Abstract of “Software Verification Techniques for Analyzing Access Control Policies”
by Jakob Louis Kreuze, Brown University, February, 2024
Formal verification is an increasingly prominent approach to ensuring the reliability of software intended for high-assurance domains such as aviation and hardware design. One particular instance of verification is the use of interactive theorem provers to provide formal proofs of functional correctness for software systems. In this paper, we use the Lean interactive theorem prover to formally state and verify properties about access control policies specified in either the eXtensible Access Control Markup Language (XACML) 3.0 [3] or the Rego language used by Open Policy Agent (OPA) [1], and we demonstrate how the tool can be used to prove equivalence between policies in each language. Furthermore, we explore the possibility of using Lean to develop high-performance “correct-by-construction” policy decision engines.
AUTHORIZATION TO LEND AND REPRODUCE THE THESIS

As the sole author of this thesis, I authorize Brown University to lend it to other institutions or individuals for the purpose of scholarly research.

Date December 20th, 2023

Jakob L. Kreuze, Author

I further authorize Brown University to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Date December 20th, 2023

Jakob L. Kreuze, Author
SOFTWARE VERIFICATION TECHNIQUES FOR ANALYZING ACCESS CONTROL POLICIES

by Jakob Louis Kreuze
B.Sc., University of Massachusetts, 2021

A thesis submitted in partial fulfillment of the requirements for the Master of Science in the Department of Computer Science at Brown University

Providence, Rhode Island
February 2024
Acknowledgments

I would to thank the United States Air Force for funding my attendance at Brown through the PALACE Acquire program, as well as my former mentor, Lieutenant Colonel Kyle Stewart, for introducing me to this problem space and greatly influencing my decision to study software verification.

I am thankful to my advisors Rob Lewis and Shriram Krishnamurthi for the many ways they have supported this project. Rob is an excellent teacher of Lean and OCaml – I certainly could not have completed this project without him as a resource. Shriram, having authored Margrave, served as an exceptional resource for navigating the literature on policy analysis, and generally providing direction to this project. The “standard implementation plan” taught in his course on programming languages was also extraordinarily helpful in engineering the various compilers which I developed over the course of this project.

I wish also to thank my partner, Oli Finley, for their emotional support. Pursuing a graduate degree is a stressful endeavor, and I am not sure I could have done it without having someone to remind me to occasionally take a break, and to take care of myself.

Finally, I cannot neglect to thank my parents: Corrina Kreuze and Michael Kreuze. I owe it to their nurturing and support that I had the grit to pursue an advanced degree in the first place. It is also due to my father that I had a leg up in typesetting this manuscript: he introduced me to GNU Emacs at the age of 11, and to \LaTeX{} in the later years of high school. I am grateful for their encouragement when I demonstrated an interest in computers and mathematics; I think it is unlikely that I would have attended university for computer science had they not helped me to acquire material for self-study in my youth.
Contents

1 Introduction 1

1.1 Summary of Contributions 2

1.2 Availability of Project Materials 2

2 An Embedding of eXtensible Access Control Markup Language 3

2.1 XACML Concepts And Their Representation in Lean 3

2.1.1 On Computability 5

2.2 Side Information and Error Handling 6

2.3 Identifier Equality 6

2.4 Concepts Modeled but Absent in the Embedding 7

2.5 Justifying our Embedding 7

2.5.1 Evaluation Tables 9

2.6 Assumptions and Limitations 11

2.7 Future Work: Evaluation Against Conformance Test Suite 12

2.8 Future Work: Verified Compilation 12

2.9 Related Work 13

3 An Embedding of the Rego Language 14

3.1 Justifying our Embedding 14

3.2 Rego Concepts 14

3.2.1 Compilation of Rules with Reference Heads 16

3.2.2 Scops 17

3.2.3 Rules With Formal Parameters 18

3.3 Summary 19

3.4 Dependency 19

3.5 Negation 20

3.6 Comparison 20
List of Tables

1. Omitted XACML 3.0 elements ................................................. 8
3. Statistics of wall clock times for evaluating IIA001 across all three implementations ................. 30
4. Statistics of wall clock times for evaluating IIA003 across all three implementations ................. 31
5. Statistics of wall clock times for evaluating IIA006 across all three implementations ................. 31
List of Figures

1. Whisker plot of wall clock times for evaluating IIA001 across all three implementations. . . 30
2. Whisker plot of wall clock times for evaluating IIA003 across all three implementations. . . 31
3. Whisker plot of wall clock times for evaluating IIA006 across all three implementations. . . 32
1 Introduction

A concept of significant interest to the field of computer security is access control: the mechanisms by which a system ensures that no entity is able to act outside of their intended permissions. A correct implementation of access control is crucial to system security; it is the means by which a system enforces separation of information based on confidentiality and integrity requirements. Failure of an access control mechanism can lead to unauthorized disclosure, modification, or destruction of critical data. The rules which an access control mechanism enforce are referred to as the policy. In modern software systems, it is often the case that the policy is specified in a domain-specific, declarative language, such as the eXtensible Access Control Markup Language (XACML) [3], to facilitate re-use and improve clarity.

Software written in any programming language is rarely free of defects, so we have no reason to expect that something written in a policy specification language is inherently correct. Tools and methods for analyzing access control policies have thus garnered great interest, and there have been several prior attempts to formalize access control [9] [24] [25] [28] [15].

A common theme among the prior work on this subject is the use of finite model checking [10] [13] – a technique that has been demonstrated to be more than sufficient for many realistic queries about properties of individual policies. A criticism of finite model checking, however, is that it is difficult to scale to large systems [7]. Furthermore, in systems such as Alloy, the undecidability of first-order logic means that analysis performed under bounds may not be complete [22]. In section 4, we present a use-case which we believe to be representative of the type of query which finite model checking can sometimes struggle with: a statement which quantifies over all policies.

This project set out to explore the feasibility of using a different kind of tool for policy analysis and verification: an interactive proof assistant. In particular, our goal was to demonstrate that Lean 4 [21] was suitable for proving statements about policies written in various policy specification languages, as well as for verifying properties of the policy specification languages themselves. Whereas prior works primarily focused on translating policies to statements in first-order logic [9], our approach put a greater focus on the semantics of the various policy language. We were interested in investigating verification as an approach to implementation-oriented problems such as the design of verified compilers – we were less concerned about the underlying logic, or how the meaning of a policy may be impacted by changes to the environment over time, which has been the concern of prior attempts at developing policy analysis tools.

Interactive proof assistants have been a common choice for semantic program analysis [23] [27] and the approach of verifying software by considering the semantics of the programming language it is written in [11]. Whereas finite model checking software reduces a software model to an instance of boolean satisfiability (SAT) or mathematical formula satisfiability (SMT) and handing the instance off to a solver [13], proof assistants work by modeling a system of logic such as the Calculus of inductive Constructions (CiC) [8] – a user applies rules of inference to axioms and statements which have already been proved to derive new truths. In this sense, it is much like writing a typical mathematical proof [11], except that the resulting proof is verified by a computer which, perhaps, has a better record than humans in being infallible.

In this work we explore two particular policy languages: release 3.0 of the eXtensible Access Control Markup Language (XACML), and the Rego language used by Open Policy Agent (OPA). XACML is a standard published by OASIS which specifies a way of declaring policies using XML. There are several conforming implementations in academic study [19] and popular use [26]. Rego is, as far as we can tell, a language specific to OPA. OPA is, however, a graduated project of the Cloud Native Computing Foundation, and the author has encountered it several times in his career, so we have confidence that it is not merely a toy. To our knowledge, this work is the first academic attempt at formalizing the Rego language.
1.1 Summary of Contributions

Our contributions are the following:

- A formal semantics in Lean 4 for the eXtensible Access Control Markup Language 3.0 (section 2).
  - Furthermore, these semantics are “executable” – a concrete implementation of the XACML language can be extracted from these semantics by compiling or interpreting them with Lean.
  - We additionally prove that the semantics meet the specification for behavior outlined in the XACML 3.0 standard.

- A formal semantics in Lean 4 for the Rego language (section 3).

- Expository proofs leveraging both formalizations:
  - The beginnings of a proof of correctness for an implementation of XACML 3.0 in Open Policy Agent (section 4).
  - A reproduction of the policy analysis example of [10] using our tool (section 5).

1.2 Availability of Project Materials

Source code for this project is available on-line (https://git.sr.ht/˜jakob/rego-proofs).
2 An Embedding of eXtensible Access Control Markup Language

Relative to other languages of interest to the PL field (such as Scheme), XACML 3.0 bears simple semantics. The standard describes execution behavior for certain policy elements represented in XML - often alongside pseudo code. Our specification of the XACML 3.0 semantics is little more than straightforward implementations of these behaviors, along with type specifications for the structures described in the XML Schema Definition. Our embedding is a single file consisting of less than 1,000 lines of Lean source code, and compilers written in OCaml to translate XML requests and policy specifications to their equivalent Lean representation. This section is an explanation of the various concepts in XACML and how they are represented in the embedding. Later sections demonstrate how these embeddings can be used to prove statements about access control policies, and how they can be used as a policy decision engine.

2.1 XACML Concepts And Their Representation in Lean

Ostensibly, the role of a Policy Decision Point (PDP) is to produce a single value describing whether access to a resource should be allowed. It is tempting, then, to consider the output of a PDP to be binary (permit or deny). But this is idealistic: we may want to consider policies which are not total - in other words, policies which do not necessarily always render a decision. To account for this, XACML is built upon a ternary system of logic, in which it is acceptable for the PDP to return an “indeterminate” value, rather than “permit” or deny.

To be more precise, XACML is built upon a senary-valued system of logic, which consists of “permit” and “deny,” three kinds of “indeterminate,” and a special value indicating that the policy is not applicable to the query in question. Previous works providing semantics for XACML have reduced the various logical constants to lattices. Although we mirror this structure in our embedding, we preserve the names and separation of types specified within the standard. Thus, we have four inductive types, used in three different contexts, to describe our system of logic.

```lean
inductive EffectType where
  | permit : EffectType
  | deny : EffectType
deriving BEq, Repr

inductive MatchResult where
  | is_match : MatchResult
  | no_match : MatchResult
  | indeterminate : MatchResult
deriving BEq, Repr

inductive Decision where
  | permit : Decision
  | deny : Decision
  | indeterminate : Decision
  | not_applicable : Decision
deriving BEq, Repr
```

---

This type is used to implement the semantics described in § 7.10 (Extended Indeterminate). It is only relevant to combining algorithms.

```lean
inductive Decision’ where
  | permit : Decision’
  | deny : Decision’
  | indeterminate : Decision’
  | indeterminate_d : Decision’
  | indeterminate_p : Decision’
  | indeterminate_dp : Decision’
  | not_applicable : Decision’
deriving BEq, Repr
```

1 This is described in §7.10 Extended Indeterminate of [3].
When juxtaposed with the above description, \texttt{EffectType} is aberrant. It is used in rules, which are the atomic policy element in XACML.

\begin{verbatim}
structure Rule where
target : Target
condition : Option Expression
effect : EffectType
obligations : List ObligationExpression
advice : List AdviceExpression
\end{verbatim}

A rule is applied to a request and results in a decision - indeterminate only if an error occurs in evaluating the rule, and inapplicable only if the target results in “no match.” Hence, \texttt{EffectType} can only take “permit” or “deny”.

A request is a collection of attributes (effectively, key-value pairs). Targets and conditions are collections of predicates over the request.

\begin{verbatim}
structure Attribute where
category : String
identifier : String
issuer : Option String
value : Primitive
deriving BEq, Repr

structure Match where
function : Identifier
designator : AttributeDesignator
value : Value
deriving BEq, Repr
def AllOf : Type := List Match
deriving BEq, Repr
def AnyOf : Type := List AllOf
deriving BEq, Repr
def Target : Type := List AnyOf
deriving BEq, Repr
\end{verbatim}

Attribute designators are the means by which a policy selects a set of attributes from the request. The similarity to \texttt{Attribute}’s structure should be explanation enough of how selection works. The one complexity is the inclusion of \texttt{must_be_present}, which indicates that the PDP should raise an error if no attribute with the given category, identifier, type, and issuer is present in the request.

\begin{verbatim}
structure AttributeDesignator where
category : String
identifier : String
type : XmlType
issuer : Option String
must_be_present : Bool
deriving BEq, Repr
\end{verbatim}

Note well that we return a \textit{set} of attributes (truthfully a bag, as we will describe later). It is perfectly acceptable in XACML for an attribute to be multi-valued – which is represented in the request as multiple attributes with the same category and identifier, but different values.

Rules are combined into Policies, which are further combined into PolicySets. A PolicySet can contain other PolicySets, but a Policy can contain only Rules. The output of a Policy or PolicySet is the result of a combinator algorithm applied to the results of its children.
structure Policy where
  combinator : RuleCombinatorAlgorithm
  target : Target
  obligations : List ObligationExpression
  advice : List AdviceExpression

structure PolicySet where
  combinator : PolicyCombinatorAlgorithm
  target : Target
  obligations : List ObligationExpression
  advice : List AdviceExpression

We chose to model the hierarchical relationships using a separate type to simplify our proofs.\footnote{When it comes to recursive definitions, Lean is far more strict than a “general purpose” programming language like OCaml or Haskell (which is not at all to insinuate that Lean could not be used as a general purpose programming language!) Specifically, all recursion must terminate unconditionally. If the Lean compiler is unable to infer this from the definition it requires a proof of termination from the user. By lifting the recursive evaluation to a single type with a single evaluation procedure, we avoid having to deal with mutually-recursive definitions, which is notoriously difficult to deal with in Lean.}

inductive PolicyElement where
  | PolicySet (ps : PolicySet)
  | Policy (p : Policy)
  | Rule (r : Rule)

inductive ElementTree where
  | Node (e : PolicyElement) (xs : List ElementTree)

Finally: XML encodes all non-markup data as strings; any sort of atomic or primitive value in XACML is encoded as an element containing a string with some type information attached. We represent this in the embedding as a pair of a type identifier and a string. Values can either be a primitive or a “bag” of primitives.\footnote{The distinction between the two is similar to how, in Scheme, a function can return multiple values by way of a \texttt{values} form.}

inductive XmlType
  | string : XmlType
  | boolean : XmlType
  ... deriving BEq, Repr

def Primitive : Type := XmlType × String 
deriving BEq, Repr

structure Bag where
  contents: List Primitive
deriving Repr

inductive Value where
  | primitive (p : Primitive) : Value
  | bag (b : Bag) : Value
deriving BEq, Repr

The Lean source file is interspersed with much more than is enumerated here. In particular, it is convenient for many of these types to be \texttt{LawfulBEq} (defined such that computational equality is consistent with propositional equality), and in the case of \texttt{Value}, the \texttt{Coe} typeclass is useful for mapping between types in XACML and types in Lean.

\subsection{On Computability}

One benefit to representing the semantics of XACML as a collection of data types and procedures is that they are \textit{computable}. This makes certain proofs quite easy – for example, contradictions. If we wish to
claim that there is a violation of some property in a policy, it suffices to construct a violation. Proving that it is a violation amounts to having Lean execute the policy and render a result. This proof methodology of leveraging computation with symbolic representations is over 22 years old - familiar to users of proof assistants as “proof by reflection.”

Another benefit is that, at the end of all of this, we have built a verified implementation of XACML “for free.” Lean 4 is a programming language with an optimizing compiler, so we merely need to compile our policy, and then the representation in Lean 4, and this results in an efficient PDP. When we describe this as a “verified implementation,” we mean that we have formally verified that the PDP correctly implements the policy according to the semantics of XACML by way of our proofs in section 2.5.1, which demonstrate that the Lean implementation meets the specification outlined in the standard. We explore the feasibility of actually using this as a PDP in section 6.

### 2.2 Side Information and Error Handling

Expanding on our model of what a PDP “should” do – there may be cases where we want some kind of effect on the outside world as a result of evaluating a policy. XACML models this through “advice” and “obligations”. They are logically equivalent, but advice may safely be ignored, whereas the standard specifies that the Policy Enforcement Point must act on obligations. Our model only considers the PDP, so we treat these as generic side information.

The standard specifies three errors which are reported through a `StatusCode` value.

- `urn:oasis:names:tc:xacml:1.0:status:missing-attribute`
- `urn:oasis:names:tc:xacml:1.0:status:processing-error`
- `urn:oasis:names:tc:xacml:1.0:status:syntax-error`

We deliberately do not model `urn:oasis:names:tc:xacml:1.0:status:syntax-error` because our OCaml compiler should handle any case in which it would be raised. The policy elements all return a sum type where the left-hand side is the decision and side information, and the right-hand side is a status in the case of an error. If the left-hand side of the sum type is populated, then we assume that the result had a `StatusCode` of `urn:oasis:names:tc:xacml:1.0:status:ok`. The standard specifies that `StatusDetail` and `StatusMessage` are not mandatory, so we omit them from our model.

In summary, we model side information using a product type, and we model errors as a sum type.

### 2.3 Identifier Equality

The standard specifies requirements for an identifier equality algorithm. Lean’s string equality procedures are compliant, and the attributes to match on are the natural choice in our embedding. Where equality is defined on an XML attribute, the corresponding Lean type has a specialized `BEq` instance.

---

4In theory we could be compliant by compiling to a constant “Indeterminate” with status code `urn:oasis:names:tc:xacml:1.0:status:processing-error`. 
2.4 Concepts Modeled but Absent in the Embedding

XACML specifies a means of giving names to particular objects and referring to them elsewhere in the policy. For example, we could define one PolicySet, give it a name, and include it as a child of another PolicySet.

We support variable references in our model, but they are erased by the compiler – we merely replace any `VariableReference`, `PolicyIDReference`, or `PolicySetIDReference` with the definition it refers to. This is permissible by the standard:

In any place where a `<VariableReference>` occurs, it has the effect as if the text of the `<Expression>` element defined in the `<VariableDefinition>` element replaces the `<VariableReference>` element. Any evaluation scheme that preserves this semantic is acceptable.

This approach ensures that circular references are handled by the compiler, which is required by the standard.

We would like to emphasize that, at the time of writing, the Lean source code by itself is not sufficient to define the semantics of XACML. Some rules specified by the standard are enforced only in the compiler. For example, the following type checks in Lean:

```
#check
ElementTree.Node
(PolicyElement.Rule ...)
[ElementTree.Node
(PolicyElement.Rule ...)
[]]
```

This represents a Rule containing another Rule, which is forbidden. Such an expression is never constructed by our compiler, but users of the embedding should exercise caution if constructing policy elements by hand.

2.5 Justifying our Embedding

The simple answer to the question of “what was our procedure for coming up with the properties to encode in the embedding?” is that we read the standard from beginning to end. Sections 1 through 4 are non-normative, so they were ignored. Section 5 lists the syntactic elements in XACML. We enumerated this list and determined which elements would need to be represented in our embedding, and proceeded to write Lean definitions for each of the structures. Our goal was to provide a model that was as complete as possible – so that this project could be leveraged to prove a refinement of a policy down to the status codes, obligations, and advice it returns. However, we did not wish to encumber ourselves or users of the embedding, so there are parts of the standard which we chose toomit. Table 2.5 provides justification for omitting the elements that we did.

Additionally,

- `PolicySetId` is not explicitly encoded in the `PolicySet`. Rather, it is encoded in the Lean identifier assigned my the compiler.
- The `MaxDelegationDepth` attribute is omitted because it is defined in `[XACMLAdmin]`.
- `CombinerParameters` is omitted because no combination algorithm specified by the standard accepts parameters.
<table>
<thead>
<tr>
<th>Section Title</th>
<th>Reason for Omission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element <code>&lt;Description&gt;</code></td>
<td>Has no semantic meaning.</td>
</tr>
<tr>
<td>Element <code>&lt;PolicyIssuer&gt;</code></td>
<td>Defined in [XMLAdmin]</td>
</tr>
<tr>
<td>Element <code>&lt;PolicySetDefaults&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;XPathVersion&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;PolicySetIdReference&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Element <code>&lt;PolicyIdReference&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Simple type <code>VersionType</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Simple type <code>VersionMatchType</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Element <code>&lt;PolicyDefaults&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;CombinerParameters&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;CombinerParameter&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;RuleCombinerParameters&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;PolicyCombinerParameters&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;PolicySetCombinerParameters&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;AttributeSelector&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Element <code>&lt;RequestDefaults&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;PolicyIdentifierList&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;MultiRequests&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;RequestReference&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;AttributesReference&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;StatusMessage&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;StatusDetail&gt;</code></td>
<td>Optional according to §10.2.1: Schema elements</td>
</tr>
<tr>
<td>Element <code>&lt;VariableDefinition&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Element <code>&lt;VariableReference&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
<tr>
<td>Element <code>&lt;Content&gt;</code></td>
<td>Dissolved by compiler translation</td>
</tr>
</tbody>
</table>

Table 1: Omitted XACML 3.0 elements.
We chose not to implement any of the extensions to XACML 3.0, hence [XMLAdmin] appearing among our comments about justification.

In the same vein as [23], we omit the AttributeSelector element because it would add significant complexity, and we aver it is is not unreasonable to assume the attribute values are already obtained from their respective sources.

### 2.5.1 Evaluation Tables

The following sections provide tables describing the semantics of various elements:

- §7.7 Target evaluation
- §7.11 Rule evaluation
- §7.12 Policy evaluation
- §7.13 Policy Set evaluation
- §7.14 Policy and Policy set value for Indeterminate Target

We have high confidence that our implementation conforms to these tables because our procedure of engineering evaluation procedures was essentially to translate the tables to Lean. Nonetheless, Lean provides us with the tools necessary to formalize these tables and prove that our procedures are accurately described by them. We will demonstrate this by formalizing and proving the contents of the AnyOf match table described in §7.7 (table 2.5.1).

The three rows in the table will correspond to three statements, which we will translate to propositions in first-order logic. Let \( r \) be the set of <AllOf> values, and let \( R \) be the resulting <AnyOf> value. Denote “Match” by \( \top \), “No match” by \( \bot \), and “Indeterminate” by \( I \).

1. \( \top \in r \Rightarrow R = \top \)
2. \( \top \notin r \land I \in r \rightarrow R = I \)
3. \( \top \notin r \land I \notin r \rightarrow R = \bot \)

Note that the syntax specified in the standard requires that an <AnyOf> element contain at least one <AllOf> element, so \( |r| > 0 \). We explicitly encode this as an assumption in our translation of the third statement because our formalization does not enforce this - rather, an empty <AnyOf> element could be produced in our formalization, but our compiler would not produce one – as we described in section 2.4.
The theorem definitions are equivalent to the preceding statements, but the proofs may look foreign to one unfamiliar with interactive proof assistants. This is written in “tactic mode,” whereby we prove a statement by backward reasoning. All three proofs rely on unwrapping the definition of \texttt{AnyOf.eval} and iteratively
rewriting based on facts we can derive from the hypothesis. For example, when we unfold the evaluation procedure in `table_1_target_match_table`, we have a “goal state” of the following:

```
Goals (1)
disj : List AllOf
category : Request

h_no_match : \( \forall (\text{conj} \in \text{disj}) \rightarrow \text{AllOf}.\text{eval} \text{conj} \text{context} = \text{MatchResult}.\text{is_match} \)

h_indep : \( \exists \text{conj} , \text{conj} \in \text{disj} \rightarrow \text{AllOf}.\text{eval} \text{conj} \text{context} = \text{MatchResult}.\text{indeterminate} \)

x : AllOf
hx : x \in \text{disj} \rightarrow \text{AllOf}.\text{eval} x \text{context} = \text{MatchResult}.\text{indeterminate}

(if List.\text{isEmpty} \text{disj} = \text{false} then MatchResult.\text{is_match}
else
  if \exists a, a \in \text{disj} \rightarrow \text{AllOf}.\text{eval} a \text{context} = \text{MatchResult}.\text{is_match} then MatchResult.\text{is_match}
  else
    if List.\text{elem} \text{MatchResult}.\text{indeterminate} \rightarrow \text{MatchResult}.\text{is_match}
      then MatchResult.\text{indeterminate}
      else MatchResult.\text{no_match}) = MatchResult.\text{indeterminate}
```

Our approach was to rule out all branches producing a value other than `MatchResult.indeterminate`. So we want to show that:

1. `List.\text{isEmpty} \text{disj} = \text{false}`
2. \( \neg \exists a, a \in \text{disj} \rightarrow \text{AllOf}.\text{eval} a \text{context} = \text{MatchResult}.\text{is_match} \)
3. `List.\text{elem} \text{MatchResult}.\text{indeterminate} \rightarrow \text{MatchResult}.\text{is_match} = \text{true}`

\(\square\) corresponds to `hnonempty`, and \(\clubsuit\) corresponds to `hbranch`. Once we justify these and simplify, we are left with the following goal state:

```
Goals (1)
disj : List AllOf
...
\( \forall (x : \text{AllOf}), x \in \text{disj} \rightarrow \neg \text{AllOf}.\text{eval} x \text{context} = \text{MatchResult}.\text{is_match} \)
```

Which is equivalent to the remaining statement above. Conveniently, this appearing among our hypotheses. A small amount of goal manipulation is required to apply said hypothesis, but it takes little effort to complete the goal from here.

### 2.6 Assumptions and Limitations

The most significant limitation in actually evaluating our formalization as an implementation of XACML 3.0 is the lack of matching function implementations. §10.2.8 lists over 250 functions whose implementation is mandatory, some of which would be exceptionally difficult to model in Lean (e.g., `string-regexp-match`, whose behavior is the regular expression language specified by XML Schema Part 2: Datatypes Second Edition – i.e., “Perl-compatible” regular expressions.) Although practitioners tend to advise against introducing arbitrary axioms, due to their ability to introduce inconsistent logic, specifying known input/output pairs of interest as axioms is a way to proceed with analysis without the need to model arbitrarily complicated matching functions.

\(\square\) Perhaps with the exception of `table_1_target_match_table`, where the simplifier is intelligent enough to do case analysis and transform the goal for us.

---

5 Perhaps with the exception of `table_1_target_match_table`, where the simplifier is intelligent enough to do case analysis and transform the goal for us.
Evaluation order is unspecified in general in the standard, and thus, the evaluation of a XACML policy is not necessarily deterministic. This is well understood by the authors of the standard - there are combinator algorithms with a specified evaluation order for this reason (e.g. Ordered-permit-overrides, Legacy Ordered-deny-overrides). Our model does not consider nondeterminism (and thus, each ordered combinator is aliased to its “unordered” counterpart), but future efforts certainly could. The \texttt{Nondet} monad of Mathlib would likely allow for a straightforward specification of the semantics comparable to what we have presented here.

2.7 Future Work: Evaluation Against Conformance Test Suite

One way to gain greater confidence that our model is accurate would be to process the conformance test suite through our implementation. OASIS provided a test suite for XACML 2.0, but has not published one for XACML 3.0. The AuthzForce test suite includes a translation of the OASIS conformance tests to XACML 3.0. These are obviously non-normative, but the results would not be entirely invaluable.

We have used some of the aforementioned test cases in evaluating our model, but have stopped short of testing our implementation against the test suite in its entirety, chiefly because there are functions (used for matches and expressions) specified by the standard that we have not yet implemented, as we described in section 2.6. A comprehensive list of the test cases we excluded can be found in \texttt{benchmarks/excluded.txt} in the supplemental materials. We were able to evaluate the Lean model against 114 of the 455 test cases. The 341 cases we excluded were due to 170 functions, in total, that have not yet been formalized – also identified in the \texttt{excluded.txt} list. Although we anticipate the process to be exacting, we do not believe there are fundamental issues that would restrict our implementation from passing all 455 test cases. All of the functions that remain unimplemented are deterministic and well-typed.

The test suite has nonetheless been helpful in identifying an issue with our evaluation of \texttt{<Apply>} expressions - one which we have not resolved at the time of writing. So in addition to implementing the remaining XACML match functions, addressing the issues the test suite has identified is also ground we intend to tread in the near future.

2.8 Future Work: Verified Compilation

A disadvantage to our compiler being implemented in OCaml is that the process of translating policies to structures in Lean is opaque to the theorem prover. As we noted in section 2.4, not every structure that can be represented in our Lean model corresponds to a legal construct in XACML (e.g., a rule containing a rule). This would be less of an issue if the theorem prover knew about how the translation procedure worked; by translating the compiler from OCaml to Lean, we could rigorously prove that the compiler respects the rules outlined in the standard.

Our primary reason for implementing the compiler in OCaml rather than using Lean at the outset was to enable a greater development velocity: the availability of high-quality third-party software libraries for OCaml was a significant advantage in this regard. In addition, and as we note in section 2.4, Lean imposes significant restrictions on recursively-defined functions. Although our compilation procedure is not subject to these restrictions, because we visit each node of the syntax tree a known and finite number of times, we would need to prove this to Lean. OCaml, on the other hand, imposes no such restrictions.

It is worth noting that the superset of Lean used for metaprogramming does permit recursively-defined functions without a proof that they are well-formed, but we then forfeit the ability to use Lean to reason about them. It is also worth noting that the aforementioned restrictions could hypothetically be encoded in Lean’s dependent type system, but it is not clear whether this would impose a burden on users of the semantics, and if so, whether the value of this would justify the burden.
2.9 Related Work

Ramli et al. [24] provide formal semantics for XACML 3.0 component evaluation and the XACML 3.0 standard combining operators. Their work focuses on the logical underpinnings of XACML and has been very influential on this paper. However, their formalization omits side information and error reporting (advice, obligations, and statuses), whereas this work includes these aspects of XACML evaluation.

A similar work is Caserio et al.’s formalization [6], which is oriented toward better coverage criteria for policy test generation. They build upon [24], tailoring the formalization to that particular domain, but also omit side information and error reporting.

Automated analysis of XACML policies is an approach which is several years old at this point. Most influential to this work was Fisler et al.’s Margrave tool [10], which translates XACML 1.0 policies to models in the Alloy specification language. It is focused on query-based verification of policies and supports approaches such as change-impact analysis, whereby the tool deduces which situations behave differently when a particular change is applied to a policy. Our tool is better suited to formally verifying properties which are already known to the user, rather than identifying them. We provide a more detailed explanation of the difference in approaches in section 5.

Hughes and Bultan [13] provide a formal model of XACML 1.0, a formal definition of policy equivalence, and a translation from properties to instances of SAT to provide query-based verification comparable to Margrave. They claim that Alloy was unsuitable for producing the sorts of results they achieved by way of SAT reduction.

The model of access control provided by XACML is typically understood to be Attribute-Based Access Control (ABAC). Access control and ABAC specifically have seen formalizations in the literature [15], often with concepts from logic programming [28].

\[^{6}\text{and policies written in other languages}\]
3 An Embedding of the Rego Language

In addition to a formal model of XACML 3.0, we have produced a formal model of the Rego language used by Open Policy Agent [1] (hereafter referred to as OPA). Much like our embedding of XACML, we provide an OCaml compiler which takes as input a Rego policy and compiles it to our Lean representation. Unlike the XACML embedding, the supplementary Lean code is far less important, comprising less than a hundred lines.

3.1 Justifying our Embedding

Unlike XACML, Rego does not have an official standard specifying its semantics. The documentation provides an EBNF grammar describing the syntax and several input/output examples, but the reality is that the semantics of Rego are specified by the implementation. To yield a compatible semantics, we resorted to effectively reverse-engineering the language through experiments and inspecting the source code.

Because the language is effectively specified by the implementation, the test suite used by OPA is somewhat more normative than the one available to us for XACML. However, for many of the same reasons outlined in section 2.5, we consider this future work, as well as the relative inability to use Lean to evaluate policies, which we describe in section 3.7.

3.2 Rego Concepts

The majority of the Lean code concerns the data types that are specified in Rego:

```lean
inductive Any where
| null
| bool (b : Bool)
| num (n : Nat)
| str (s : String)
| array (xs : Array Any)
| set (elts : List Any)
| object (field : Lean.AssocList Any Any)
deriving Inhabited
```

We admit that there are significant limitations to our model – in particular, the `num` case is being modeled as \( \mathbb{N} \), whereas a truly accurate model of the semantics would use `Lean.JsonNumber` or similar. We chose not to use such a representation because of the difficulty involved in converting to and from natural numbers, which is something we needed to do to provide the semantics of array indexing.

Rego is evidently based on Datalog, and many of its concepts correspond to ideas in Datalog (and logic programming in general). Rego policies make decisions based on hierarchical structured data referred to as the base document - in database terminology, the extensional database. A Rego policy is comprised of rules which define the content of a virtual document - in database terminology, the intensional database. It is worth being aware that Rego, unlike Datalog, does not support recursive rules.

An explanation more appropriate for those not familiar with database theory is that a policy is comprised of declarative functions (rules) which define identifiers and their corresponding values. The following is an example of a Rego rule:

7See further: section 3.7
The rule above defines a set named `q` which contains all of the names of websites in some set in the base document named `sites`. A rule is represented as a conjunction of queries, which are functions of the current document. We say that a rule is applicable if none of its queries evaluate to the boolean value false. i.e., all of the following are semantically valid and will result in `x` being assigned to 7.

\[
\begin{align*}
  x &:= 7 \{ 0 \} \\
  x &:= 7 \{ "false" \} \\
  x &:= 7 \{ [] \}
\end{align*}
\]

Which is why, in the the compiler output of the first example, the query parts of the rule are compiled to comparisons against `Rego.Any.bool False`:

```lean
def rego_error := (False)
```

This is an inductive predicate whose arms correspond to the rules defining `q`. The example we provided contains a single rule, but Rego policies are permitted to use multiple rules to define a variable, in which case the compiled inductive predicate would have as many arms as there are rules.

Sets are compiled to inductive predicates which specify a membership predicate. In our example, `q` is parameterized by `vx_0`. We interpret this as saying that `q` contains `x` if and only if `q x` is true. We take a similar approach for rules defining scalars and objects.

```lean
def scalar_specified_by (x : Any) (ind : Any → Prop) : Prop := ind x
def set_specified_by (x : Any) (ind : Any → Prop) : Prop :=
  match x with
  | Any.set xs => \( \forall \( k : Rego.Any \), k ∈ xs ⇔ ind k \) \\
  | _ => False
def object_specified_by (x : Any) (ind : Any → Prop) : Prop :=
  match x with
  | Any.object xs => \( \forall \( k v : Rego.Any \), (xs.find? k = some v) ⇔ ind k \) \\
  | _ => False
```

This is convenient for us as compiler developers because we can compile references to other rules as quantification over instances of `Any` satisfying one of the predicates given above. On the other hand, this puts us as a disadvantage as proof engineers, because we lose the ability to simply ask Lean to evaluate a policy. For a user to show that `q` takes a particular value would demand a proof. Our approach is, nonetheless, the natural one to shallowly-embedding such a language in Lean. Indeed, even *The Hitchhiker's Guide to Logical Verification*\[5\] appeals to Prolog as an analogy when introducing inductive predicates.

Evaluation certainly could be implemented; the top-down evaluation algorithm used by OPA could be translated to Lean, using Lean's reflection facilities (`TacticM`) to extract the predicate arms and derive new facts about the resulting document. At this point, we consider this future work.

In the example we began with, what remains to be explained is the `rego_error` term. We will explain it by considering the following policy:

\[
\begin{align*}
  q &:= 1 \\
  q &:= 2
\end{align*}
\]
which is compiled to the following Lean code:

```lean
inductive q (vx__0 : Rego.Any) where
| rule0 : (Ne (eq vx__0 (Rego.Any.num 1)) (Rego.Any.bool False)) → q vx__0
| rule1 : (Ne (eq vx__0 (Rego.Any.num 2)) (Rego.Any.bool False)) → q vx__0

def rego_error := ¬(∃! (vthe__5 : Rego.Any), (scalar_specified_by vthe__5 q))
```

By itself, the semantics described by \( q \) permits something illegal: a scalar variable having more than one value. In OPA, this is reported as an error.

```
1 error occurred: test.rego:4: eval\_conflict\_error: complete rules must not produce multiple outputs
```

Our embedding of XACML modeled error conditions with a sum type, but such an approach is inconvenient here. Instead, the compiler produces a predicate which is true when a policy would cause an error in OPA. To prove that an evaluation of this policy was correct according to the semantics, one would need to prove that there exists exactly one \( vthe_5 \) satisfying \( q \) - which is obviously false in this example.

Note that \( rego\_error \) is a nontrivial expression here, but merely \( False \) in the first example - this issue applies only to scalar values, whereas the first example only contained a rule defining a set.

### 3.2.1 Compilation of Rules with Reference Heads

What prevents us from using a simple name-based mapping of rules to variables is that a rule head can be a reference. That is, a single rule can actually partially specify more than one variable.

The following program is legal in Rego[6]:

```rego
fruit.apple.seeds = 12
fruit.orange.color = "orange"
```

and results in the following document:

```json
{
  "fruit": {
    "apple": {
      "seeds": 12
    },
    "orange": {
      "color": "orange"
    }
  }
}
```

If we consider the example above, it should be clear that the following is an equivalent program:

```rego
apple['seeds'] = 12
fruit['apple'] = apple'
orange['color'] = "orange"
fruit['orange'] = orange'
```

Given a rule of the form \((x_0, \ldots, x_n) = x_{n+1}\), our compiler strips the tail, producing the rule: \(x'_n[x'_{n+1}] = x_{n+1}\), and then recurses on \((x'_0, \ldots, x'_{n-1}) = x'_n\) until we reach a basic rule.
3.2.2 Scope

By far, the aspect of Rego which was most difficult to characterize was its approach to scope. Rego permits an operation for introducing local variables in a rule, called assignment and denoted by the := operator.

\[
x \{ \\
y := 7 \\
y == 8 \\
\}
\]

In this example, \(y\) is bound to 7 in the extent following the assignment. The ordering is important; this rule is illegal:

\[
x \{ \\
y == 8 \\
y := 7 \\
\}
\]

OPA produces the following error:

1 error occurred: test.rego:5: rego_compile_error: var y referenced above

We raise an “unbound variable” error. These are equivalent notions.

However, variables need not be explicitly assigned (or introduced), so long as they are constrained. For example, this is a legal program. \(y\) refers to \(z\) before it is introduced.

\[
x \{ \\
y := z \\
z = 7 \\
\}
\]

This is, however, an illegal program, because \(z\) is unconstrained. Open Policy Agent refers to such a variable as “unsafe,” meaning that the domain of possible values for \(z\) is unbounded.

It is easier to consider the definition of the converse: a variable is “safe” if it appears as the output of at-least-one non-negated expression. The naïve interpretation of this is that some expression must containing it and no other free variables. This is not quite the case, because \([x, \text{"world"}]= [\text{"hello"}, y]\) is legal (and will bind \(x\) to “hello” and \(y\) to “world”), but the following program is illegal:

\[
test \{ \\
z := [\text{"hello"}, y] \\
[x, \text{"world"}]= z \\
\}
\]

Even though there is clearly only one value of \(z\), OPA rejects this. So some care is needed in the treatment of safety. Cases that can be determined statically via destructuring are safe. Cases where this is not the case are considered unsafe.

To conclude:

- If a variable appears on the left-hand side of an assignment, it is "bound".  
  - Bound variables can occur only after the initial assignment.
- Otherwise, it is free.  
  - Free variables can be referred to from anywhere within the query.
– They can be explicitly introduced with the *some* keyword, but this is not necessary.
– Free variables must appear as the output of at-least-one non-negated expression.

Furthermore, variables defined through this sort of assignment are scoped to the query. Queries can be nested, and scope is lexical. For example, this policy is illegal:

```rego
x {
    every i in {0} {
        sites[i].servers[j].hostname
    }
    i == 0
}
```

While this is legal:

```rego
x {
    i := 0
    every j in {0} {
        sites[i].servers[j].hostname
    }
}
```

Top-level definitions are rules, which are not ordered, so this is legal:

```rego
z := x + y
x := 1
y := 2
```

### 3.2.3 Rules With Formal Parameters

Rules can be defined in terms of variables which are constrained within the rule body. For example:

```rego
y := 3
x := y { y := 2 }
```

Produces the document

```json
=> {"test":{"x":2,"y":3}}
```

Note that this is semantically different from the program

```rego
y := 3
x := y { true }
```

whose resulting document is

```json
=> {"test":{"x":3,"y":3}}
```

This is analogous to the formal parameters of a function shadowing any identifiers that might exist in the top-level scope, except that a variable is only considered a formal parameter if it is assigned within the body of the rule. Though, we should note, Rego has a notion of user-defined functions which are more specific than the sorts of rules exhibited above. Note that the identified has to appear on the left-hand side of an assignment – not used in a way that would introduce it as a free local variable.

```rego
myarray = [1,2,3]
y := 3
x := y { myarray[y] == 2 }
```
Compared to

\[
\text{myarray} = [1,2,3] \\
y := 3 \\
x := z \{ \text{myarray}[z] == 2 \}
\]

### 3.3 Summary

There are four cases we care about:

- Local variables
- Formal parameters
- Implicitly quantified local variables
- Anything outside of the scope (used)

Our procedure for identifying the cases is the following:

1. Look at the rule head and identify potential formal parameters.
2. If they’re bound in the body, they are a formal parameter.
3. Otherwise, if they’re in the environment, then they’re “used.”
4. If they’re not in scope, then they’re a formal parameter.

For scalar, set, and object rules, we compile references to a formal parameter as a reference to the “subject” of the inductive predicate.

### 3.4 Dependency

This is a valid Rego policy:

```rego
package experiment
x := y
y := 6
```

However, Lean requires forward declarations. If we were to compile each rule individually and concatenate the output, we would produce the following program.

```lean
def x := y
def y := 6
```

This does not compile, with Lean stating that `y` is an unknown identifier. An easy solution would be to make all definitions mutual:

```rego
mutual
  def x := y
  def y := 6
end
```
This comes at a significant cost to the user, however, as mutually-recursive definitions are cumbersome to use in Lean. Instead, we can use static analysis to build a graph and perform a topological sort on the nodes to order the definitions in our output, deferring to `mutual` in the case of a cycle.

### 3.5 Negation

Like Datalog, Rego has a concept of negation in the form of the `not` keyword, which can be used the following ways:

```plaintext
not obj.foo.bar.baz
```

Which is true if `foo.bar.baz`, `foo.bar`, or `foo` does not exist, or is false.

```plaintext
not a_set["foo"]
```

Which is true if `"foo` does not belong to `a_set`.

```plaintext
not any_match
```

Which is true if no values in `set` make function `f` true.

```plaintext
not any_not_match
```

Which is true if all values in `set` make function `f` true.

Negation of an expression `expr` is compiled to `(Not expr)`, which we can justify for each of the examples above:

1. This is acceptable because `obj.foo.bar` is constrained to an object and then checked for `false`. If `foo` or `bar` or `baz` doesn’t exist then it’s vacuous because there’s no object that satisfies the constraint.
2. The same explanation above applies, just with sets.
3. and 4. are natural given what behavior we would expect of `not` in Lean.

### 3.6 Comparison

The ordering operators (`>`, `>=`, `<`, `<=`) are implemented as translations of OPA’s Go source code, because their behavior is unintuitive when applied to values of differing types. (e.g., `7 > "hi"` is a legal query and is false; `7 < "hi"` is true.) Equality (`==`) is actually syntactic sugar for unification (`=`). We defer to Lean’s implementation of equality because, to the extent that our inspection is comprehensive, it is consistent with Rego’s notion of unification.

### 3.7 Assumptions and Limitations

Much like our embedding of XACML, we lack an implementation of many of the standard library functions provided by Rego. Rego provides several procedures such as `http.send` which could not reasonably be modeled by pure Lean code. If one were to analyze a policy containing calls to such procedures, it is likely that the user would need to define a mocked implementation of the function themselves – to come up with
a reasonable stand-in for such a function necessarily requires some understanding of the underlying problem domain.

There are several areas of the language which are quite poorly defined in our embedding; we hope that we can address these shortcomings through continued work on this project. In particular, we do not have good support for comprehension expressions or the `with` keyword.

With the exception of keywords, our compiler does not handle imports.

As mentioned in the beginning of this section, we would like to refine the data model to use a more accurate representation of numbers. We would also like to more thoroughly specify the behavior of operations we consider unintuitive, such as indexing a set by a number. It is perhaps more accurate to think of our work as an embedding of merely a subset of Rego.
4 Case Study: Policy Equivalence

Our motivation in modeling two distinct policy specification languages was to explore the feasibility of proving that a policy written in one language has equivalent behavior to a policy written in another. This has potential for real-world application. Suppose an organization publishes a mechanization of an institutional policy as XACML, and another organization is required to conform to that institutional policy, but is unable to use XACML for whatever reason (e.g., cost, hardware limitations, using interchange formats other than XML). The latter organization could re-implement the policy in their language of choice and prove that its behavior is equivalent to the former organization’s mechanization. To explore this idea, we implemented the XACML 3.0 semantics in Rego and set out to prove that said Rego implementation is equivalent to the formal semantics we provided for XACML.

We chose this direction - to model XACML inside of Rego rather than the other way around - because Rego is arguably more general than XACML. One could draw comparison between our approach here and Dougherty et al. [9], as they translate XACML 1.0 to Datalog – our transformation of XACML 3.0 to a Datalog-like language is remarkably similar.

Equivalence, here, is a strong statement – in software verification, it is somewhat more common to show refinement, by which we mean that if the XACML policy produces a decision, then the Rego implementation would produce the same decision when run on that policy. By equivalence, we mean that if the XACML policy produces a decision, then the Rego implementation would produce the same decision when run on that policy, and that if the Rego implementation produce a decision, then the model would produce that same decision. Access control is a domain demanding exceptionally high assurance, so we believe that equivalence is far more desirable.

There is a significant restriction which we must consider, however: Rego does not permit recursive rules and so, by extension, does not permit recursion in general. The minor syntactic inconvenience this causes is that we cannot defer to rule combinators in a natural way. I.e., we cannot write the following:

```rego
evaluate_rule_combinator("ordered_deny_overrides", results) := result if {
    result := evaluate_rule_combinator("deny_overrides", results)
}
```

The more salient issue is that we cannot naturally express PolicySet evaluation.

```rego
evaluate_policyset_combinator(_, results) := { "decision": "not_applicable" }
evaluate_policyset_decision(policyset) := decision if {
evaluate_target(policyset.target) == { "match_result": "is_match" }
child_policy_results := [result | policy := policyset.policies[_;]; result :=
evaluate_policy_decision(policy)]
child_policyset_results := [result | policyset := policyset.policysets[_;]; result :=
evaluate_policyset_decision(policyset)]
child_results := array.concat(child_policy_results, child_policyset_results)
decision := evaluate_policyset_combinator(policyset.combinator, child_results)
} else := decision if {
evaluate_target(policyset.target) == { "match_result": "indeterminate" }
child_policy_results := [result | policy := policyset.policies[_;]; result :=
evaluate_policy_decision(policy)]
child_policyset_results := [result | policyset := policyset.policysets[_;]; result :=
evaluate_policyset_decision(policyset)]
child_results := array.concat(child_policy_results, child_policyset_results)
decision := lift_decision_indeterminate(evaluate_policyset_combinator(policyset.combinator, child_results))
} else := { "decision": "not_applicable" }
```

Rego complains that we have a recursive call in the definition of child_policyset_results. This is a familiar issue, as Lean also places a restriction on recursion, namely that it be well-founded. This is why we used the ElementTree structure.
Our situation here is a special case of well-founded: we know that the tree of policy elements is finite. Of course, we cannot dispense this knowledge to the Rego evaluator in the way we did to Lean, so we adopt the workaround of having \texttt{child\_results} be a parameter and compiling a policy to an explicit post-order traversal of the tree.

\begin{verbatim}
evaluate_policyset_decision(policyset, child_results) :=
    decision if {
        evaluate_target(policyset.target).match_result.value == true
        decision := evaluate_policy_combinator(policyset.combinator, child_results)
    } else if {
        is_indeterminate(evaluate_target(policyset.target))
        decision := lift_decision_indeterminate(evaluate_policy_combinator(policyset.combinator, child_results))
    } else := {
        "decision\_extended": "not\_applicable"
    }

... decision := evaluate_policyset_decision(..., [
    evaluate_policy\_decision(...),
    evaluate_policyset\_decision(...),
    ...])

advice := evaluate_policyset\_advice(..., [
    evaluate_policy\_advice(...),
    evaluate_policyset\_advice(...),
    ...])

obligations := evaluate_policyset\_obligations(..., [
    evaluate_policy\_obligations(...),
    evaluate_policyset\_obligations(...),
    ...])
\end{verbatim}

Ultimately, this means that we cannot use Lean to formalize a refinement proof that is quantified over all XACML policies - because to get a model of the Rego implementation of a particular policy, we need to first compile the policy to Rego, and then compile the Rego to Lean. If the compilation step were written in Lean rather than OCaml, then this would be tractable. Instead, we settle for proofs that each of the elements of XACML policy evaluation are correct, and that (in the cases we are able to dispense) they are also correct when combined.

One difference from our XACML embedding that we adopted for convenience sake is rather than adopting an analog to the \texttt{Result} structure, we defined the \texttt{decision}, \texttt{advice}, and the \texttt{obligations} separately. Rules in Rego are more ergonomical when small and self-contained. This affects our proof obligations, as we want to show equivalence for each of these three values.

At the time of writing, we have not completed a proof of correctness for the entire Rego policy. But we have proved the following theorem to demonstrate the approach:

\begin{verbatim}
theorem designator\_matches\_correct :
    \forall (designator : AttributeDesignator) (attr : Attribute),
    designator\_matches
        (rego\_of\_attribute\_designator\_designator)
    (rego\_of\_attribute\_designator\_attr)
    (Rego.Any.bool true)
    \leftrightarrow
    attribute\_matches designator attr :=
\end{verbatim}

Where \texttt{rego\_of\_attribute} and \texttt{rego\_of\_attribute\_designator} transform XACML \texttt{<Attribute>} and \texttt{<AttributeDesignator>} elements to JSON objects, and \texttt{designator\_matches} is the compiled version of the following Rego rule:

\begin{verbatim}

\end{verbatim}

\begin{verbatim}

\end{verbatim}

\[\text{In general, the behavior for advice and the behavior for obligations are precisely the same.}\]
designator_matches(designator, attribute) if {
  attribute.category == designator.category
  attribute.identifier == designator.identifier
  attribute.value.type == designator.type
  attribute.attr.issuer == null
}
designator_matches(designator, attribute) if {
  attribute.category == designator.category
  attribute.identifier == designator.identifier
  attribute.value.type == designator.type
  attribute.attr.issuer == designator.issuer
}

Our proof was nearly 350 lines of Lean code. The theorem itself was only 100, but we proved several very similar rewrite rules about rego_of_attribute and rego_of_attribute_designator. For example, that the “category” field of the resulting object is equal to the category field of the structure that was passed as input.

```lean
theorem category_inv : 
∀ (attr : Attribute),
  Rego.Any.deref (rego_of_attribute attr) (Rego.Any.str "category")
= Rego.Any.str attr.category := by
  intro attr
  simp [Rego.Any.deref, rego_of_attribute]
match attr with
| { issuer := some issuer, .. } =>
  simp [Lean.AssocList.find?, Option.getD]
| { issuer := none, .. } =>
  simp [Lean.AssocList.find?, Option.getD]

theorem designator_category_inv :
∀ (attr : AttributeDesignator),
  Rego.Any.deref (rego_of_attribute_designator attr) (Rego.Any.str "category")
= Rego.Any.str attr.category := by
  intro attr
  simp [Rego.Any.deref, rego_of_attribute_designator]
match attr with
| { issuer := some issuer, .. } =>
  simp [Lean.AssocList.find?, Option.getD]
| { issuer := none, .. } =>
  simp [Lean.AssocList.find?, Option.getD]
```

Because of the repetitive nature of these proofs, we explored the possibility of leveraging aesop to automatically derive them. aesop is not quite clever enough to do the unfolding itself, but with a slight shove, it was able to prove the theorems above.

```lean
theorem category_inv' : 
∀ (attr : Attribute),
  Rego.Any.deref (rego_of_attribute attr) (Rego.Any.str "category")
= Rego.Any.str attr.category := by
  unfold Rego.Any.deref
  unfold rego_of_attribute
  aesop

theorem category_inv'' : 
∀ (attr : Attribute),
  Rego.Any.deref (rego_of_attribute attr) (Rego.Any.str "category")
= Rego.Any.str attr.category := by
  aesop (add norm unfold [Rego.Any.deref, rego_of_attribute])
```

This should, ideally, be a one-liner, but the following fails to prove the goal after exhaustive search on Aesop v4.2.0.
We conclude that there is no push-button approach to writing proofs about our formalization, but white-box automation is at least helpful. It requires some intuition about what sort of theorems would be helpful in proving a particular goal, and our hope is that our compiler would be able to inject some intuition into the output to enable aesop to prove more complicated statements about policies - a major appeal to using aesop is that it is highly configurable by way of custom rules. We expect further experimentation to inform which rules are helpful to produce but simple enough to automatically be derived by aesop or Lean itself.

In addition to automatically deriving certain theorems, we also found aesop helpful is that it will sometimes fail, but have made enough progress that you see that it is trying to prove \texttt{False} with no obvious contradictions among the hypotheses. We were often able to interpret this as an indication that something had gone awry in our specification of the theorem we were attempting to prove – typically a hypothesis we neglected to include.

Generally speaking, our proof amounted to a great deal of rewriting and relatively little novel intuition. We have confidence that many of these sorts of proofs can be automated.

A natural question is whether we can use these results to build high-performance and high-assurance implementations of XACML 3.0 using Rego. We explore performance in section 6.

4.1 Future Work

As noted in the introduction, we introduce a boundary insurmountable in Lean by having our compilers written in OCaml. Now that the general approach to compiling these languages to Lean has been explored, we hope to re-implement the tools in Lean itself, so that we can write proofs of statements about the compilation procedure.

Much more salient to our goal, however, is augmenting the compiler to produce aesop rules that aid in automatically proving theorems derived by the user. We hope to continue to use the formalization to write proofs in hopes that it will help us to identify which aesop rules are worth writing.
5 Case Study: Property Checking

As stated in section \[\textsection 1\] the Margrave tool by Fisler et al. was very influential on this work. To compare our tool with Margrave, we have taken the examples provided therein and run them through our tool to yield shallow embeddings of the policies.\[\textsection 2\] We then devised the following theorems, which roughly correspond to properties 1-3 in \[\textsection 3.1\], as well as the property explored in \[\textsection 3.2\] that “adding TAs should not [affect] external grades in any way.”

\textbf{theorem pr}_1 :
\[
\exists (\text{req : Request}), \quad \text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:subject" "role"}
& \text{none (XmlType.string, "Student"))} \\
\text{(Attribute.mk "[…]\:action" "command"}
& \text{none (XmlType.string, "Assign"))}
\end{cases}
\]
\[
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:resource" "resource-class"}
& \text{none (XmlType.string, "ExternalGrades"))}
\end{cases}
\]
\[
(\text{ElementTree.eval req compiled_college100_element}).fst
\text{== } \text{Decision'.permit := by}
\ldots
\]

\textbf{theorem pr}_2 (\text{req : Request}) :
\[
\text{well_formed request}
\rightarrow \text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:subject" "role"}
& \text{none (XmlType.string, "Faculty"))} \\
\text{(Attribute.mk "[…]\:action" "command"}
& \text{none (XmlType.string, "Assign"))}
\end{cases}
\rightarrow \text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:resource" "resource-class"}
& \text{none (XmlType.string, "InternalGrades"))}
\end{cases}
\rightarrow (\text{ElementTree.eval req compiled_college100_element}).fst
\text{== } \text{Decision'.permit := by}
\ldots
\]

\textbf{theorem pr}_3 :
\[
\exists (\text{user_roles : List Attribute}),
\begin{cases}
\exists (\text{req : Request}), \quad \text{user_roles roles req}
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:action" "command"}
& \text{none (XmlType.string, "Receive"))}
\end{cases}
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:resource" "resource-class"}
& \text{none (XmlType.string, "ExternalGrades"))}
\end{cases}
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:resource" "resource-class"}
& \text{none (XmlType.string, "ExternalGrades"))}
\end{cases}
(\text{ElementTree.eval req compiled_college100_element}).fst
\text{== } \text{Decision'.permit})
\end{cases}
\]
\[
\exists (\text{req : Request}), \quad \text{user_roles roles req}
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:command"}
& \text{none (XmlType.string, "Assign"))}
\end{cases}
\text{req.contains}
\begin{cases}
\text{(Attribute.mk "[…]\:resource-class"}
& \text{none (XmlType.string, "ExternalGrades"))}
\end{cases}
(\text{ElementTree.eval req compiled_college100_element}).fst
\text{== } \text{Decision'.permit})
\]
\ldots

\textbf{theorem pr}_5 :
\[\text{Permission was given to us by one of the authors to do so.} \quad \text{\textsection 4.4 also ends with "[w]e welcome the use of these examples by developers of other tools for reasoning about access-control policies."}\]
∀ (req : Request),
  separation_of_duty req
→ well_formed req
→ req.contains
  (Attribute.mk "[...]:resource "resource-class"
   none (XmlType.string, "ExternalGrades"))
→ (ElementTree.eval req compiled_RPSlist100_element).fst
  == (ElementTree.eval req compiled_RPSlist100_element').fst := by ...

where separation_of_duty is an assertion that the request contains at most one role, and well_formed is an assertion that the request contains some assignment to the attributes we care about (role and resource-class).

Full proofs are available at the repository identified in the introduction[1]. To our knowledge, there is no tool to automatically translate XACML 1.0 to XACML 3.0, so we translated each policy file by hand. The identifier compiled_college100_element corresponds to the college PolicySet in simple-college-example.xml. compiled_RPSlist100_element corresponds to the RPSlist PolicySet in CodeA, and compiled_RPSlist100_element’ corresponds to the same PolicySet but in CodeC. To compile the multi-file policies, we ran the compiler with an argument list containing each constituent filename.

jakob@enseal:~/_build/default/bin/main.exe ~/Research/icse2005-benchmarks/CodeA/*xml
jakob@enseal:~/_build/default/bin/main.exe ~/Research/icse2005-benchmarks/CodeC/*xml

In the output of CodeC, we had to reorder definitions, delete some number of duplicate definitions, and manually add a prime to each definition to disambiguate them from the definitions produced by the first command. These are things that our tool should ostensibly be capable of performing itself, but we have not yet implemented the capability.

pr1 and pr3 were easily proven constructively, thanks in large part to the emphasis on decidability in our semantics[2]. That is, we were able to complete the proof by constructing the Request structure corresponding to the counterexamples produced by Margrave. This demonstrates the first significant difference between this tool and Margrave: Margrave is able to search for and find counterexamples. With our tool, either the user already knows of some counterexample, or they spend time attempting to write a proof of a theorem such as pr2, only to find that it is impossible to prove – indicating that the property being verified is not in fact true. The latter situation was not uncommon while writing this paper. We have not yet identified a situation where Margrave is unable to find a counterexample to an erroneous property and our tool fails to complete the proof of said property.

The approach we took to proving pr2 was bottom-up. We identified the parts of the policy which concerned faculty members assigning internal grades, and proved lemmas about certain attributes being sufficient for the relevant targets matching said attributes. For example:

\[
\text{theorem pr2_lemma1} \quad (\text{req : Request}) : \\
\text{req.contains} \quad \\
\text{(Attribute.mk "[...]:action *= command" \text{Assign")})} \\
→ \text{req.contains} \quad \\
\text{(Attribute.mk "[...]:resource *= resource-class" \text{Assign")})} \\
→ \text{Target.eval} \quad \\
\quad \text{req = MatchResult.is_match := by} \\
\]

\[
\text{theorem pr2_lemma2} \quad (\text{req : Request}) : \\
\text{req.contains} \quad \\
\text{(Attribute.mk "[...]:action *= command" \text{Assign")})} \\
→ \text{req.contains} \\
\]
We proved all of these lemmas by rewriting, but structural induction would have been a reasonable approach as well, given that a Request is just modeled as a list of attributes.

In contrast to the proofs of violation, the computability of policy semantics is not helpful for statements such as these because, in each theorem statement, req is an arbitrary request. Although we found some use in determining the decision a policy renders on a concrete request, where the choice to use Lean really shines is in proving properties over every possible request – our tool is not limited to some bounded domain.

Our approach to proving \texttt{pr}_5 was similar in the sense that we proved lemmas about the targets of child policies comprising the greater policyset. We leveraged more than rewriting, however - it was helpful to do case analysis on whether or not a request contains a particular attribute. This is another area where we saw benefit from modeling with decidable propositions wherever possible: we did not need to resort to classical logic to do this sort of case analysis, because whether or not a \texttt{List} contains a particular element is decidable\textsuperscript{10}.

We omit a comparison between the performance of our tool and that of Margrave because users interact with each tool in drastically different ways; the latency in responding to Margrave queries is completely unlike the latency one may experience while interactively writing a proof in Lean. We will note that we experienced significant latency in certain cases where we unfolded an entire XACML policy element, which is a large part of why we approached these theorems with the bottom-up approach of writing lemmas about constituent policy elements.

As we experienced in \textsuperscript{11}, writing these theorems was onerous. However, our approach of proving properties about particular pieces of a policy seems sufficiently general that we anticipate great improvements to usability if our compiler were to automatically generate rewriting rules - ones which could hypothetically be proven automatically by aesop. It took approximately one human-week to write the proofs outlined in this section.

In this regard, our approach may be more appropriate for when we’re confident that the property holds true. Margrave seems to be the better choice when it comes to mining properties to verify.

\textsuperscript{10}Provided comparison is decidable, which is is in our case.
6 Case Study: Performance of XACML 3.0 Implementations

As noted in section 2, by formalizing the semantics of XACML 3.0 in Lean, we have incidentally built an implementation. We also have an implementation of XACML 3.0 built in Rego due to our work in section 4. It is natural to ask if the performance of either implementation would be favorable to other implementations of XACML 3.0.

We were familiar with two implementations of XACML 3.0 which see use in real-world settings: AuthzForce Community Edition, and Balana. We chose to compare our implementations to the former because we were unable to build Balana on any version of OpenJDK we had available to us. We attempted to generate policies for this section using XACBench [4], but were unable to run the software on the build of OpenJDK we had available to us. So, instead, we evaluated the performance of each implementation on the following policies in the AuthzForce test suite:

- pdp-testutils/src/test/resources/conformance/xacml-3.0-from-2.0-ct/mandatory/IIA001
- pdp-testutils/src/test/resources/conformance/xacml-3.0-from-2.0-ct/mandatory/IIA003
- pdp-testutils/src/test/resources/conformance/xacml-3.0-from-2.0-ct/mandatory/IIA006

We selected these policies because the XACML functions they use are among those which we had actually implemented (namely string-equal and anyURI-equal). We used hyperfine\(^\text{11}\) to run each PDP against the test case.

6.1 Interpreting These Results

The collection of performance data across XACML implementations was an idea that arose relatively late in the course of this project. There was therefore some haste in producing results, and we admit that there is room for improvement with our methods. For now, we caution against drawing conclusions from the comparisons presented here besides that there is potential for competitive verified implementations.

The most glaring practical issue is that, when benchmarking AuthzForce, we leveraged their “CLI” application. In practice, deployments of AuthzForce typically run the software as a server. Our measurements thus include the latency incurred by the initialization of the Java virtual machine that would likely not be observed in practical usage.

Second: our compiler (or, rather, parser in this case) for XACML requests was written in OCaml, not Lean. We collected measurements by compiling the policy and request to produce a Lean script, and executing it with lean -run. I.e., we did not compile the PDP.

Second, we expect that the performance of the compiled Lean implementation would be far more competitive. Compiling the resulting Lean script, we obtain a runtime nearly 1400 times faster – this excludes the cost of transforming the XML request into a format usable by the Lean PDP, so this is necessarily a very generous approximation for speedup, but shows that it may be worth further investigation. Lean does, after all, have an XML parser\(^\text{12}\), so a follow-on effort to build a PDP in Lean that is more self-reliant should not be especially enervating.

Finally: we are somewhat diffident about the Rego benchmarks. We observed some issues in running the Rego implementation, and we believe there may still be some left unresolved. We did not write a compiler

---

\(^{11}\) https://github.com/sharkdp/hyperfine
\(^{12}\) https://github.com/leanprover/lean4/tree/master/src/Lean/Data/Xml
<table>
<thead>
<tr>
<th>AuthzForce</th>
<th>Lean</th>
<th>Open Policy Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runs</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Mean</td>
<td>1.846 s</td>
<td>4.419 s</td>
</tr>
<tr>
<td>Std. Dev.</td>
<td>0.062 s</td>
<td>0.009 s</td>
</tr>
<tr>
<td>Median</td>
<td>1.824 s</td>
<td>4.418 s</td>
</tr>
<tr>
<td>Min</td>
<td>1.768 s</td>
<td>4.406 s</td>
</tr>
<tr>
<td>Max</td>
<td>1.953 s</td>
<td>4.431 s</td>
</tr>
</tbody>
</table>

Table 3: Statistics of wall clock times for evaluating IIA001 across all three implementations.

Figure 1: Whisker plot of wall clock times for evaluating IIA001 across all three implementations.

for translating XACML policies to the JSON format that the Rego implementation expects, so we evaluated it on a small fraction of the conformance suite. While we do not expect further fixes to significantly hinder the runtime, the numbers also do not include the time it takes to serialize the decision to XML. We expect runtimes to increase slightly in a more proper benchmark.

These results were collected on an Intel Core i5-2500 @ 4x 3.7GHz running Gentoo testing with Linux version 6.6.5.

With caveats enumerated, we can now give the results.
<table>
<thead>
<tr>
<th></th>
<th>AuthzForce</th>
<th>Lean</th>
<th>Open Policy Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Runs</strong></td>
<td>10</td>
<td>10</td>
<td>61</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>1.854 s</td>
<td>4.420 s</td>
<td>0.047 s</td>
</tr>
<tr>
<td><strong>Std. Dev.</strong></td>
<td>0.070 s</td>
<td>0.011 s</td>
<td>0.001 s</td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td>1.864 s</td>
<td>4.421 s</td>
<td>0.047 s</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>1.722 s</td>
<td>4.402 s</td>
<td>0.046 s</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>1.928 s</td>
<td>4.434 s</td>
<td>0.049 s</td>
</tr>
</tbody>
</table>

Table 4: Statistics of wall clock times for evaluating IIA003 across all three implementations.

![Whisker plot of wall clock times for evaluating IIA003 across all three implementations.](image)

Figure 2: Whisker plot of wall clock times for evaluating IIA003 across all three implementations.

<table>
<thead>
<tr>
<th></th>
<th>AuthzForce</th>
<th>Lean</th>
<th>Open Policy Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Runs</strong></td>
<td>10</td>
<td>10</td>
<td>61</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td>1.854 s</td>
<td>4.420 s</td>
<td>0.047 s</td>
</tr>
<tr>
<td><strong>Std. Dev.</strong></td>
<td>0.070 s</td>
<td>0.011 s</td>
<td>0.001 s</td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td>1.864 s</td>
<td>4.421 s</td>
<td>0.047 s</td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td>1.722 s</td>
<td>4.402 s</td>
<td>0.046 s</td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td>1.928 s</td>
<td>4.434 s</td>
<td>0.049 s</td>
</tr>
</tbody>
</table>

Table 5: Statistics of wall clock times for evaluating IIA006 across all three implementations.
Future Work

We intend to continue efforts on this project and submit to a conference in the near-term. This section outlines the tasks we have identified as future work to complete before then.

7.1 Refining Semantics

As noted in sections 2.6 and 3.7, there exist quasi-normative test suites available for both languages. It is possible, but nontrivial to turn our Lean embeddings into compliant implementations of both XACML and Rego. To make the effort more tractable, we aim to clearly delineate a subset of each language that can be formalized without significant hassle and implement more standard functions of both languages within these boundaries, such that we can reasonably evaluate our formalizations against both test suites (and ideally instances of policies used in the real world).

We hope that clearly defined boundaries will ease the task of resolving the issues in each formalization we identified in this paper. Still, there may be others which arise in evaluating our formalizations against the languages’ test suites. Therefore, we not only need to implement the remaining features of each language, but also address discrepancies between implementation and formalization, and update our proofs respectively.

7.2 Proof Automation

As we identified in sections 4 and 5, domain-specific proof automation would have to be implemented for this tool to see any practical use. Although we have some ideas about how we might approach this issue – such as having the respective compilers generate statements about policy elements in a bottom-up fashion which are easily dispelled with aesop – we believe there is a need for more experimentation to determine what kind of automation would be best suited to the typical workflow of a user of the tool.
7.3 A Complete Proof of Equivalence

We have written a small part of a proof of equivalence for XACML policies translated to Rego. While this is compelling as a demonstration, it falls short of giving a claim of correctness about the the translation in its entirety. As we noted in section 6, there may be a compelling reason to proceed with a verification of the translation: performance. We first resolve to re-write the tool which translates XACML policies to Rego in Lean, so that we can properly state a correctness claim that quantifies over all policies. From there, we would need to write proofs of equivalence for the policy elements other than attribute designators, ideally using the automation facilities we have developed.

7.4 Evaluating Performance

We noted shortcomings with the methods we used in section 6 which could reasonably be resolved to produce slightly more meaningful results. In particular: (i) collecting data against AuthzForce in a “server” configuration rather than using the CLI application used for its test harness, (ii) writing the request parser in Lean so that we can collect results from a compiled Lean PDP, and (iii) more extensively testing the Rego implementation and writing a parser to transform responses from the Rego PDP to compliant XML. Results which more accurately describe the performance of each implementation would provide justification for the effort involved in proving equivalence between policies specified in two distinct languages: a use-case we developed this tool to support.
8 Conclusions

In this work, we explored the feasibility of using the Lean 4 interactive proof assistant for analyzing and verifying digital policies written in the eXtensible Access Control Markup Language (XACML) 3.0 and Rego. We began by demonstrating how Lean can be used to formalize the semantics of both policy languages, with the benefit that the formalization of XACML is, incidentally, also a usable implementation of the language. Armed with formal models of the respective policy languages, we proceeded to demonstrate Lean’s capability to formally state and prove properties of policies written in each language. We provided an implementation of the XACML 3.0 semantics in Rego and demonstrated how a proof of equivalence would proceed, and we demonstrated how property verification with Lean would look by analyzing the example policies provided in [10] and replicating their analysis results. Finally, we gave preliminary numbers which characterize the performance of our incidental PDP implementation, as well as the implementation of XACML 3.0 in Rego.

We concluded that property verification with Lean, although tractable, is prohibitively difficult without usable proof automation. Properties which are verified almost instantaneously by tools such as Margrave can take human-days or human-weeks when proved by hand by a relatively inexperienced proof engineer, depending on the complexity of the policy. However, automation facilities in Lean such as aesop show great promise, and we expect that improvements to our compiler to automatically state and deduce lemmas (i.e., to dispel impossible cases when rewriting) are likely to make the tool usable - enabling analyses which can be completed in significantly less time.

Furthermore, we concluded that the task of verifying is potentially valuable, as preliminary results indicate that the PDP implementations we developed here have the potential to be competitive with XACML implementations used in practice.
References


