A Lean Formalization of Cedar

BHAKTI SHAH∗, University of Chicago, USA

Advisor: Emina Torlak

1 THE CEDAR LANGUAGE

CEDAR [4] is an open source authorization policy language, developed at Amazon Web Services (AWS). CEDAR allows for controlled access to resources via a simple and expressive syntax that supports different authorization paradigms, such as attribute-based access control (ABAC) and role-based access control (RBAC). CEDAR policies define who (the principal) can do what (the action) on what target (the resource) when (the context). Policies have a specified effect: either permit or forbid.

Here is an example of a policy in CEDAR:

```cedar
permit(principal in Group::"admins",
    action in [Action::"create", Action::"delete"],
    resource in Pages::"admin_pages")
unless(principal in Group::"blacklisted_admins");
```

This policy permits a principal with the "admins" role to perform the "create" or "delete" action on resources that have the "admin_pages" role, unless the principal also has the "blacklisted_admins" role (don’t question their system).

CEDAR allows a user to define such a set of policies, and then make an authorization request, which can be either allowed or denied, based on some conditions on the satisfied policies in the policy set. CEDAR is dynamically typed, and type-checking takes place during an optionally run phase called validation.

2 VERIFICATION GUIDED DEVELOPMENT

CEDAR uses a process called verification-guided development (Fig. 1) [2], to ensure the correctness of the authorization engine. The authorizer and validator are modeled in Dafny [3], and using Dafny’s automated reasoning capabilities, a collection of security properties are checked and proved. Via differential random testing (DRT), the production implementation in Rust is checked for equivalence with the Dafny model.

![Fig. 1. Verification Guided Development in Cedar](image-url)

∗Work done while at Amazon Web Services, USA.

Author’s address: Bhakti ShahUniversity of Chicago, Chicago, USA.
3 AN ALTERNATIVE VERIFICATION ENVIRONMENT

Dafny was chosen for the balance it provides between usability and the ability to automatically prove basic properties. However, meta-theoretic properties of Cedar, such as the soundness of the validator, have proved less suitable for Dafny’s automation. To ensure robust performance and minimal maintenance, highly detailed proofs are better, and this favors the use of an interactive theorem prover over an automated one. Hence, we are experimenting with porting the Cedar formalization to the interactive theorem prover Lean [1], to see what the feasibility of this decision would be. We try to answer the question: Can Lean be used for verifying a project at the scale of Cedar, with performance and proof size metrics comparable to the existing formalization in Dafny?

This involves three major steps for both the authorization model and the validation model: formalizing the specification in Lean, verifying key properties of the specification, and setting up differential random testing between the new model and the production implementation. Currently, we have completed about half of these steps: the specifications of both the authorizer and validator have been formalized, and the properties of the authorizer have been verified.

4 LANGUAGE SPECIFICATION

The language specification is made up of the authorization and evaluation models. The authorization model is quite simple: we collect the list of forbid and permit policies satisfied; if there is at least one permit policy satisfied, and no forbid policies are satisfied, then the authorizer allows the request, else it denies it.

```lean
def isAuthorized (req : Request) (entities : Entities) (policies : Policies) :
  Response :=
  let forbids := satisfiedPolicies .forbid policies req entities
  let permits := satisfiedPolicies .permit policies req entities
  if forbids.isEmpty && !permits.isEmpty
    then { decision := .allow, policies := permits }
  else { decision := .deny, policies := forbids }
```

A request is evaluated against each policy in the given policy set. Evaluation can return either true, false, or error.

Each constraint in the policy scope is an expression; members of the context also form expressions. Unconstrained principal, action, or resource clauses evaluate to true.

4.1 Modeling sets

One of the more interesting changes that had to be made was our modeling of sets. Cedar makes heavy use of sets, and in Dafny, sets are axiomatized; in Lean, this is not the case. We had to make a decision about how to model sets, and this was a bit more difficult than a data structure and an equivalence relation.

```lean
inductive Value where
| set (s : Set Value) ...
```

We have a type Value where set : Set Value -> Value, that is, we have a constructor that takes in a set of values as a parameter. To define a quotient type representing sets on Value, we would need to define a function and a type in a mutually recursive fashion, something that is not permitted in Lean. Hence, we had to settle on an alternative definition: a set is a wrapper around a list, but we only deal with well-formed sets, that is, sets where the underlying list is sorted and duplicate free. Utility functions do not assume well-formedness, but we instead prove properties for each utility function that state that a well-formed input produces a well-formed output.
4.2 Theorems proven
We prove the following theorems for the authorizer:

- If some forbid policy is satisfied, then the request is denied.
- A request is allowed only if it is explicitly permitted (i.e., there is at least one permit policy that is satisfied).
- If not explicitly permitted, a request is denied.
- Authorization produces the same result regardless of policy evaluation order or duplicates.

A trade-off we had to consider here was the size of the proofs: as Dafny is able to discharge proof obligations to an SMT solver, just stating the lemma was enough for the language to succeed at verification. However, with Lean, all proofs needed some amount of manual intervention: the proof term must be manually constructed, and hence Lean proofs were often longer than Dafny proofs, that could be empty.

5 VALIDATION MODEL
In the existing Dafny formalization, Cedar validation followed a slightly complex model known as permissive validation. During the Lean formalization, we decided to change the validation model to be stricter, by simplifying the type system quite a bit. The only non-standard part of the type system is to do with booleans: when we are able to make certain judgements, we type boolean values strongly with \( \texttt{tt} \) and \( \texttt{ff} \) representing the true and false types, over and above the regular anyBool type which corresponds to the more familiar boolean type.

```lean
inductive BoolType where
| anyBool
| tt
| ff
```

The subtyping relation is also simple. Records have width subtyping, but not depth subtyping: \( \texttt{tt} <: \texttt{anyBool} \) & \( \texttt{ff} <: \texttt{anyBool} \); every type is a subtype of itself.

6 STATISTICS
As one of the concerns was an increase in sizes of proofs, we collected some data with respect to lines of code. Surprisingly, the Lean specification outperformed the Dafny specification in almost all regards (except for the set utilities, which were inbuilt in Dafny).

<table>
<thead>
<tr>
<th>Element</th>
<th>Dafny LOC</th>
<th>Lean LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic datatype definitions</td>
<td>0</td>
<td>738</td>
</tr>
<tr>
<td>Language model specification</td>
<td>1687</td>
<td>936</td>
</tr>
<tr>
<td>Validation model specification</td>
<td>1109</td>
<td>460</td>
</tr>
<tr>
<td>Language model theorems &amp; proofs</td>
<td>394</td>
<td>354</td>
</tr>
</tbody>
</table>

We also compared the verification times for the two specifications. Averaged over 5 runs, we found Lean’s average verification time to be 61.79s and Dafny’s average verification time to be 115.12s: suggesting an almost 2x speedup.

7 CONCLUSIONS & FUTURE WORK
Our experience pointed towards conclusive evidence about the usability of Lean as an alternative verification tool for Cedar. Further, this is the first software verification project of this scale in Lean, and provides a promising result for the future of the same.

The future work intends to complete proofs of soundness for the validator, as well as formally set up differential random testing between the model and the production implementation. We hope this is a step towards easier maintenance and higher readability of the Cedar model.
REFERENCES


