How We Built Cedar: A Verification-Guided Approach

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

John Kastner  
Amazon Web Services  
USA

Anwar Mamat  
University of Maryland  
USA

Matt McCutchen  
Unaffiliated  
USA

Neha Rungta  
Amazon Web Services  
USA

Bhakti Shah  
University of Chicago  
USA

Emina Torlak  
Amazon Web Services  
USA

Andrew Wells  
Amazon Web Services  
USA

ABSTRACT

This paper presents verification-guided development (VGD), a software engineering process we used to build Cedar, a new policy language for expressive, fast, safe, and analyzable authorization. Developing a system with VGD involves writing an executable model of the system and mechanically proving properties about the model; writing production code for the system and using differential random testing (DRT) to check that the production code matches the model; and using property-based testing (PBT) to check properties of unmodeled parts of the production code. Using VGD for Cedar, we can build fast, idiomatic production code, prove our model correct, and find and fix subtle implementation bugs that evade code reviews and unit testing. While carrying out proofs, we found and fixed 4 bugs in Cedar’s policy validator, and DRT and PBT helped us find and fix 21 additional bugs in various parts of Cedar.

CCS CONCEPTS

• Security and privacy → Formal methods and theory of security; Authorization; • Software and its engineering → Software development techniques.

KEYWORDS

formal methods, fuzz testing, differential testing, policy authorization language, interactive theorem proving

ACM Reference Format:


1 INTRODUCTION

Cedar [7, 11] is a new, open-source authorization policy language. Developers express permissions for their applications as policies written in Cedar. When the application needs to perform a sensitive user operation, it sends a request to the Cedar authorization engine, which allows or denies the request by evaluating the relevant policies. Because Cedar policies are separate from application code, they can be independently authored, updated, analyzed, and audited.

Cedar’s authorization engine is part of an application’s trusted computing base (TCB), which comprises components that are critical to the application’s security. To provide assurance that the engine’s authorization decisions are correct, we develop Cedar using a two-part process we call verification-guided development (VGD). First, we construct simple and readable formal models of Cedar’s components. We write these models in the Lean programming language [31], and carry out mechanized proofs to show that they satisfy important correctness properties. Second, we use differential random testing (DRT) [29] to show that the models match the production code, which is written in Rust. DRT involves generating millions of inputs—consisting of policies, data, and requests—and...
testing that the model and production code agree on the outputs. We also perform property-based testing (PBT) of the Rust code directly (in the style of QuickCheck [8]) when there is no corresponding Lean model. We may also property-test conjectured properties on the Rust code before we prove them in Lean.

VGD provides a practical balance of competing concerns around high assurance, ease of development, and maintainability. To see why, consider two other processes we might have followed.

One approach would be to develop Cedar entirely in Lean, compile to C, and deploy the generated C code in production. In addition to the formal model, we would write an optimized and full-featured implementation (which handles errors and messages more carefully, provides parsers for various formats, etc.) in Lean, and then prove the two equivalent. A key benefit of this approach is that we would formally verify—rather than just test—the equivalence of the deployed code and model. But there are significant downsides to writing production applications in Lean and deploying its generated C code. Lean is a new programming language, so it has a limited developer pool and lacks useful libraries available in mainstream languages. Debugging a failure might necessitate stepping through the generated C code and mapping it back to the Lean source, requiring expertise in C and Lean. Doing so could be particularly difficult if the failure is due to an interaction between handwritten code and Lean-generated C code. In addition, C is a memory-unsafe language, so a bug in the Lean compiler could lead to security issues.

Another approach would be to develop only in Rust and prove correctness of the Rust code directly, using a tool like Aeneas [19], Kani [23], Prusti [1], Creusot [12], or Verus [25]. The main challenge here is that tools for verifying Rust code are not (yet) up to the task, e.g., they cannot handle much of the standard library, they support only certain code idioms, they sometimes have trouble scaling, and they are limited in the properties one can specify. The model consumable by one of these tools is unlikely to be as readable as the Lean model, due to Rust’s low-level nature. These limitations present challenges to guaranteeing high assurance, but also to ease of development and maintainability, because teams would likely need to wrangle code into less idiomatic forms with each release just to enable proofs.

Using VGD has proved beneficial for Cedar. It has helped us improve Cedar’s design: While implementing the formal model and carrying out proofs of soundness of the Cedar policy validator, we found and fixed four bugs. It has also allowed us to write fast, idiomatic Rust code with increased confidence in its correctness: Using DRT and PBT we have so far found and fixed 21 bugs in various Cedar components.

We believe VGD represents a practical approach to leveraging the benefits of formal methods while also assuring the deployed code is easy to use, develop, and maintain. The remainder of this paper presents our experience with VGD for Cedar in detail, beginning with background on Cedar (Section 2), our Lean models of and proofs about Cedar’s components (Section 3), how we use DRT and PBT on our Rust code (Section 4), and how VGD compares to related work (Section 5).

Cedar is open source. Our Lean models, Rust implementation, and testing setup are all available at https://github.com/cedar-policy.

// Policy 1
permit(principal, action, resource)
  when {
    resource.has.owner &&
    resource.owner == principal
  };

// Policy 2
permit(
  principal,
  action == Action::"GetList",
  resource)
  when {
    principal in resource.readers ||
    principal in resource.editors
  };

// Policy 3
forbid (
  principal in Team::"interns",
  action == Action::"CreateList",
  resource == Application::"TinyTodo"
);

Figure 1: Cedar policies for TinyTodo

2 THE CEDAR POLICY LANGUAGE

Cedar is a language for writing authorization policies, supporting idioms in the style of role-based access control (RBAC) [14], attribute-based access control (ABAC) [20], and their combination. Cedar policies use a syntax resembling natural language to define who (the principal) can do what (the action) on which target (the resource) under what conditions. To see how Cedar works, consider a simple application, TinyTodo [38], designed for managing task lists. TinyTodo uses Cedar to control who can do what.

Figure 1 shows three sample policies from TinyTodo. The first is an ABAC-style policy that allows any principal (a TinyTodo user) to perform any action on a resource (a TinyTodo list) they own, as defined by the resource’s owner attribute matching the requesting principal. The second policy allows a principal to read the contents of a task list (Action::"GetList") if the principal is in the list’s readers or editors group. The third is an RBAC-style policy that forbids interns (a role) from creating a new task list (Action::"CreateList") using TinyTodo (Application::"TinyTodo").

When the application needs to enforce access, as when a user of TinyTodo issues a command, it makes a corresponding request to the Cedar authorizer. The authorizer invokes the Cedar evaluator for each policy, to see whether it is satisfied by the request (and provided application data). The authorizer returns decision Deny if no permit policy is satisfied or if any forbid policy is satisfied. It returns Allow if at least one permit policy is satisfied and no forbid policies are. Cedar supports indexing policies so that they can quickly sliced to a subset relevant to a particular request. For example, if a given request does not have Application::"TinyTodo" as its resource, then Policy 3 is not included in the slice.

Principals, resources, and actions are Cedar entities. Entities are collected in an entity store and referenced by a unique identifier consisting of the entity type and entity ID, separated by ":": 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. Expression $A$ in $B$ evaluates to true if $B$ is a (transitive) parent of $A$.

Cedar policies have three components: the effect, the scope, and the conditions. The effect is either permit or forbid, indicating whether the policy is granting or removing access. The scope comes after the effect, constraining the principal, action, and resource components of a request; policy indexing is based on scope constraints. The conditions are optional expressions that come last, adding further constraints, oftentimes based on attributes of request components. Policies access request components using the variables principal, action, resource, and context.

Policy evaluation could result in a dynamic type error, e.g., if a when expression tries to access resource $pwner$ but resource has no $pwner$ attribute, or tries to use a numeric operation like $<$ on a pair of entities. When this happens, the erring policy does not factor into the final authorization decision—it is ignored. Users can avoid this situation by using the Cedar validator to statically check their policies against a schema, which defines the names, shapes, and parent relations of entity types, as well as the legal actions upon them. If the validator is satisfied, users can be sure that when requests conform to the schema, their policies’ evaluation will never result in run-time type errors.

To help users understand the meaning of their policies, we designed Cedar to be amenable to automated reasoning via an encoding of its semantics into formal logic. The Cedar symbolic compiler translates Cedar policies to SMT-lib [2], the language of SMT solvers, producing an encoding that is sound, complete, and decidable. With this compiler we can, for example, prove that two sets of policies are equivalent, meaning that they authorize exactly the same requests in the same entity stores. To do this, we encode each policy set as a formula and ask the SMT solver to search for a request and entity store that is allowed by one policy set but not the other. A response of UNSAT guarantees that the policy sets are equivalent.

3 LEAN MODELS AND PROOFS

The first part of the verification-guided development process is constructing an executable, formal model of the system, using a proof-oriented language. We wrote models of the Cedar evaluator, authorizer, and validator in Lean [31], and are in the process of writing one for the symbolic compiler. The models serve as Cedar’s specification, and we had two goals when writing them. First, the models should be human readable, and thus favor simplicity and understandability (Section 3.1). Second, they should be as feature complete as possible, so that proofs about them (Section 3.2) apply to the full language, not just an abstraction, and so that the models can be used as oracles for differential testing (Section 4).

3.1 A human-readable specification

Lean allows us to write concise specifications. As an illustration, here’s the Lean code for the Cedar authorizer:

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

Table 1 shows the size of the Lean specification and proofs against the corresponding Rust implementation and tests, measured in lines of code (LOC). The Lean models are an order of magnitude smaller than their Rust counterparts, which include optimizations, extra code to provide useful diagnostic output, and some unmodeled code. For example, our parsers are not modeled in Lean because there is currently no library support for parser generators, and it is unclear what properties we could prove about a parser model. We also did not port our unit tests to Lean because we can instead prove properties of interest.

3.2 Proofs of properties

We used Lean to formalize and prove several properties of our Cedar models, listed below.

1. **Forbid trumps permit**: If any forbid policy is satisfied, the request is denied.
2. **Default deny**: If no permit policy is satisfied, the request is denied.
3. **Explicit allow**: If a request is allowed, some permit policy was satisfied.
Throughout development, we benefited from Lean’s extensive library of theorems and its IDE extension, which checks proofs interactively and provides instant feedback. Our Lean proofs are fast to generate, with 6 microseconds compared to 10 microseconds for Rust.

The differential and other properties we test are given in Table 2. Some properties have multiple testers with different generators. For example, the property Rust and Lean authorizer parity is tested by two different policy generation strategies—ABAC and RBAC. The

4 DIFFERENTIAL RANDOM TESTING
The second part of the verification-guided development process is to use differential random testing [29] to increase our confidence that the behavior of our formal models matches that of our production code. Figure 2 shows the workflow of DRT. Using the cargo-fuzz fuzz testing framework [6], we randomly generate millions of inputs—access requests, entities, and policies—and send them to both the Lean model and the corresponding Rust production implementation. If the two versions agree on the output, then we obtain a piece of evidence that Rust production code is on par with the Lean model. If they disagree, then we have found a bug (e.g., production code incorrectly implements the specification).

With each version of Cedar, we save a minimized set of corpus tests generated by cargo-fuzz to use as part of continuous integration testing (Section 4.3). Using the same cargo-fuzz framework, we also perform property-based testing in the style of QuickCheck [8] to directly check properties of our production components. PBT is complementary to DRT because it allows us to test properties we have yet to prove in Lean, and those of production components (such as the Cedar policy parser) for which no model exists.

This section discusses how we generate useful random inputs (Section 4.1), what properties we test (Section 4.2), and what we have learned in the process (Section 4.3).

4.1 Input generation
To use DRT effectively requires generating inputs that thoroughly exercise the targeted code. Naïve input generation is insufficient: If we randomly generate a policy, entity store, and request independently, policies are likely to be ill-typed and to reference non-existent entities or attributes. This would over-exercise error-handling code and fail to cover much of a component’s core logic. We tackle this challenge by correlating the generation of policies, entity stores, and requests.

Specifically, our typical input generation strategy is type directed in the style of Palka et al. [33]: we first generate a schema, then an entity store that conforms to the schema, and then policies and requests that access those entities in conformance with the schema. This approach assures we target the core logic. But it has the problem that well-typed policies do not exercise error handling code. Therefore we developed another generator whose policies always refer to entities, actions, and attributes in the schema, but whose conditions can be ill-typed. Because cargo-fuzz is a coverage-guided testing tool, leveraging llibfuzzer [27] to bias input generation toward unexecuted code, we hoped this generator would gravitate toward unexecuted conditions, too, and we could lean solely on it for input generation. However, our experience proved otherwise. The type-directed generator produces more complicated inputs, like set operations, than the non type-directed version, leading to deeper coverage of core logic and faster bug discovery. So we use both.

4.2 Properties to test
The differential and other properties we test are given in Table 2. Some properties have multiple testers with different generators. For example, the property Rust and Lean authorizer parity is tested by two different policy generation strategies—ABAC and RBAC.
formers a single ABAC-style policy for each run and mainly targets the evaluator since generated policies can have nontrivial conditions. The latter generates multiple RBAC-style policies for each run and aims to exercise the authorization logic that makes a decision involving more than one policy.

### 4.3 Experience

We use Amazon’s Elastic Container Service (ECS) to test our properties daily on currently supported Cedar versions. We allocate 4096 CPU units (4 vCPUs) and 8GB memory to fuzz each target for 6 hours. This setup generates millions of inputs for most targets. We do not have a particular reason for the 6 hour duration, although it ensures that block coverage for each target saturates or grows slowly near the end of a daily run. We plan to investigate approaches to choosing an optimal fuzzing duration in the future.

As shown in Table 2, differential and property testing have uncovered 21 bugs in total. As one example, differential testing against the Lean model helped us find a bug in a Rust package we used for parsing IP addresses. This finding eventually motivated us to write our own IP address parser, replacing the buggy external package. Some bugs found by differential testing even affected Cedar’s language design. For example, an unreleased early version of Cedar provided a method to get the size of a string. The model and production code ended up having different implementations (using bytes vs. codepoints vs. graphemes), causing DRT to fail. We eventually dropped this feature after agreeing that it confuses users more than it benefits them. Property testing helped us uncover subtle bugs in the Cedar policy parser, in how the formatter handles comments, and how namespace prefixes on application data (e.g., `TinyTodo::List` is interpreted. All bugs found by the validation soundness property were found before we had completed a soundness proof.

Complete line coverage alone does not guarantee effective testing [4]. The same lines of code executed with different program state can lead to very different outcomes. Therefore it is important to test the same or similar code paths with diverse inputs, and to focus on paths of importance.

For Cedar DRT, we found that even when the ABAC policy generator achieves full line coverage, many of the generated inputs exercise error handling code. Furthermore, 35.5% of the generated condition-expressions generated by the type-directed ABAC policy generator, for a typical DRT run, are boolean literals, which means that one third of policy conditions are trivially true or false and do not trigger interesting code paths. To address these limitations, we added a new input generator that generates expressions of various types (as opposed to just those of Boolean type, as produced by the ABAC policy generators). For the expression generator, only 9.7% of the generated condition-expressions contain literals, but not Boolean ones—the majority are strings (e.g., as part of `==` expressions) whose evaluation leads to more code coverage because string handling logic (e.g., unescaping raw strings) is non-trivial and error-prone.

Integration tests made from corpus tests turn out to be a valuable asset for Cedar developers: they helped us quickly catch tricky bugs when developing new features. For instance, they contain subtle wildcard patterns and Cedar values that exposed bugs in the prototypes of wildcard matching and schema-based parsing. Integration tests allowed us to fix these bugs before pushing the code, avoiding the delay for the daily DRT run to uncover them.

**Limitations.** Unsurprisingly, Cedar’s testing framework has missed bugs, too. The most notable example is that it did not discover the non-termination bug described in Section 3.1. The reason for this is that the probability of generating inputs triggering this bug is extremely low. Our framework is also inherently unable to find bugs outside the scope of the test generators. For example, we did not discover some parser bugs triggered by malformed policies because our test generators create abstract syntax trees of Cedar policies and thus are limited to produce only syntactically correct policies. We are investigating grammar-based mutation testing [3] to avoid missing this type of bug in the future.

Appendix A enumerates all the bugs found, and several missed, by DRT and PBT.

<table>
<thead>
<tr>
<th>Property</th>
<th>Input Generator</th>
<th>Description</th>
<th># bugs found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust and Lean authorizer parity</td>
<td>ABAC (type-directed)</td>
<td>Differentially test production authorizer against its Lean model using a single ABAC policy with a mostly well-typed condition</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>ABAC</td>
<td>Differentially test production authorizer against its Lean model using a single ABAC policy with an arbitrary condition</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RBAC</td>
<td>Differentially test production authorizer against its Lean model using multiple RBAC (condition-free) policies</td>
<td>0</td>
</tr>
<tr>
<td>Rust and Lean validator parity</td>
<td>ABAC (type-directed)</td>
<td>Differentially test production validator against its Lean model</td>
<td>4</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>ABAC</td>
<td>Test that the composition of pretty-printing and parsing produces a policy identical to the one generated</td>
<td>6</td>
</tr>
<tr>
<td>Formatter roundtrip</td>
<td>ABAC</td>
<td>Test that the composition of pretty-printing, formatting, and parsing produces a policy identical to the one generated</td>
<td>2</td>
</tr>
<tr>
<td>Parser safety</td>
<td>Random bytes</td>
<td>Simply fuzz the Cedar parser with arbitrary inputs</td>
<td>0</td>
</tr>
<tr>
<td>Validation soundness</td>
<td>ABAC (type-directed)</td>
<td>Test that evaluating policies that validate does not produce type errors</td>
<td>3</td>
</tr>
</tbody>
</table>
5 RELATED WORK

Various communities have developed tools to increase software dependability. The automotive industry follows the MISRA standards [30]. However, these standards aim to avoid common pitfalls rather than guarantee functional correctness. The FAA requires critical aerospace software have MCDC coverage [24]. This addresses functional correctness but does not guarantee it. The highest assurance can be gained by formally verifying that deployed software meets a specification and that the specification has particular safety/security properties. Alternatively, rather than prove software against a specification, we could develop an executable model for the specification and differentially test the code against the model, and likewise test that the model has certain properties. Verification-guided development offers a pleasant compromise: We prove properties about a readable formal model, and rigorously test that the deployed code matches that model.

**Formal methods.** A variety of software, including the CompCert optimizing C compiler [26], the SeL4 microkernel [18], and the Ev- erCrypt cryptography library [34], have been formally verified and deployed in practice. These systems employ the formal verification frameworks Coq [10], Isabelle/HOL [32], and F* [16], respectively, which center around a domain-specific proof-oriented programming language (PPL). The code from the PPL is either mapped to/from code written in a mainstream language, like C or OCaml, or directly extracted (compiled) to it.

Lean is similar to Coq, and in principle we could have deployed our formally verified Lean model of Cedar as extracted C code, rather than building a separate Rust version. However, as discussed in Section 1, such extracted code would be challenging to maintain and operate, e.g., when debugging broader system failures in deployment, because it is not intended to be readable. To address this issue, EverCrypt deploys readable C code using a purpose-built idiomatic compiler, KaRaMeL [35], which works on the Low* subset of F*.

Even with such a compiler, developing industrial-strength code in Lean (or indeed, any PPL) is challenging because of its limited library support and limited base of developer expertise, compared to a mainstream language. Alternatively, one could try to formally verify a software system written in a language like Java (e.g., using tools like OpenJML [9] or Krakatoa [15]), or Rust (e.g., using tools such as Aeneas [19], Kani [23], Prusti [1], Creusot [12], or Verus [25]). However, these tools have limitations in scope, scalability, and tooling that prevent their use on an industrial scale. As Lean and these other tools develop, the tradeoffs may change.

**Differential and property-based random testing.** Perhaps the best-known example of differential testing is the Csmith tool developed by Yang et al. [39], which tests C/C++ compilers against each other on randomly generated programs, looking for discrepancies in their results. Other examples include Bornholt et al. [5], who apply DRT to AWS S3’s ShardStore, writing a simple model in Rust that serves as a test oracle and applying stateless model checking to prove properties of this simplified code. Groce et al. [17] use DRT as a precursor to formal methods, but they focus on correctness in the presence of hardware faults. SybilFS [37] proposes a reference model of a POSIX file system that other implementations can use as an oracle for differential testing.

QuickCheck [8] introduced property-based random testing, testing that a property holds on automatically-generated inputs, rather than on a few hand-defined ones. Property testing is used in the S3 ShardStore paper mentioned above [5], in addition to model checking. Hughes et al. [21] apply property testing to the distributed file systems Dropbox, Google Drive and ownCloud (an opensource equivalent) and found several bugs. Property testing enjoys moderate popularity, e.g., the Hypothesis Python library [28] has more than 200,000 downloads per day as of January, 2024. Defining a property and randomly testing it blurs the bounds between traditional testing and formal methods, and has been identified as a promising onramp to the use of formal methods [36].

**Dependability cases.** A dependability case, as proposed by Jackson [22], is a careful collection of different sorts of evidence showing that a software system is correct. Verification-guided development could be used to produce a dependability case: (1) evidence for good design is in the form of mechanized proofs of properties of the model, and (2) evidence of correct implementation is in the form of differential and property tests of the deployed code. Ernst et al. [13] report that constructing a dependability case can lead to a clearer view of what assumptions underlie formal modeling and testing, which helps identify gaps to be shored up with further testing, proofs or other techniques.

6 CONCLUSION

This paper presented verification-guided development (VGD), a high-assurance engineering process that we use to develop the Cedar authorization language and tools. The process has two parts.

1. We write a readable, executable model of Cedar in Lean and prove that the model satisfies key correctness and security properties. Our proof effort leverages Lean’s extensive theorem libraries, interactive IDE support, and fast verification.
2. We use differential random testing (DRT) to check that the Cedar production code, written in Rust, matches the model, and use property-based testing (PBT) to test properties against the production code for which there is no analogue in the model (or no proof, yet). Our DRT/PBT input generators are carefully crafted to achieve good code coverage and balanced input distributions.

Both proofs and DRT helped us to uncover and fix subtle bugs in various Cedar components prior to release. Our experience shows that VGD is a practical approach for developing high-assurance code: it leverages the benefits of formal methods while producing code that is easy to use, develop, and maintain.

Cedar is open source: The Lean models, Rust code, and testing setup are all available at https://github.com/cedar-policy.

REFERENCES

Table 3: Bugs found by DRT/PBT

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust and Lean authorizer parity (ABAC)</td>
<td>The Rust implementation and definitional model computed string sizes differently</td>
</tr>
<tr>
<td>Rust and Lean authorizer parity (ABAC type-directed)</td>
<td>The Rust implementation and definitional model named extension functions differently (e.g., <code>lt</code> vs. <code>lessThan</code>)</td>
</tr>
<tr>
<td>Rust and Lean authorizer parity (ABAC type-directed)</td>
<td>The Rust implementation allowed octal numbers in IPv4 addresses while the definitional model did not</td>
</tr>
<tr>
<td>Rust and Lean authorizer parity (ABAC type-directed)</td>
<td>The definitional model incorrectly rejected <code>::</code> when parsing IPv6 addresses</td>
</tr>
<tr>
<td>Rust and Lean authorizer parity (ABAC type-directed)</td>
<td>The Rust implementation supported embedding IPv4 addresses in IPv6 addresses while the definitional model did not</td>
</tr>
<tr>
<td>Rust and Lean authorizer parity (ABAC type-directed)</td>
<td>The Rust implementation required the operands of the <code>isInRange</code> extension function to be both IPv4 or IPv6 addresses while the definitional model did not</td>
</tr>
<tr>
<td>Formatter roundtrip</td>
<td>Comments on records were dropped by the formatter (<a href="https://github.com/cedar-policy/cedar/pull/257">https://github.com/cedar-policy/cedar/pull/257</a>)</td>
</tr>
<tr>
<td>Formatter roundtrip</td>
<td>Comments could be dropped in an <code>is</code> expression (<a href="https://github.com/cedar-policy/cedar/pull/460">https://github.com/cedar-policy/cedar/pull/460</a>)</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>The parser did not unescape raw strings</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>The parser did not parse the pattern literals of the <code>like</code> operation correctly</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>The parser did not parse namespaced extension function names correctly</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>The parser performed constant folding incorrectly</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>Pretty-printing did not consistently add parentheses to negation operations</td>
</tr>
<tr>
<td>Parser roundtrip</td>
<td>Pretty-printing certain function call ASTs resulted in a crash</td>
</tr>
<tr>
<td>Rust and Lean validator parity</td>
<td>The Rust implementation and definitional model named extension functions differently (e.g., <code>ipaddr</code> vs. <code>IPAddr</code>)</td>
</tr>
<tr>
<td>Rust and Lean validator parity</td>
<td>The Rust implementation incorrectly rejected certain policies with schemas containing unspecified entity types</td>
</tr>
<tr>
<td>Rust and Lean validator parity</td>
<td>The Rust implementation incorrectly handled policies containing certain types of records (<a href="https://github.com/cedar-policy/cedar/pull/165">https://github.com/cedar-policy/cedar/pull/165</a>)</td>
</tr>
<tr>
<td>Rust and Lean validator parity</td>
<td>The Rust implementation and definitional model disagreed on the type of <code>resource in [ ]</code> when <code>resource</code> has unspecified entity type (<a href="https://github.com/cedar-policy/cedar/pull/615">https://github.com/cedar-policy/cedar/pull/615</a>)</td>
</tr>
<tr>
<td>Validation soundness</td>
<td>The validator ignored certain entity type namespaces</td>
</tr>
<tr>
<td>Validation soundness</td>
<td>The validator did not parse extension function call arguments correctly</td>
</tr>
<tr>
<td>Validation soundness</td>
<td>The validator did not correctly typecheck certain has expressions</td>
</tr>
</tbody>
</table>

A TROPHY AND ANTI-TROPHY CASES

In this appendix, we list trophy (bugs found) and anti-trophy (bugs missed) cases of DRT and PBT, all of which have been promptly fixed by Cedar developers. Table 3 lists the bugs found by Cedar’s DRT/PBT. We annotate violations of the property Rust and Lean authorizer parity with the generators used to find them. Table 4 lists missed bugs by Cedar’s DRT/PBT. We include links to the relevant pull requests when possible. Items without links were fixed on earlier versions of Cedar, prior to open-sourcing.

Received 2024-02-08; accepted 2024-04-18
### Table 4: Bugs missed by DRT/PBT

<table>
<thead>
<tr>
<th>Description</th>
<th>Component</th>
<th>Root Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Rust evaluator incorrectly implemented the <code>in</code> operation</td>
<td>Evaluator</td>
<td>DRT failed to generate inputs that trigger this bug</td>
</tr>
<tr>
<td>The Rust evaluator accepted the string representation of an invalid decimal literal</td>
<td>Evaluator</td>
<td>Triggering input is too hard to generate</td>
</tr>
<tr>
<td>The parser crashed on certain malformed policies</td>
<td>Policy parser</td>
<td>DRT does not methodically generate malformed policies</td>
</tr>
<tr>
<td>The parser did not reject certain malformed policies (<a href="https://github.com/cedar-policy/cedar/pull/594">https://github.com/cedar-policy/cedar/pull/594</a>)</td>
<td>Policy parser</td>
<td>DRT does not methodically generate malformed policies</td>
</tr>
<tr>
<td>The API to link a policy to a template could crash on invalid inputs (<a href="https://github.com/cedar-policy/cedar/pull/203">https://github.com/cedar-policy/cedar/pull/203</a>)</td>
<td>Public API</td>
<td>DRT did not test the relevant APIs</td>
</tr>
<tr>
<td>Certain parsable policies could fail to be converted to their JSON representation (<a href="https://github.com/cedar-policy/cedar/pull/601">https://github.com/cedar-policy/cedar/pull/601</a>)</td>
<td>Public API</td>
<td>DRT did not test the relevant APIs</td>
</tr>
<tr>
<td>The JSON schema parser accepted inputs with unknown attributes</td>
<td>Schema parser</td>
<td>DRT does not test malformed schemas</td>
</tr>
<tr>
<td>The validator did not terminate on certain inputs</td>
<td>Validator</td>
<td>Triggering input is too hard to generate</td>
</tr>
<tr>
<td>The validator did not typecheck template-linked policies correctly (<a href="https://github.com/cedar-policy/cedar/pull/371">https://github.com/cedar-policy/cedar/pull/371</a>)</td>
<td>Validator</td>
<td>DRT did not test the relevant APIs</td>
</tr>
</tbody>
</table>