-

lence for concrete and symbolic structures, prove correctness of each SymCert component, and show how these proofs compose to verify the denies-all analysis.

A. Well-Formedness and Equivalence

SymCert's correctness theorems are stated with respect to well-formed inputs: policies, concrete environments, symbolic environments, and interpretations. A concrete environment $\langle \rho, \sigma \rangle$ is *well-formed* for policies C, $\mathit{WellFormed}(\langle \rho, \sigma \rangle, C)$, if (1) every entity reference in C, ρ, and σ is bound in σ (*closure*), and (2) the ancestor relation defined by σ is acyclic and transitive. Similarly, a symbolic environment $\langle \hat{\rho}, \hat{\sigma} \rangle$ is well-formed for C, $\mathit{WellFormed}(\langle \hat{\rho}, \hat{\sigma} \rangle, C)$, if all terms and functions in $\langle \hat{\rho}, \hat{\sigma} \rangle$ are well-typed, all entity types referenced in C, $\hat{\rho}$, and $\hat{\sigma}$ are bound in $\hat{\sigma}$, and literal parts of the ancestor relation defined by $\hat{\sigma}$ are acyclic and transitive. An interpretation I is well-formed for a symbolic store $\hat{\sigma}$, $\mathit{WellFormed}(I, \hat{\sigma})$, if it maps variables and uninterpreted functions to well-formed literals and tabular functions of matching types. All SymCert components preserve well-formedness: given well-formed inputs, they produce well-formed outputs.

Proving correctness of SymCert-based analyses requires relating concrete and symbolic structures, so we can show that policies and their translations produce equivalent outputs given equivalent inputs. The key challenge is defining equivalence for environments: a concrete entity store σ is a finite map over entities, while a symbolic entity store $\hat{\sigma}$ defines attributes and ancestors for *all* entities of each type. We resolve this by observing that closure makes concrete evaluation insensitive to extra bindings: if $\langle \rho, \sigma \rangle$ is well-formed for C and $\sigma \subseteq \sigma'$, then the policies in C evaluate to the same result in both $\langle \rho, \sigma \rangle$ and $\langle \rho, \sigma' \rangle$. This lets us view literal symbolic environments as infinitely large concrete environments and define equivalence as follows.

Definition 2 (Equivalence)**.** A concrete and symbolic environment are equivalent, $\langle \rho, \sigma \rangle \sim \langle \hat{\rho}, \hat{\sigma} \rangle$, iff their requests and entity stores are equivalent, $\rho \sim \hat{\rho}$ and $\sigma \sim \hat{\sigma}$. Requests are equivalent iff their variable bindings are equivalent: $\forall x, \rho[x] \sim \hat{\rho}[x]$. Here, a value v and literal term t are equivalent, $v \sim t$, iff $v = \mathit{toValue}(t)$, i.e., t is a term representation of v. We lift equivalence to evaluation results so that $\mathsf{error} \sim (\mathsf{none}\,\hat{\tau})$ and $v \sim (\mathsf{some}\,t)$ iff $v \sim t$. Entity stores are equivalent iff they bind every entity reference $E{::}s$ in σ to equivalent attributes and ancestor sets: if $\sigma[E{::}s] = \langle v_r, v_h \rangle$, then $\hat{\sigma}[E] = \langle f, \hat{h} \rangle$, $v_r \sim \mathit{appUF}(f, E{::}s)$ and $v_h = \bigcup_{f_h \in \hat{h}} \mathit{toValue}(\mathit{appUF}(f_h, E{::}s))$.

B. Compiler and Authorizer Correctness

SymCert uses the standard definition of correctness for symbolic compilers [23] stated in Theorem 1: an expression and its translation must produce equivalent outputs on equivalent well-formed inputs.

Theorem 1 (Compiler correctness)**.** *Let e be a Cedar expression, ε a concrete environment, $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$ a symbolic environment, and I an interpretation, where $\mathit{WellFormed}(\varepsilon, e)$, $\mathit{WellFormed}(\hat{\varepsilon}, e)$, and $\mathit{WellFormed}(I, \hat{\sigma})$. Then, if $\varepsilon \sim \mathrm{eval}_s(\hat{\varepsilon}, I)$, we have $\mathrm{eval}_c(e, \varepsilon) \sim \mathrm{eval}_s(\mathrm{compile}(e, \hat{\varepsilon}), I)$.*

Proving Theorem 1 directly by induction on e requires each step to reason simultaneously about concrete evaluation, symbolic compilation, and term interpretation. We initially attempted this approach in Dafny [19], producing roughly 8,000 lines of proof for a subset of Cedar. The resulting proof was fragile, difficult to maintain, and its lemmas could not be reused for proving correctness of other SymCert components because they were closely tied to the statement of Theorem 1.

Because Cedar is an actively developed language that evolves with new features, SymCert needs proofs that are modular, maintainable, and reusable across components. We achieve this by decomposing Theorem 1 into two general properties of the compiler: *reducibility* (Theorem 2) and *interpretability* (Theorem 3). Reducibility says that the compiler is a strong partial evaluator: given a literal symbolic environment, it produces a literal output that matches Cedar's semantics on equivalent concrete environments. Prior work [23] identified reducibility as key to efficiency and testability of symbolic compilers. Interpretability says that compilation commutes with term interpretation: interpreting the compiler's output is the same as compiling against the interpreted environment. Interpretability is a new property that we identify in this work; prior compilers [23] satisfy it, but the property was not recognized or exploited for proof decomposition.

Theorem 2 (Reducibility)**.** *Let ε and $\hat{\varepsilon}$ be equivalent and well-formed for e: $\varepsilon \sim \hat{\varepsilon}$, $\mathit{WellFormed}(\varepsilon, e)$, $\mathit{WellFormed}(\hat{\varepsilon}, e)$. Then,* $\mathrm{eval}_c(e, \varepsilon) \sim \mathrm{compile}(e, \hat{\varepsilon})$.

Theorem 3 (Interpretability)**.** *Let $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$ be well-formed for e and I for $\hat{\sigma}$: $\mathit{WellFormed}(\hat{\varepsilon}, e)$ and $\mathit{WellFormed}(I, \hat{\sigma})$. Then,* $\mathrm{eval}_s(\mathrm{compile}(e, \hat{\varepsilon}), I) = \mathrm{compile}(e, \mathrm{eval}_s(\hat{\varepsilon}, I))$.

Together, interpretability and reducibility give compiler correctness for free. Our Lean proof of Theorem 1 is 4 lines of code: interpretability rewrites the conclusion of Theorem 1 to $\mathit{eval}_c(e, \varepsilon) \sim \mathit{compile}(e, \mathit{eval}_s(\hat{\varepsilon}, I))$, and reducibility completes the proof since $\varepsilon \sim \mathit{eval}_s(\hat{\varepsilon}, I)$. Interpretability and reducibility both hold because all SymCert components build terms exclusively through factory functions, and each factory function is individually interpretable and reducible. This enables proving interpretability and reducibility of the compiler (and all other components) by structural induction, with each case bottoming out in a factory function lemma that is proved once and reused everywhere. This modular approach enables lemma reuse across all SymCert components and generalizes to other symbolic compilers (e.g., [23]).

As with the symbolic compiler, we prove correctness of the symbolic authorizer by showing that it behaves equivalently to the Cedar authorizer (Theorem 4). This proof is decomposed into a reducibility and interpretability lemma for the authorizer, which follow from Theorem 2 and Theorem 3.

Theorem 4 (Authorizer correctness)**.** *Let C be Cedar policies, ε a concrete environment, $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$*

a symbolic environment, and I an interpretation, where $\mathit{WellFormed}(\varepsilon, C)$, $\mathit{WellFormed}(\hat{\varepsilon}, C)$, and $\mathit{WellFormed}(I, \hat{\sigma})$. Then, if $\varepsilon \sim \text{eval}_s(\hat{\varepsilon}, I)$, we have $\text{isAuthorized}(C, \varepsilon) \sim \text{eval}_s(\text{authorize}(C, \hat{\varepsilon}), I)$.

C. Enforcer and Extractor Correctness

SymCert-based analyses use the enforcer to constrain the entity hierarchy to be acyclic and transitive, and the extractor to produce counterexamples from SMT models. We prove that the enforced constraints are satisfied by all interpretations corresponding to well-formed environments (soundness), and that the extractor produces well-formed counterexamples from any interpretation satisfying these constraints (completeness). These two theorems are sufficient to prove soundness and completeness of any analysis whose correctness property depends only on policy semantics. All practical policy analyses check such properties (e.g., [2]), including the analyses we present in the next section.

Theorem 5 establishes enforcer soundness: the hierarchy constraints for policies C and symbolic environment $\hat{\varepsilon}$ are satisfied by all well-formed interpretations I that correspond to well-formed concrete environments ε. As a result, the hierarchy constraints cannot cause analyses to miss well-formed counterexamples, preserving soundness. The proof shows that ε binds every entity reference $E{::}s \sim \mathit{eval}_s(t, I)$ obtained by interpreting a term $t \in \mathit{footprints}(C, \hat{\varepsilon})$, so I satisfies the acyclicity and transitivity constraints on t by well-formedness of ε and its equivalence to $\mathit{eval}_s(\hat{\varepsilon}, I)$. The proof relies on Theorem 1 and the interpretability and reducibility of SymCert's factory functions.

Theorem 5 (Enforcer soundness)**.** *Let C be Cedar policies, ε a concrete environment, $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$ a symbolic environment, and I an interpretation, with $\mathit{WellFormed}(\varepsilon, C)$, $\mathit{WellFormed}(\hat{\varepsilon}, C)$, and $\mathit{WellFormed}(I, \hat{\sigma})$. If $\varepsilon \sim \text{eval}_s(\hat{\varepsilon}, I)$, then $\forall t \in \text{enforce}(C, \hat{\varepsilon}), \text{eval}_s(t, I) = \mathsf{true}$.*

Theorem 6 establishes completeness of the enforcer's constraints with respect to counterexamples produced by the extractor. The theorem makes three claims: (1) given an interpretation I that satisfies the enforcer's constraints for policies C and symbolic environment $\hat{\varepsilon}$, the extractor produces a concrete environment ε that is well-formed for C; (2) ε is equivalent to $\hat{\varepsilon}$ under a well-formed interpretation I'; and (3) the translations of policies C have the same meaning under I and I'. Together, these ensure that ε is a well-formed counterexample represented by $\hat{\varepsilon}$, and that policies C behave identically under both $\mathit{eval}_s(\hat{\varepsilon}, I)$ and $\mathit{eval}_s(\hat{\varepsilon}, I')$. For any analysis property that depends only on policy semantics, if the encoded property is violated by I, then it is also violated by I' and its corresponding environment ε. The proof picks I' to be $\mathit{repair}(I, C, \hat{\varepsilon})$ and shows that I' and I agree on the ancestors and attributes of entities in $\mathit{footprints}(C, \hat{\varepsilon})$, using interpretability and reducibility lemmas for SymCert's factory functions.

Theorem 6 (Enforcer and extractor completeness)**.** *Let C be Cedar policies, $\hat{\varepsilon} = \langle \hat{\rho}, \hat{\sigma} \rangle$ a symbolic environment, I an interpretation, and $\varepsilon = \text{extract}(I, C, \hat{\varepsilon})$ a concrete environment. Suppose that $\hat{\varepsilon}$ and I are well-formed, and I satisfies the enforced hierarchy terms: $\mathit{WellFormed}(\hat{\varepsilon}, C)$, $\mathit{WellFormed}(I, \hat{\sigma})$, and $\forall t \in \text{enforce}(C, \hat{\varepsilon}), \text{eval}_s(t, I) = \mathsf{true}$. Then, ε is well-formed, $\mathit{WellFormed}(\varepsilon, C)$, and there exists a well-formed interpretation I', $\mathit{WellFormed}(I', \hat{\sigma})$, such that $\varepsilon \sim \text{eval}_s(\hat{\varepsilon}, I')$ and $\text{eval}_s(\text{compile}(\mathit{toExpr}(c), \hat{\varepsilon}), I) = \text{eval}_s(\text{compile}(\mathit{toExpr}(c), \hat{\varepsilon}), I')$ for every policy $c \in C$.*

Component	Model LOC	Model Hours	Proof LOC	Proof Hours
Terms & factories	1059	81	10018	327
Compiler	318		6520	
Authorizer	16	1	411	11
Enforcer	69	16	4798	167
Extractor	211	13		
Solver (trusted)	566	47	0	0
	2239	158	21747	505

TABLE I: Development metrics for SymCert models and proofs. Lines of code (LOC) are measured using `cloc` [13], and development time is given in person-hours.

D. Soundness and Completeness of the Denies-All Analysis

We formulate soundness and completeness of our denies-all analysis by viewing symbolic environments $\hat{\varepsilon}$ as sets of concrete environments, where $\varepsilon \in \hat{\varepsilon}$ iff $\varepsilon \sim \mathit{eval}_s(\hat{\varepsilon}, I)$ for some well-formed interpretation I. Assuming the trusted *solve* function is correct, Theorem 7 and Theorem 8 ensure that the analysis produces no false negatives or positives with respect to well-formed concrete environments in $\hat{\varepsilon}$. That is, the analysis returns qed if policies C deny all requests in $\hat{\varepsilon}$, or $\mathsf{cex}\,\varepsilon$ if $\varepsilon \in \hat{\varepsilon}$ is well-formed and allowed by C. The theorems follow from SymCert's correctness and the fact that the analyzed property depends only on policy semantics.

Theorem 7 (Denies-all soundness)**.** *Let C be Cedar policies and $\hat{\varepsilon}$ a symbolic environment with $\mathit{WellFormed}(\hat{\varepsilon}, C)$. If the solver is correct and* $\text{verifyDeniesAll}(C, \hat{\varepsilon}) = \mathsf{qed}$*, then for all concrete environments $\varepsilon \in \hat{\varepsilon}$ that are well-formed for C, $\mathit{WellFormed}(\varepsilon, C)$, we have* $\text{isAuthorized}(C, \varepsilon) = \mathsf{false}$.

Theorem 8 (Denies-all completeness)**.** *Let C be Cedar policies and $\hat{\varepsilon}$ a symbolic environment with $\mathit{WellFormed}(\hat{\varepsilon}, C)$. If the solver is correct and* $\text{verifyDeniesAll}(C, \hat{\varepsilon}) = \mathsf{cex}\,\varepsilon$*, then $\varepsilon \in \hat{\varepsilon}$, $\mathit{WellFormed}(\varepsilon, C)$, and* $\text{isAuthorized}(C, \varepsilon) = \mathsf{true}$.

V. Evaluation

This section presents the effort of building SymCert, and evaluates its utility and maintainability. We model and verify five analyses from a Cedar analysis service at AWS, and present our experience extending SymCert to support three new Cedar features.

A. Development Effort

SymCert is implemented in Lean as an extension of the open-source Cedar formalization [11], [15], supporting all

Analysis	Property
verifyDeniesAll$(C, \hat{\varepsilon})$	C denies all $\varepsilon \in \hat{\varepsilon}$
verifyAllowsAll$(C, \hat{\varepsilon})$	C allows all $\varepsilon \in \hat{\varepsilon}$
verifyImplies$(C_1, C_2, \hat{\varepsilon})$	C_2 allows all $\varepsilon \in \hat{\varepsilon}$ allowed by C_1
verifyEquiv$(C_1, C_2, \hat{\varepsilon})$	C_1 and C_2 allow the same $\varepsilon \in \hat{\varepsilon}$
verifyDisjoint$(C_1, C_2, \hat{\varepsilon})$	No $\varepsilon \in \hat{\varepsilon}$ is allowed by both C_1 and C_2

	LOC	Hours
Models for all analyses	39	< 1
Proofs for all analyses	512	7

TABLE II: Development metrics for five Cedar analyses modeled and verified with SymCert.

Benchmark	Policies	Sym-Envs	Queries	Gen. (ms)	Solve (ms)	Time (s)	Findings
DocumentCloud	15	11	448	0	0	7	2
GitHub	9	11	224	0	0	2	0
HotelChains	6	12	198	0	23	3	0
SalesOrgs	10	13	318	0	0	4	2
StreamingService	6	8	117	0	0	2	0
TagsNRoles	2	3	20	18	37	1	0
TaxPreparer	2	1	7	1	25	0	0
TinyTodo	7	9	195	0	24	4	2

TABLE III: Performance of a demo tool built on SymCert's Lean analyses, applied to example Cedar applications [9]. Gen. and Solve report the median time, in milliseconds, to generate and solve an analysis query. Time reports the total time, in seconds, to run all analysis queries for a benchmark. Benchmarks used CVC5 1.1.1 [3] on a 2 GHz Quad-Core Intel Core i5 with 16 GB RAM.

features of Cedar 4.4. Table I summarizes the effort of building SymCert, showing code size and development time for each component model and proof. Code size is measured with `cloc` [13]. Time is given in person-hours, obtained from detailed time tracking logs.

Most effort went into developing the symbolic compiler (81 hours) and its proof (327 hours). The compiler itself is small (318 LOC), but required formalizing SymCert's Term language and proving well-formedness and semantic properties for each of its 42 operators. It also required modeling SymCert's factory functions and proving their well-formedness, interpretability, and reducibility. This supporting infrastructure makes up the majority of the compiler model (77%) and proofs (61%).

The authorizer, enforcer, and extractor reuse core parts of the compiler model and proofs. Collectively, their development took 208 hours versus 408 hours for the compiler—about half the effort. The overall verification effort for SymCert is 5:1 in person-hours spent on proofs versus models.

B. Utility

Table II summarizes our evaluation of SymCert's utility on five analyses from an internal AWS service. The analyses share a common helper function for constructing verification conditions. We implemented all five analyses in less than an hour using 39 lines of code, then proved them sound and complete by establishing soundness and completeness of the shared helper function, as well as interpretability and reducibility of the analyzed properties. The proofs took 7 hours and 512 lines of code. Overall, modeling and proving these analyses required one work day and 551 lines of code.

Our Lean models are efficient enough for prototyping and testing of production code. Service developers used them to prototype a demo of the Cedar analysis service, then ported both SymCert and the analyses to Rust for production deployment. The Rust service now runs in production, while the original Lean implementation serves as a correctness oracle for daily differential random testing.

Table III shows the performance of the demo tool on all example policies and schemas published by Cedar developers [9]. For each benchmark, the tool analyzes all policies across all symbolic environments defined by the schema, checking if any policy always allows or denies, if one policy implies another, or if two policies are equivalent. The resulting analysis queries take under 20 ms to generate and under 40 ms to solve. The tool finds two issues in three benchmarks: unintended policy shadowing and redundancy, demonstrating its ability to detect subtle policy interactions.

RFC	Feature	Cedar +Lines	Cedar Days	SymCert +Lines	SymCert Days
RFC-53	Enumerated entity types	107	3	22	0
RFC-80	Datetime extension type	950	26	2537	8
RFC-82	Entity tags	367	1	1261	4

TABLE IV: Development metrics for extending Cedar and SymCert with three major features. Added lines are measured with `git diff`. Development time is given in person-days.

C. Maintainability

We completed the initial version of SymCert for Cedar version 3.3. Since then, three major features were added via RFC-53 [6], RFC-80 [7], and RFC-82 [8]. Table IV compares the effort of adding these features to SymCert versus the Cedar formalization. We measure incremental coding effort as added lines (via `git diff`, excluding blanks) and development effort in person-days. Four developers contributed these features via pull requests. Time estimates are based on commit activity or logged hours converted to 8-hour days.

RFC-53 [6] adds enumerated entity types to Cedar schemas. Extending the Cedar formalization required updating the typechecker to reject references to undeclared enum entities (107 lines over 3 days). The SymCert extension took just 22 lines and a few minutes by reusing existing support for actions, which are a special case of enumerated entities.

RFC-80 [7] extends Cedar with Datetime and Duration types and 15 functions for operating on these types. Formalizing this required refining the RFC to resolve ambiguities and updating Cedar's evaluator and typechecker models and proofs (950 lines over 26 days). A developer new to Lean extended SymCert to support the RFC in just 8 days. This required

adding 2537 lines to update the symbolic compiler and write interpretability and reducibility lemmas for the new functions.

RFC-82 [8] extends Cedar with entity tags, which cloud services use to attach key-value pairs to entities. Tags are maps from strings to values, accessed via e_1`.hasTag(`e_2`)` and e_1`.getTag(`e_2`)`. Formalizing tags required changing the entity store representation, updating the evaluator and typechecker, and patching soundness proofs (367 lines, 1 day). The SymCert changes updated the symbolic entity store and all components except the authorizer. The symbolic compiler was patched first with interpretability and reducibility proofs for tag functions, which were then reused for the enforcer and extractor. Thanks to this proof reuse, the entire SymCert patch (1261 lines) took just 4 days.

VI. Related Work

SymCert builds on prior work on authorization policy analysis [2], [16], [17] and formal verification of analysis tools [18], [21], [23], [26]. We discuss the most closely related work: Zelkova [2] and Cedar's SMT encoding [12] for the former, and verified symbolic compilers [23] for the latter.

Zelkova is a symbolic compiler for policies written in the Amazon Web Services (AWS) IAM language. It is invoked a billion times a day [24] to answer analysis questions about who has access to AWS resources [1], [5]. Zelkova reduces these questions to SMT-LIB queries consisting of bitvector, string, and regular expression constraints, with key challenges in encoding string constraints [20] and model counting queries [14]. Compared to Cedar, the AWS IAM language is much simpler, lacking nested expressions, entity stores, hierarchies, or user-defined types. Zelkova's encoding is sound but not complete. Unlike Zelkova, SymCert is formally verified to be both sound and complete, and we provide building blocks for showing that analyses built on top of SymCert are also sound and complete. We believe SymCert's verification approach could be applied to Zelkova as well.

The Cedar language has been designed to support sound and complete reduction of policy semantics to a decidable fragment of SMT-LIB. This fragment includes the theory of uninterpreted functions and a subset of the theories of bitvectors, strings, finite sets, and algebraic data types. Prior work [12] outlined symbolic compilation and hierarchy enforcement for a subset of Cedar, with a paper-and-pencil proof of correctness of only the symbolic compiler. However, this work provided no method for mapping the resulting infinite SMT models back to well-formed finite Cedar counterexamples, limiting analysis usefulness and making it impossible to prove analysis completeness. SymCert addresses these gaps by providing fully-featured, formally verified models of Cedar's symbolic compiler and hierarchy enforcer. Most importantly, SymCert provides a novel algorithm for counterexample extraction—a key component missing from prior work that is necessary for both building practical analysis tools and proving their completeness.

SymCert differs from prior work on verifying symbolic compilers [21], [23], [26] in two key ways. First, prior efforts focused on solver-aided languages where analysis correctness follows directly from symbolic compiler correctness. SymCert addresses domain-specific languages like Cedar, where analyses must combine multiple verified components, requiring separate correctness proofs for this composition. Second, prior symbolic compilers could establish correctness theorems using simple bijections between finite source and target states. SymCert must instead define equivalence between finite Cedar entity stores and infinite SMT models to prove compiler correctness, and also extract finite counterexamples from infinite models to prove analysis completeness.

SymCert develops a modular proof approach based on decomposing compiler correctness into two properties: *reducibility* [23] and a novel property we call *interpretability*. Prior work identified reducibility as key to symbolic compiler efficiency and testability, but did not recognize interpretability as a distinct property, even though existing compilers (e.g., [23]) are interpretable. This decomposition avoids the monolithic proofs typical of direct compiler correctness approaches, which we found to produce thousands of lines of unmaintainable proof code. The modular approach enables lemma reuse across SymCert components and generalizes to other symbolic compilers that are both reducible and interpretable.

VII. Conclusion

This paper presented SymCert, a formal framework for modeling and verifying SMT-based analyses of Cedar policies. SymCert provides executable models and correctness theorems for Cedar's symbolic compiler, authorizer, hierarchy enforcer, and counterexample extractor. These components enable rapid development of verified policy analyses with strong guarantees of soundness and completeness. SymCert introduces a modular proof approach that decomposes symbolic compiler correctness into reducibility and interpretability, producing lemmas that are easier to prove, reuse, and extend. We demonstrated SymCert's utility by modeling and verifying five core analyses used in industrial practice, and its maintainability by extending it to support three new Cedar language features with modest incremental effort. SymCert models and proofs are publicly available as part of the Cedar language specification [25].

References

[1] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan. Stratified abstraction of access control policies. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, page 165–176, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53288-8_9.

[2] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9, 2018. doi:10.23919/FMCAD.2018.8602994.

[3] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *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, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.

[4] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://smt-lib.org/, 2016.

[5] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta, C. Schlesinger, C. Stephens, C. Varming, and A. Warfield. Block public access: trust safety verification of access control policies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 281–291, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3409728.

[6] RFC 53: Enumerated entity types. https://github.com/cedar-policy/rfcs/blob/main/text/0053-enum-entities.md, 2024.

[7] RFC 80: Datetime extension. https://github.com/cedar-policy/rfcs/blob/main/text/0080-datetime-extension.md, 2024.

[8] RFC 82: Entity tags with dedicated syntax and semantics. https://github.com/cedar-policy/rfcs/blob/main/text/0082-entity-tags.md, 2024.

[9] Cedar examples repository. https://github.com/cedar-policy/cedar-examples/, 2025.

[10] Cedar language. https://www.cedarpolicy.com/en, 2025.

[11] Cedar source code repository. https://github.com/cedar-policy/, 2025.

[12] J. W. Cutler, C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, E. Ioannidis, J. Kastner, A. Mamat, D. McAdams, M. McCutchen, N. Rungta, E. Torlak, and A. M. Wells. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proc. ACM Program. Lang.*, 8(OOPSLA1), 2024.

[13] A. Danial. cloc, 2025. URL: https://github.com/AlDanial/cloc.

[14] L. D'Antoni, A. Gacek, A. Goel, D. Jovanović, R. G. Kıcı, D. Peebles, N. Rungta, Y. Sharoda, and C. Sung. Projective model counting for ip addresses in access control policies. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pages 208–216, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_26.

[15] C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, J. Kastner, A. Mamat, M. McCutchen, N. Rungta, B. Shah, E. Torlak, and A. Wells. How we built cedar: A verification-guided approach. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 351–357, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3663529.3663854.

[16] W. Eiers, G. Sankaran, A. Li, E. O'Mahony, B. Prince, and T. Bultan. Quantifying permissiveness of access control policies. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1805–1817, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510233.

[17] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 196–205, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1062455.1062502.

[18] S. Keuchel, S. Huyghebaert, G. Lukyanov, and D. Devriese. Verified symbolic execution with Kripke specification monads (and no meta-programming). *Proc. ACM Program. Lang.*, 6(ICFP), Aug. 2022. doi:10.1145/3547628.

[19] K. R. M. Leino. *Program Proofs*. MIT Press, Cambridge, MA, 2023.

[20] K. Lotz, A. Goel, B. Dutertre, B. Kiesl-Reiter, S. Kong, R. Majumdar, and D. Nowotka. Solving string constraints using sat. In *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*, page 187–208, Berlin, Heidelberg, 2023. Springer-Verlag. doi:10.1007/978-3-031-37703-7_9.

[21] S. Lu and R. Bodík. Grisette: Symbolic compilation as a functional programming library. *Proc. ACM Program. Lang.*, 7(POPL), Jan. 2023. doi:10.1145/3571209.

[22] L. d. Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[23] S. Porncharoenwase, L. Nelson, X. Wang, and E. Torlak. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.*, 6(POPL), Jan. 2022. doi:10.1145/3498709.

[24] N. Rungta. A billion smt queries a day (invited paper). In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I*, page 3–18, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-13185-1_1.

[25] Verified SMT-based policy analysis for Cedar, 2025. Models in `SymCC/`, proofs in `Thm/`. URL: https://github.com/cedar-policy/cedar-spec/tree/4720ffd9b8fb77a641662eab556cf245ff1d0dd2/cedar-lean/Cedar.

[26] E. Voogd, Å. A. A. Kløvstad, E. B. Johnsen, and A. Wąsowski. Compositional symbolic execution semantics. *Theoretical Computer Science*, 1044:115263, 2025. doi:10.1016/j.tcs.2025.115263.