International Journal of Information Security (2019) 18:465–479
https://doi.org/10.1007/s10207-018-0421-5

REGULAR CONTRIBUTION

Analyzing XACML policies using answer set programming

Mohsen Rezvani[1] · David Rajaratnam[2] · Aleksandar Ignjatovic[2] · Maurice Pagnucco[2] · Sanjay Jha[2]

Published online: 26 November 2018

Abstract
With the tremendous growth of Web applications and services, eXtensible Access Control Markup Language (XACML) has been broadly adopted to specify Web access control policies. However, when the policies are large or defined by multiple authorities, it has proved difficult to analyze errors and vulnerabilities in a manual fashion. Recent advances in the answer set programming (ASP) paradigm have provided a powerful problem-solving formalism that is capable of dealing with policy verification. In this paper, we employ ASP to analyze various properties of XACML policies. To this end, we first propose a structured mechanism to translate a XACML policy into an ASP program. Then, we leverage the features of off-the-shelf ASP solvers to specify and verify a wide range of properties of a XACML policy, including redundancy, conflicts, refinement, completeness, reachability, and usefulness. We present an empirical evaluation of the effectiveness and efficiency of a policy analysis tool implemented on top of the Clingo ASP solver. The evaluation results show that our approach is computationally more efficient compared with existing approaches.

Keywords XACML · Policy analysis · Anomaly detection · Answer set programming

1 Introduction

Due to the impressive growth of Web applications, access control policy languages for these applications have received considerable attention, which provides adequate security and privacy support for such applications. The eXtensible Access Control Markup Language (XACML) is an XML-based language standardized by the Organization for the Advancement of Structured Information Standards (OASIS) to express security policies, request context, and response context statements (all written in XML) [1]. XACML has become a widely accepted solution for modeling access control policies for various Web applications as it provides a rich data model for the specification of complex conditions. XACML (particularly version 3.0) enables the use of arbitrary attribute types, hierarchical role-based access control (RBAC), and several rule (policy) combination algorithms to resolve conflicts.

Although XACML is an expressive specification language, it lacks an effective and comprehensive policy[1] analysis framework [6]. The problem becomes more prevalent when the policy is specified by different authorities, making it harder for policy administrators to perceive the overall effect and consequences of the policy execution. For example, it is complicated to manually check essential properties, such as query analysis which determines the accessibility of a resource by a principal [21]. Furthermore, when an administrator updates the policy, understanding the impact of such changes becomes a daunting task. Moreover, policy anomalies, including redundancies and conflicts, remain significant issues that may lead to security leakages through unauthorized access. However, resolving the anomalies through manually changing the XACML policies

✉ Mohsen Rezvani
mrezvani@shahroodut.ac.ir

David Rajaratnam
david.rajaratnam@unsw.edu.au

Aleksandar Ignjatovic
a.ignjatovic@unsw.edu.au

Maurice Pagnucco
m.pagnucco@unsw.edu.au

Sanjay Jha
sanjay.jha@unsw.edu.au

[1] Faculty of Computer Engineering, Shahrood University of Technology, Shahrood, Iran

[2] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

[1] In this paper, the term policy refers to a security policy specified by XACML. Also terms "policy," "security policy," and "XACML policy" are used interchangeably.

is impossible in practice [19]. Thus, a policy verification tool is required to verify various properties at design time.

Answer set programming (ASP) [23] is a declarative programming approach using non-monotonic reasoning aimed toward solving difficult search problems. Due to a high-level expressiveness and providing convenient constructs for application-specific problem representation, it has gained significant attention in recent years [11,13]. ASP has also become an attractive formal language for policy analysis, and several works have already employed ASP for XACML analysis [4,7,22,28,29]. However, none of them has tackled the policy analysis problem by providing a comprehensive solution which takes into account various policy properties such as anomalies, including conflicts and redundancies, refinement, completeness, reachability, and usefulness. Another important challenge in XACML analysis is to consider both XACML 2.0 and 3.0 for checking the policy properties. The newer version of XACML proposes more complex syntax for defining the *target* elements along with additionally combining algorithms for resolving conflicts [1].

To address the above challenges, we propose a comprehensive and structured policy analysis framework by employing ASP as an underlying reasoner. As a basis for formal specification of a XACML policy, we transform the XACML policy into an ASP program. Such a transformation is independent of any policy evaluation mechanism, which helps us to not only support both XACML 2.0 and 3.0 but also provide a compact policy specification. Employing ASP provides ease of specification and offers additional benefits such as optimization in policy verification and the potential for dynamic policy analysis such as checks. Furthermore, we specify the policy evaluation (query matching mechanism) as a separate policy property.

We also demonstrate that our framework is general enough to specify a wide range of policy properties proposed in the literature. To this end, we specify each of these policy properties in an ASP program. Specifying a XACML policy and the properties as ASP programs allows us to employ several efficient ASP solvers, such as Clasp[2] and Smodels[3], to verify the policy against its required properties. As a result of the policy verification, our framework provides detailed evidence in the form of answer sets, which helps the policy administrator to understand the consequences of the policy and helps revise it accordingly. Finally, we implement a prototype of the framework on top of Clingo [15] and conduct experiments using both real-world and synthetically generated XACML policies to evaluate the efficiency and effectiveness of our policy analysis solution. We consider various metrics to compare the efficiency of our method with existing approach for policy translation, grounding, and solving phases. The evaluation results show that our approach is computationally more efficient compared with existing approaches.

In summary, we make the following contributions.

- We propose a new approach for transforming a XACML policy into a set of ASP programs which supports both XACML 2.0 and 3.0.
- We specify the matching operation of various elements in a XACML policy using ASP. Such specification is independent of any specific policy.
- We specify a wide range of policy properties using our policy transformation, such as query analysis, anomaly detection, policy refinement, isomorphism, completeness, and reachability.
- We develop a prototype of our policy analysis framework and evaluate its efficiency and effectiveness using both real-world and synthetically generated XACML policies.

The rest of this paper is organized as follows. Section 2 describes the basic concepts about XACML and ASP. We present our translation of XACML into ASP in Sect. 3. Section 4 shows how to analyze anomalies as well as other properties, such as completeness and reachability of a policy. Section 5 describes our implementation and experimental results. Section 7 presents the related work. Finally, the paper is concluded in Sect. 8.

2 Preliminaries

In this section, we briefly describe the basic concepts of XACML and ASP.

2.1 XACML policy language

Since 2003, XACML has three standard versions, in which the last version (XACML 3.0) was introduced in 2010 [1]. XACML is enforcing authorizations on the resources provided on the Web and specifies how a Web application can access these resources. In this section, we present a summary of the syntax and semantics of XACML 3.0. Figure 1 shows an abstract syntax of XACML 3.0.

There are three main levels in a policy written in XACML: *rule*, *policy*, and *policy set*. A rule is the most elementary unit of a policy and can contain three components: an *effect*, a *target*, and a *condition*. The effect of a rule indicates the consequence of evaluating the rule which can be either *Permit*, *Deny*, or *Indeterminate*. The target of a rule defines the applicability of the rule to a set of access requests. A target consists of a conjunctive sequence of *AnyOf* elements. An AnyOf element contains a disjunctive sequence of *AllOf* elements. An AllOf element includes a conjunctive sequence of *Match* elements. A Match element identifies an entity by a

[2] http://www.cs.uni-potsdam.de/clasp/.

[3] http://www.tcs.hut.fi/Software/smodels/.

XACML Policy Components

```
<PolicySet> :- PolicySetID = [<Target>,≪ PolicySetID* ≫, CombID ]
             | PolicySetID = [<Target>, ≪ PolicyID* ≫, CombID ]
<Policy>    :- PolicyID = [<Target>, ≪ PolicySetID+ ≫ CombID ]
<Rule>      :- RuleID = [ Effect, <Target> , <Condition> ]
<Condition> :- propositional formulae
<Target>    :- Null
             | ⋀ <AnyOf> +
<AnyOf>     :- ⋁ <AllOf> +
<AllOf>     :- ⋀ <Match> +
<Match>     :- AttrType( attribute value )
CombID      :- po | do | fa | ooa
Effect      :- deny | permit
AttrType    :- subject | action | resource | environment
```

XACML Request Component

```
<Request>   :- { Attribute+}
Attribute   :- AttrType( attribute value ) | error(AttrType(attribute value))
             | external state
```

Fig. 1 An abstract syntax of XACML 3.0 [29]

matching attribute value. For example, the target is a predicate over the *subject* (e.g., developer), the *resource* (e.g., codes), and the *action* (e.g., read) of requests to which the rule can be applied. The condition of a rule is a Boolean expression that refines the applicability of the rule beyond its target. If a request satisfies both the target and condition of a rule, the rule's effect is returned as a matching decision; otherwise, *NotApplicable* is returned.

A *policy* combines several rules and comprises three components: a target, a *rule combining algorithm*, and a sequence of rules. Similarly, a *policy set* combines policies and policy sets. A policy set comprises three components: a target, a *policy combining algorithm*, and a sequence of policies or policy sets. The rule (policy) combining algorithm specifies the procedure by which the results of evaluation of rules (policies/policy sets) within a policy (policy set) are combined to evaluate the policy (policy set). XACML supports four common combining algorithms defined as follows:

- *First-applicable* (fa): In this algorithm, each rule (policy/policy set) is evaluated in the order in which it is listed in the policy (policy set). In other words, it returns the decision of the first-applicable rule (policy/policy set).
- *Permit-overrides* (po): In this algorithm, a permit decision has priority over a deny decision. Thus, the algorithm returns permit for a request if there is a rule (policy), which permits the request; returns deny only if all rules (policies/policy sets) deny the request.
- *Deny-overrides* (do): In this algorithm, a deny decision has priority over a permit decision. Thus, the algorithm returns deny for a request if there is a rule (policy/policy set) that denies the request; returns permit only if all rules (policies/policy sets) permit the request.
- *Only-one-applicable* (ooa): This algorithm is defined only for policy sets. In this algorithm, if only one policy (or policy set) is applicable, then its result is returned; if no policy (nor policy set) is applicable, then the result is *NotApplicable*; if more than one policy is applicable, the result is *Indeterminate*.

2.2 XACML 2.0 versus 3.0

There are several enhancements in XACML 3.0 compared with version 2.0 [1]. In XACML 2.0, a target defines a set of attributes to which a rule (policy/policy set) is intended to apply. A target groups attributes of the same attribute type together under elements that reflect such attribute types. The attributes are combined by either disjunctive or conjunctive relationship. The attribute types are organized into *Action*, *Environment*, *Resource*, and *Subject*. As discussed in the previous section, XACML 3.0 removes such grouping of attributes and introduces the AnyOf and AllOf elements that help to define disjunction or conjunction between attribute types. Note that there are other updates in XACML 3.0 which are out of the scope of this paper [1].

2.3 Answer set programming

The basic idea in ASP is to express a search problem by a logic program and then employ an ASP solver to calculate its stable models (or *answer sets*) which encodes the solutions to the problem. In general, ASP solving proceeds in two steps: *grounding* in which a propositional representation of the ASP program is generated, and *solving* in which the stable models (answer sets) are computed from the propositional representation. The final solution is obtained from the resulting answer sets [15].

An ASP program over a set $\mathcal{A}$ of ground atoms consists of a finite set of declarative *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \textbf{not } a_{m+1}, \ldots, \textbf{not } a_n \tag{1}$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ is a ground atom for $0 \leq i \leq n$ and **not** is a symbol for default negation. The set of *literals* consists of all atoms in $\mathcal{A}$ and their default negations. Intuitively, rule (1) means that a_0 must be true if $a_1, \ldots, a_m$ are (provably) true and if $a_{m+1}, \ldots, a_n$ are (possibly) false. Here, a_0 is the *head* of the rule, and right-hand side of the implication symbol constitutes the *body* of the rule. If the body of a rule is empty, the rule is a *fact* written without the implication symbol. A rule with an empty head is an integrity constraint that eliminates unwanted solution candidates described in its body.

Recently, ASP has been significantly extended to support advanced constructs, such as optimization, preference building, and multi-shot solving which can be utilized in policy analysis. There are several off-the-shelf solvers for ASP, such as Clasp, Smodels, and many more, which efficiently compute answer sets. Since the solvers work on variable-free programs, a grounder is needed to compute a ground

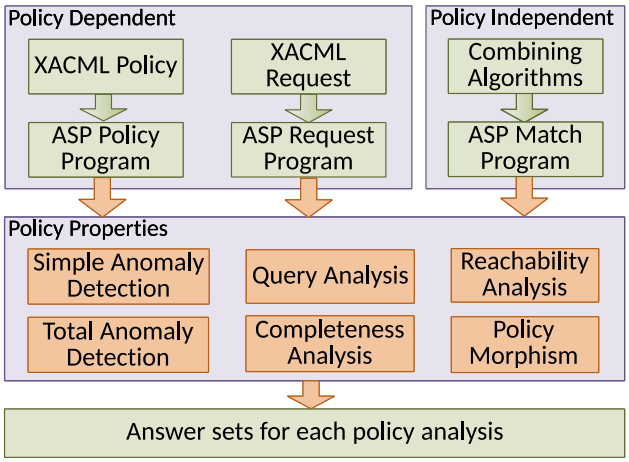

Fig. 2 Our policy analysis framework

(variable-free) program from an ASP program. In this paper, we use Clingo which combines two tools from the Potassco project [16]: Gringo (as a grounder) and clasp (as a solver) into a monolithic system.

3 Mapping XACML into ASP

In this section, we first describe the conceptual framework of our policy analysis. We then explain the details of its components.

3.1 Solution overview

The main idea in our policy analysis framework is to transform both XACML policies and policy properties into ASP programs and then leverage off-the-shelf ASP solvers to verify the properties of the policies. Figure 2 shows our policy analysis framework. The top layer of this framework contains three modules for translating XACML components into ASP programs. We develop a modular translation approach that helps us to specify XACML policies, queries, and combining algorithms in separate ASP programs. As results of the top layer translation, we obtain three ASP programs containing the translation of an XACML policy, a query, and all combining algorithms. Note that the transformation defines a formal semantics of XACML in terms of Answer Set semantics. We plan to extend this research by formally proving that our translation holds the semantics of XACML.

In the middle layer of our framework, we specify each policy property as an ASP program. The property program along with the translation results is sent to an ASP solver to verify the satisfiability of the property on the policy. The final results provide evidence for satisfiability of each property in the form of answer sets generated by the solver.

3.2 Request transformation

In access control systems, an access request generates a *decision request* that contains a set of attributes of the entity making the access request. Thus, we translate a request as a list of facts in which each fact specifies the value of the corresponding attribute type. For example, an access request stated as *John who is a developer and tester, wants to read the reports at 9:00 AM* is translated as

$$\text{subject}(\texttt{developer};\texttt{tester}).\quad \text{resource}(\texttt{reports}).\quad \text{action}(\texttt{read}).\quad \text{time}(9).$$

$$\text{request}(X, Y, Z, T) \leftarrow \text{subject}(X), \text{resource}(Y), \text{action}(Z), \text{time}(T).$$

In the rest of this paper, a request is denoted by variable Q which represents a tuple of variables, such as $Q = (X, Y, Z, T)$ in the above example.

3.3 Policy elements transformation

In this section, we propose a bottom-up approach to transforming a XACML policy into an ASP program, denoted as Π_{xacml}. We bind each component into its upper layer component. To transform a rule r, we first transform the target of the rule. As discussed in the previous section, a target element in XACML 3.0 is a conjunction of disjunctions of conjunctions of match elements. A conjunction of attribute values forms an *AllOf* object, a disjunction of *AllOf* objects forms an *AnyOf* object, and a conjunction of *AnyOf* objects forms an *Target* object.

An important extension in XACML 3.0 is to support various types of match functions, such as *string-equal*, *integer-less-than*, and *integer-greater-than*. Thus, we need to generalize the specification of *AllOf* objects to cover different types of match functions. To this end, we include the specification of the match functions of an *AllOf* object into the specification of the object. A match function inside of an *AllOf* object is used to specify the match of the container *AnyOf* object which is implemented using the Python API in ASP. Thus, we specify *AnyOf* object anyof $_j$ in rule r using ASP as

$$\text{anyof}_j(r.rid) \leftarrow \text{allof}(attr\_val_1, matchf_1 \ldots, attr\_val_n, matchf_n)\ (1 \leq i \leq n),$$

where $r.rid$ is the unique identifier of rule r, n is the number of attribute types defined in the policy, and $match\_fn_i$ is the match function specified for the ith attribute in the *AllOf* object. Note that we assume that each rule, policy, and policy sets have a unique identifier in a XACML policy. An AllOf may match all values for an attribute type in which we set a *wildcard* for the attribute in the ASP program. Moreover, the

disjunction of *AllOf* objects in an *AnyOf* object is specified by repeating the above rule for each *AnyOf* object.

The target of a rule is a conjunction of the AnyOf objects in the rule. Thus, we specify the target object of rule r as

$$\text{target}(r.rid) \leftarrow \text{anyof}_1(r.rid), \ldots, \text{anyof}_m(r.rid),$$

where m is the number of AnyOf objects in rule r. In the case that a target element accepts every request, we define $target(r.rid)$ as a fact term.

A target in XACML 2.0 is a conjunction of groups of attribute values in which each group is corresponding to an attribute type. Each group is specified by a list of facts, one for each attribute value in the group. For example, a group element for subjects with values *manager* and *designer* in rule r is defined in ASP as

$$\text{subjects}(r.rid, \texttt{manager}).$$
$$\text{subjects}(r.rid, \texttt{designer}).$$

The condition component of an XACML rule is a boolean expression which can be evaluated merely using facts and constraints in an ASP program. For example, assume that rule r_1 has no condition and rule r_2 matches in the working hours. The condition components of these two rules are defined as

$$\text{bool\_expr}(\texttt{true}, Q).$$
$$\text{bool\_expr}(working\_hours, Q) \leftarrow T >= 8, T <= 17, \text{time}(T).$$
$$\text{condition}(\texttt{r}_1, \texttt{true}).$$
$$\text{condition}(\texttt{r}_2, working\_hours).$$

In the above translation, we bind each target and condition components into their corresponding rule. Now, for a rule r with identifier $r.rid$, effect $r.effect$, and a policy holder $r.pid$, we define the following ASP rule:

$$\text{rule}(r.rid, r.pid, r.effect).$$

Note that the above ASP rule not only defines the effect of rule r but also binds the rule to its corresponding policyholder. As described, a policy p is represented by an identifier $p.pid$, a policy set holder (parent) with identifier $p.psid$, a list of rules, a target, and a rule combining algorithm $p.comb\_alg$. A policy p is defined by an ASP rule as

$$\text{policy}(p.pid, p.psid, p.comb\_alg).$$

Similarly, a policy set ps is defined by an identifier $ps.psid$, a policy set holder (parent) with identifier $ps.ppsid$, a list of policies and policy sets, a target, and a policy combining algorithm $ps.comb\_alg$. A policy set ps is translated as

$$\text{policyset}(ps.psid, ps.ppsid, ps.comb\_alg).$$

It is worth noting that the target of a policy (policy set) is defined by a similar approach we used for the target of a rule. It is clear that there is a recursive relationship between policies and policy sets. We assume that there is a root policy set with identifier ps_0, with its parent identifier also being ps_0.

3.4 Match transformation

We aim to specify the matching of XACML components independently of any specific policy. In other words, there is no need to update the matching program when the XACML policy is changed. To this end, we define an ASP program, denoted as Π_{match} to specify the matching of various components in a XACML policy against an access request, denoted as Q. We follow a bottom-up methodology to define the matching operations in a XACML policy. As described, a policy (policy set) may be composed of some individual rules (policies). At the lowest level, a rule matches a request if both target and condition of the rule match; consequently, the rule's effect is returned. A target in XACML 3.0 matches a request if all AnyOf objects in the target match. An AnyOf object matches a request if at least one of its AllOf objects matches. Thus, we first define matching functions for an AnyOf construct as

$$\begin{aligned}
\text{match_anyof}(R, Q) \leftarrow\ & request(Q),\\
& \text{anyof}(R, \text{allof}(V_1, F_1, \ldots, V_n, F_n)),\\
& \text{match}(Q.Attr_1, V_1, F_1), \ldots,\\
& \text{match}(Q.Attr_n, V_n, F_n),\\
\text{no_match_anyof}(R, Q) \leftarrow\ & request(Q), \text{anyof}(R, _),\\
& \textbf{not } \text{match_anyof}(R, Q),\\
\text{no_indeter_allof}(R, Q) \leftarrow\ & request(Q),\\
& \text{anyof}(R, \text{allof}(V_1, F_1, \ldots, V_n, F_n)),\\
& \textbf{not } \text{indeter}(Q.Attr_1, V_1, F_1), \ldots,\\
& \textbf{not } \text{indeter}(Q.Attr_n, V_n, F_n),\\
\text{indeter_anyof}(R, Q) \leftarrow\ & request(Q),\\
& \text{no_match_anyof}(R, Q),\\
& \textbf{not } \text{no_indeter_allof}(R, Q),
\end{aligned}$$

where R is a variable corresponding to a rule identifier, $Q = (Attr_1 \ldots, Attr_n)$ is an access request containing a value corresponding to each n attribute types in the XACML policy, and $match(A, V, F)$ is a general function for matching the values A and V based on the F function which considers a wildcard symbol as well. We implemented this function using an external Python function in our experiments, thanks to the Python API developed for the gringo

grounder and clasp solver packages. Note that the Python implementation of this function helps us to support the complex composition operators and various matching functions introduced in XACML 3.0, such as regular expressions and both integer and real linear arithmetic. Moreover, such implementation addresses an intrinsic problem in ASP for expressing arithmetic constraints without generating a large number of clauses. The $match(A, V, F)$ function returns true if attribute A matches value V based on the matching function F, and returns false if it is not matched.

According to the XACML 3.0 specification, if an operational error were to occur while evaluating an attribute value presented in an AllOf object, then the result of the entire expression SHALL be *Indeterminate* [1]. For example, the absence of matching attributes in the request context for any of the attribute designators may result in an enclosing AllOf element to return a value of *Indeterminate*. In such missing attribute scenarios, the policy decision point (PDP) indicates that more information is needed for a definitive decision to be rendered. We use indeter_anyof here to specify an indeterminate result for matching an AnyOf object. Similarly, the indeter(A, V, F) function returns true if an error occurs during the matching of attribute A with value V based on the matching function F; otherwise, it returns false. This function is also implemented using an external Python script.

In the above listing, the second rule, no_match_anyof, shows that there is a non-satisfied AnyOf object. We employed this to specify the matching of a target object as

$$\begin{aligned}
\text{match_target}(T, Q) &\leftarrow request(Q), \text{target}(T),\\
\text{match_target}(T, Q) &\leftarrow request(Q), \text{anyof}(T, _),\\
&\quad\ \textbf{not}\ \text{no_match_anyof}(T, Q),\\
\text{indeter_target}(T, Q) &\leftarrow request(Q), \text{anyof}(T, _),\\
&\quad\ \textbf{not}\ \text{match_target}(T, Q),\\
&\quad\ \text{indeter_anyof}(T, Q),
\end{aligned}$$

where the first match_target term supports the case that a target object accept every request which is called *empty target*. The second term checks all the AnyOf elements within the target construct and the target value shall be matched if all the AnyOf objects specified in the target match values in the request context. Clearly, the body of the second rule specifies a target which includes AnyOf objects matching the request.

In the third rule of the above listing, indeter_target specifies the indeterminate results for matching of a target object. According to the XACML 3.0 specification, if any one of the AnyOf specified in the target is *not matched*, then the target shall be *not matched*. Otherwise, the target shall be *indeterminate* [1]. Note that there is no need to define the *not matched* results. Clearly, a target is *Not Matched* if it is neither matched nor indeterminate. Above listing shows that how we can specify various values for AllOf, AnyOf, and target objects in XACML 3.0. We can simply use a similar method to specify various values for other constructs such as rule, policy, and policy set. In the rest of this paper, we only present our specification for matching results.

In XACML 2.0, the target construct is different from XACML 3.0 and specified by the accepted values of each attribute type separately. Thus, the translation of the target in XACML 2.0 is simpler, as follows:

$$\begin{aligned}
\text{match_target}(T, Q) \leftarrow\ & request(Q),\\
& \text{match_actions}(T, Q.ActReq),\\
& \text{match_resources}(T, Q.ResReq),\\
& \text{match_subjects}(T, Q.SubReq).
\end{aligned}$$

$$\begin{aligned}
\text{match_actions}(T, ActReq) &\leftarrow \text{actions}(T, ActReq).\\
\text{match_resources}(T, ResReq) &\leftarrow \text{resources}(T, ResReq).\\
\text{match_subjects}(T, SubReq) &\leftarrow \text{subjects}(T, SubReq).
\end{aligned}$$

Note that we generalize the matching of a target object in order to reuse the above specification for targets in policies and policy sets.

Now, using the above functions we define the matching of a rule as

$$\begin{aligned}
\text{match_rule}(R, P, E, Q) \leftarrow\ & request(Q), \text{rule}(R, P, E),\\
& \text{match_target}(R, Q),\\
& \text{condition}(R, B), \text{bool_expr}(B, Q),
\end{aligned}$$

where P is the identifier of the policy holding R and E is the matching results.

The matching of a policy (policy set) is defined based on its combining algorithm. The rule (policy) combining algorithm defines a procedure for deciding on a request given the individual matching results of a set of rules (policies or policy sets). We present the matching of a policy based on four combining algorithms explained in Sect. 2.1, and the matching of a policy set can be specified similarly.

In the first-applicable combining algorithm, the result is determined by the matching result of the first rule whose target and condition are matched to the decision request. Thus, we specify the algorithm in ASP as

$$\begin{aligned}
\text{dom_match_rule}(R1, P, E, Q) \leftarrow\ & request(Q), \text{rule}(R1, P, E),\\
& \text{rule}(R2, P, _),\\
& \text{match_rule}(R1, P, E, Q),\\
& \text{match_rule}(R2, P, _, Q),\\
& R2 < R1,\\
\text{match_policy_alg}(P, \texttt{fa}, E, Q) \leftarrow\ & request(Q), \text{rule}(R, P, E),\\
& \text{policy}(P, _, \texttt{fa}),\\
& \text{match_rule}(R, P, E, Q),\\
& \textbf{not}\ \text{dom_match_rule}(R, P, E, Q),
\end{aligned}$$

where the first statement selects the rules that match the request and are dominated by a higher priority rule within the same policy. The second statement defines the effect of a rule that matches the policy and is not dominated.

In the permit-overrides combining algorithm, if there is a permit rule, which matches the request, then the result is Permit; otherwise, the result is obtained by deny rules[4]. We specify the algorithm in ASP as

$$\begin{aligned}\text{match\_policy\_alg}(P, \texttt{po}, \texttt{permit}, Q) \leftarrow\ & request(Q),\\ & \text{policy}(P, \_, \texttt{po}),\\ & \text{match\_rule}(\_, P, \texttt{permit}, Q).\\ \text{match\_policy\_alg}(P, \texttt{po}, \texttt{deny}, Q) \leftarrow\ & request(Q),\\ & \text{policy}(P, \_, \texttt{po}),\\ & \textbf{not } \text{match\_rule}(\_, P, \texttt{permit}, Q),\\ & \text{match\_rule}(\_, P, \texttt{deny}, Q).\end{aligned}$$

Likewise, the deny-overrides combining algorithm can be specified.

In the only-one-applicable combining algorithm, if exactly one policy is matched, the result of the combining algorithm is identified by such a policy. Thus, we specify this algorithm in ASP as

$$\begin{aligned}\text{match\_policyset\_alg}(PS, \texttt{ooa}, E, Q) \leftarrow\ & request(Q),\\ & \text{policy\_set}(PS, \_, \texttt{ooa}),\\ & \text{match\_policy}(\_, PS, E, Q),\\ & 1\{\text{match\_policy}(\_, PS, \_, Q)\}1.\end{aligned}$$

Using above specifications for combining algorithms, the matching of a policy is defined as

$$\begin{aligned}\text{match\_policy}(P, PS, E, Q) \leftarrow\ & request(Q), \text{policy}(P, PS, ALG),\\ & \text{match\_target}(P, Q),\\ & \text{match\_policy\_alg}(P, ALG, E, Q).\end{aligned}$$

We use a similar approach to specify the matching of a policy set. It is worth noting that the specification of the matchings is independent of any specific XACML program as we utilize variables for referring to the policy elements. Moreover, our specification provides detailed witnesses for each matching request. In other words, it reports the identifier of the matched rule, the identifiers of the policies, and the identifiers of the policy sets holding the matched rule in the hierarchical of the XACML policy. This can help the policy administrator to debug the policy for finding any possible error. Moreover, an administrator can employ the above specification to find the results of a query matching over a part of a XACML policy. For example, the administrator may be interested in finding all policy sets matched an access request. Clearly, this can be simply done by filtering the output of the solving process accordingly.

[4] The combining algorithms are more complex, as described in [1], and we simplified them to show the main parts of our specifications.

4 Policy analysis

In this section, we employ our policy translation to analyze various properties proposed in the literature. To this end, each property is first specified as an ASP program, then, it is combined by the programs obtained from the above specifications, and finally, an ASP solver is used to verify the property.

4.1 Query analysis

In Sect. 3.2, we presented an encoding of a simple access request as a conjunction of attribute values. In practice, it is necessary to find policy responses for more complex queries. We define a *query* as a set of simple requests in which some attributes are left as a wildcard. In this section, we show that our framework allows an administrator to define all possible complex queries and verify the policy for such queries.

An important query used for defining other policy properties is to generate all possible access requests. To define this query, we need to determine the domain of each attribute type. Clearly, the domain of an attribute type is the set of all possible values for the attribute. In order to obtain the attribute type, we parse the XACML policy and extract a set of all values existing for each attribute in the policy as the domain of the attribute. Moreover, a policy administrator may need to customize this module of our approach by clearly defining the domain of each attribute type. For example, the following ASP program, denoted as $\Pi_{all\_requests}$, generates all possible request for a XACML policy where the first three lines of the program specify the domain of three attribute type used in this policy.

$$\begin{aligned}&subject\_dom(administrator; developer; programmer).\\ &resource\_dom(codes; reports).\\ &action\_dom(read; write).\\ \\ &request(X, Y, Z) \leftarrow subject\_dom(X), resource\_dom(Y),\\ &\qquad\qquad action\_dom(Z).\end{aligned}$$

Likewise, we verify the policy for a complex query. For example, an administrator may want to check whether the policy forbids any action of a developer on source codes. We define this query by the following ASP program, denoted as Π_{query}:

$$\begin{aligned}request(X, Y, Z) \leftarrow\ & subject(\texttt{developer}),\\ & resource(\texttt{codes}), action\_dom(Z).\\ scenarios(X, Y, Z) \leftarrow\ & match\_policyset(\texttt{ps0}, \_, \texttt{deny}, Q).\end{aligned}$$

Now, an ASP solver can find the scenarios which satisfy the query, by returning the answer sets of program $\Pi_{xacml} \cup \Pi_{match} \cup \Pi_{query}$.

4.2 Policy anomaly detection

Al-Shaer et al. [5] introduced four types of pairwise anomalies among rules in a network policy: *Shadowing*, *Correlation*, *Generalization*, and *Redundancy*. Basi et al. [9] also classified the anomalies into two categories: *conflict* where a request is matched with multiple rules with conflicting actions, and *suboptimality*

where there is a rule such that its removal has no effect on the policy.

Since both policies and policy sets match based on the combining algorithms, a conflict is automatically resolved using these combining algorithms. However, reporting the conflicts can help the administrator to discover the hidden errors in the policy. Moreover, the conflicts in a XACML policy may lead to several security problems such as safety problem (where a user can access to resources which truly forbidden for the user in the policy) [19,32] and attribute hiding attacks (where a user is able to obtain more favorable authorization decision by hiding some of her attributes) [12,33].

The response time of an access request evaluation largely depends on the number of rules, policies, and policy sets in a XACML policy [17,25]. Redundancy in a policy can adversely affect the efficiency of the policy evaluation as it increases the policy length. The complex syntax of XACML policies raises the chance of redundancy among rules, policies, and policy sets. Moreover, detection and removal of such redundancies are probably very complicated due to the fact that one rule (policy or policy set) may overlap with multiple other rules (policies or policy sets).

In this section, we employ our framework to define two pairwise anomalies in an XACML policy: *conflict* and *redundancy*. We concentrate on intra-policy anomalies where there is an anomaly between two rules within the same policy. Similarly, we extend our solution to analyze inter-policy anomalies. Rule r_1 is conflicting with rule r_2 if there are some requests that match both rules while they have different effects. Rule r_1 is redundant with rule r_2, if every request that could match r_1 is matched by r_2. Accordingly, the definition of conflicting and redundancy is based on two pairwise rule relationships *subset* and *overlap*, respectively. Now using above definitions, we define the conflicting and redundancy among rules within a policy as

$$
\begin{aligned}
\text{no\_subset\_rule}(R1, R2) \leftarrow\ & \text{rule}(R1, P, _), \text{rule}(R2, P, _),\\
& \text{request}(Q),\\
& R1 \neq R2, \text{match\_rule}(R1, \_, \_, Q),\\
& \textbf{not } \text{match\_rule}(R2, \_, \_, Q),\\
\text{redundancy}(R1, R2) \leftarrow\ & \text{rule}(R1, P, E1), \text{rule}(R2, P, E2),\\
& R1 > R2,\ E1 = E2,\\
& \textbf{not } \text{no\_subset\_rule}(R1, R2),\\
\text{conflict}(R1, R2) \leftarrow\ & \text{rule}(R1, P, E1), \text{rule}(R2, P, E2),\\
& \text{request}(Q), R1 \neq R2, E1 \neq E2,\\
& \text{match\_rule}(R1, \_, \_, Q),\\
& \text{match\_rule}(R2, \_, \_, Q),
\end{aligned}
$$

where term no_subset_rule defines pairwise non-inclusive rules within a policy. This helps us to detect inclusive rules which forms the redundancy anomaly. Solving a combination of the above program with $\Pi_{xacml} \cup \Pi_{match} \cup \Pi_{all\_requests}$ generates all existing pairwise conflicts and redundancies in the XACML program. Note that the pairwise redundancy and conflict between policies (policy sets) are specified by a similar method.

4.3 Total redundancy and reachability

In the previous section, we defined mutual (simple) anomalies between two rules within a policy. More generally, an anomaly can occur between a rule and a set of other rules, called *total anomaly* [8,30]. We define that a rule is *totally redundant* if a subset of higher priority rules covers this rule. In other words, a rule is totally redundant if every request matched by this rules is also matched by other rules in the policy. We formulate this definition in ASP as

$$
\begin{aligned}
\text{match\_by\_others}(R, Q) \leftarrow\ & \text{rule}(R, P, \_), \text{rule}(R2, P, \_),\\
& \text{request}(Q), R > R2,\\
& \text{match\_rule}(R, \_, \_, Q),\\
& \text{match\_rule}(R2, \_, \_, Q).\\
\textit{no\_total\_redundancy}(R) \leftarrow\ & \text{rule}(R, \_, \_), \text{request}(Q),\\
& \text{match\_rule}(R, \_, \_, Q),\\
& \textbf{not } \text{match\_by\_others}(R, Q).\\
\text{total\_redundancy}(R) \leftarrow\ & \text{rule}(R, \_, \_),\\
& \textbf{not } \text{no\_total\_redundancy}(R).
\end{aligned}
$$

Now we generalize such redundancy by defining *unreachability*. A rule is unreachable if there is no request matched by it. Clearly, a redundant rule is unreachable, but an unreachable rule may not be redundant as we defined the redundancy in the scope of a policy. We specify an unreachable rule in ASP as

$$
\begin{aligned}
\text{reachable\_rule}(R) \leftarrow\ & \text{rule}(R, \_, \_), \text{request}(Q),\\
& \text{match\_policyset}(\mathtt{ps}_0, \_, \_, Q),\\
\text{unreachable\_rule}(R) \leftarrow\ & \text{rule}(R, \_, \_), \textbf{not } \text{reachable\_rule}(R).
\end{aligned}
$$

We use a similar method to detect unreachable policies and policy sets. Removing an unreachable rule (policy and policy set) has no effect on the semantics of a policy. Thus, the discovery of such rules (policies and policy sets) helps the administrator to remove them and improve the efficiency of the policy analysis.

4.4 Usefulness and completeness

A rule (policy/policy set) is *useful* if there is a request matched by such rule (policy/policy set). For example, if the conjunction of the condition and target components of a rule is a *contradiction*, such a rule never matches any request and is *useless*. Note that there is a subtle difference between uselessness and unreachability. An unreachable rule might be useful but covered by other rules. We formally define a useful and useless rule as

$$
\begin{aligned}
\text{useful\_rule}(R) \leftarrow\ & \text{rule}(R, \_, \_), \text{request}(Q),\\
& \text{match\_rule}(R, \_, \_, Q).\\
\text{useless\_rule}(R) \leftarrow\ & \text{rule}(R, \_, \_), \textbf{not } \text{useful\_rule}(R).
\end{aligned}
$$

A XACML policy is *complete* if it matches every request generated by $\Pi_{all\_requests}$. Similarly, a XACML policy is *incomplete* if there is a request which is not matched by the policy. An incomplete policy might lead to a security problem when an attacker can compromise such incompleteness to gain unauthorized access

[28]. For example, an attacker can obtain an access by generating a query such that there is no response to the query in the policy. Consequently, an application with a default policy of *Permit* will allow the attacker to access the system. Thus, although the completeness property of a policy seems to be an excessive requirement, a policymaker needs to know the incompleteness of the policy. We express the completeness and incompleteness properties as

$$\begin{aligned} \text{incomplete}(Q) &\leftarrow \text{request}(Q), \textbf{not}\ \text{match\_policyset}(\mathtt{ps}_0, \_, Q), \\ \text{complete} &\leftarrow \textbf{not}\ \text{incomplete}(\_), \end{aligned}$$

where term $incomplete(Q)$ presents the witnesses of incompleteness as a list of requests which are not matched by the XACML policy.

4.5 Policy subsumption, morphism, and disjointness

Hughes and Bultan [20] define a partial ordering relation among XACML policies, called *subsumption*. Accordingly, XACML policy p_1 subsumes policy p_2 if and only if policy p_2 always returns a decision identical to the decision provided by p_2 [33]. We formally express the subsumption relation between two policies specified using our framework as

$$\begin{aligned} \text{no\_subsume}(P1, P2, Q) \leftarrow\ & \text{request}(Q), P1 \neq P2, E1 \neq E2, \\ & \text{match\_policyset}(P1.Root, \_, E1, Q), \\ & \text{match\_policyset}(P2.Root, \_, E2, Q), \\ subsume(P1, P2) \leftarrow\ & P1 \neq P2, \\ & \textbf{not}\ \text{no\_subsume}(P1, P2, \_), \end{aligned}$$

where terms $P1.Root$ and $P2.Root$ defines the identifiers of the root policy set in p_1 and p_2, respectively. Also, if there is no subsumption relation between two policies, the term no_subsume shows all requests for which the decisions returned by p_1 and p_2 are different.

Two XACML policies are isomorphic if and only if they return identical decision for every request. The main difference between isomorphic and subsumption is that two isomorphic policies must have the same coverage, while in subsumption, the coverage of the first policy is a subset of the second. Thus,

$$\text{isomorphic}(P1, P2) \leftarrow \text{subsume}(P1, P2), \text{subsume}(P2, P1).$$

Two XACML policies are disjoint if and only if they always return different decisions for every access request. In other words, two policies are disjoint if and only if there is no request such that the policies return an identical decision for such request. Likewise the previous properties, we first specify the negation of the property and then if there is no evidence for proving its negation, the property is satisfied. We specify the disjointness between two XACML policies as

$$\begin{aligned} \text{no\_disjoint}(P1, P2, Q) \leftarrow\ & \text{request}(Q), P1 \neq P2, E1 = E2, \\ & \text{match\_policyset}(P1.Root, \_, E1, Q), \\ & \text{match\_policyset}(P2.Root, \_, E2, Q). \\ disjoint(P1, P2) \leftarrow\ & P1 \neq P2, \\ & \textbf{not}\ \text{no\_disjoint}(P1, P2, \_). \end{aligned}$$

Note that we decouple the specifications of a policy from the matching procedure. Thus, in order to check the subsumption, isomorphism, and disjointness of two XACML policies, we first transform the policies into ASP using the method described in Sect. 3.3 to obtain two programs Π_{xacml1} and Π_{xacml2}. Now we solve a combination of the above definitions with $\Pi_{xacml1} \cup \Pi_{xacml2} \cup \Pi_{match} \cup \Pi_{all\_requests}$ to verify the property. Moreover, for solving subsumption, isomorphism, and disjointness programs, we must assign a different set of identifiers to the rules/policies/policy sets within two XACML policies during the policy transformation.

4.6 Change impact detection

Once an access control policy is defined for an organization, administrators often need to update the policy because of new requirements. Thus, administrators need to analyze any unintentional effect of the changes in the policy. The main objective of a change impact detection problem is to present all differences of the decisions returned by the original policy after applying the changes.

We convert the change impact detection problem into the subsumption problem discussed in the previous section. To this end, we generate two XACML policies, the original without the changes and the new policy including the changes. Solving the subsumption problem for these two policies reports all requests that the decisions of the original policy have been changed for them.

5 Implementation and evaluation

In this section, we detail the steps taken to implement the proposed framework and evaluate the performance of our approach for analyzing various properties in XACML policies.

5.1 Prototype implementation

We have implemented our policy analysis framework in Java. As shown in Fig. 2, the framework consists of two main components: policy translation and property analysis. The translation module takes either a XACML policy (supports both versions 2.0 and 3.0) or a request context in XML format as an input and generates the corresponding ASP programs, as described in Sect. 3. This module utilizes the Java Architecture for XML Binding (JAXB) API[5] to parse the policies and construct the corresponding ASP programs. The property analysis module takes the ASP programs obtained from the translation module and utilizes Clingo to verify the properties described in Sect. 4. We specified all of these properties with around 70 ASP rules in 300+ lines of ASP code. The result of the verification is shown as a set of witnesses in the form of answer sets.

[5] https://jaxb.java.net/.

5.2 Experimental environment

We assume that a policy administrator employs our tool to validate the properties for a policy update. The main objective of the experiments is to investigate the efficiency and effectiveness of our tool for validating the policy properties. We divide the properties described in Sect. 4 in six categories: (1) **query matching** which checks the policy against a simple request specified by values for attributes; (2) **scenario finding** which checks the policy against a complex query; (3) **intra-policy anomaly** which checks the anomalies for a rule with other rules within a policy. These anomalies consist of simple shadow, simple redundancy, correlation, generalization, total redundancy, usefulness, and reachability of a rule; (4) **inter-policy anomaly** which checks these anomalies for a policy with other policies within a policy set; (5) **refinement** which checks the subsumption and isomorphism between two XACML policies. Note that the change impact detection property can be evaluated using the refinement category as such property can be simply converted into the subsumption (as we discussed in the previous section.); and (6) **redundancy** which reports all rules totally covered by others in a XACML policy.

In all experiments, we evaluate the processing time of grounding and solving for checking the above properties over each XACML policy. We also conduct our experiments by using both real-world datasets and synthetic datasets generated with parameters similar to the real-world dataset. All the experiments were conducted on UNSW Leonardi Cluster[6] in which each node consists of a 2.30 GHz processor core with 8 GB memory running CentOS Linux. We also utilize Clingo 4.5.4 for both the grounding and solving the ASP programs.

5.3 Efficiency of policy analysis

In order to evaluate the efficiency of our policy analysis approach, we generate synthetic policies based on a real-world security policy called *Continue-a*, used in [14] and designed for a real-world Web application for conference management. It consists of 298 rules, 266 policies, and 111 policy sets. We also conduct all the experiments over both XACML 2.0 and 3.0. Note that the Continue-a policies for two versions are not identical as we obtained them from different sources. We generate the synthetic policies by randomly removing and replicating the rules, policies, and policy sets within the original policy. Each experiment is repeated over 100 randomly generated policies, and then, the results are averaged. For each policy, the translation module in our tool generates the corresponding ASP programs for both the policy and all the six properties. Since there are some parameters in the properties, such as attribute values, rule number, or policy number, we again repeat the experiment 100 times to randomly assign values to these parameters and average the results.

Figure 3a, b, respectively, shows the processing time of the matching properties for XACML 2.0 and 3.0. As one can see, in both versions our approach performs the query matching faster than the scenario finding. This is due to the fact that the number of requests in a scenario is more than the number in a simple query. Note that the scenario finding reports every request which matches

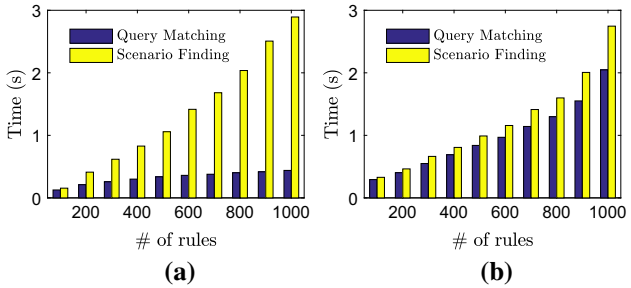

Fig. 3 Performance of matching a simple and complex query. **a** XACML V2. **b** XACML V3

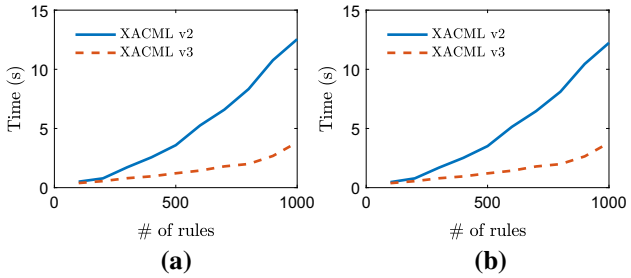

Fig. 4 Performance of detecting intra- and inter-policy anomalies. **a** Intra-policy anomalies. **b** Inter-policy anomalies

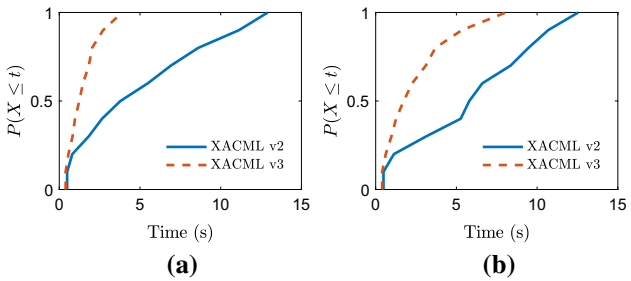

Fig. 5 Performance of policy refinement and total redundancy. **a** Policy refinement. **b** Total redundancy

the scenario. Thus, the tool needs to check all requests specified in the scenario. Comparing two plots in Fig. 3 shows that the query matching in XACML 2.0 is significantly faster than 3.0. This can be explained by the fact that the target structure in XACML 3.0 is more complex than the construct in 2.0 which allows an administrator to define its own attribute types. The figure also shows that the processing time of the scenario finding in XACML 2.0 raises more sharply than the processing time in 3.0. This is because the fact that the number of attribute values in Continue-a 2.0 is around double that of the number in the version 3.0. This considerably raises the number of requests within a scenario which leads to a significant increase in the number of atoms generated during the grounding process. For example, the number of atoms obtained for a policy with 900 rules in version 2.0 is several orders of magnitude more than the atoms for the same policy in version 3.0.

Figure 4a, b illustrates the processing time of detecting intra and inter-policy anomalies, respectively. The results show that the processing time of the anomaly detection for XACML 2.0 signifi-

[6] http://leonardi.unsw.wikispaces.net/.

cantly rises as the number of rules increases in the policy. This is an artifact of a large number of attribute values in this version. Note that increasing the number of attribute values exponentially raises the number of requests generated in the $\Pi_{all\_requests}$ program.

Figure 5a, b shows the results of checking the policy refinement and total rule redundancy properties, respectively. Since the trend is similar to Fig. 4, we instead show the cumulative distribution function (CDF) when the policy length increases.

5.4 Effectiveness of policy analysis

In this section, we evaluate the effectiveness of our approach for detecting unnecessary rules, policies, and policy sets in several real-world XACML policies: *FreeCS*, *Pluto*, *Continue-a*, and *Continue-b* used in [4], *KMarket* [3], and *AU2EU* [2]. A rule (policy or policy set) is unnecessary if it is shadowed, redundant, useless, or unreachable, as described in Sect. 4. Clearly, an unnecessary rule (policy or policy set) can be removed from a policy without changing the intention of the policy. Note that discovery and removal of the unnecessary elements can directly improve the performance of policy evaluation as the response time of evaluating an access request significantly depends on the policy length, as shown in Fig. 3. Another objective of the effectiveness analysis in this section is to detect isomorphic policy sets within a XACML policy. This can help an administrator to detect some hidden errors including the replicated segments.

Table 1 reports the number of unnecessary rules (policies or policy sets) and isomorphisms in the real-world policies. The results show that around one-third of the rules and half of the policies are unreachable in both Continue-a and Continue-b. These unreachable policies are well distributed within the policy sets as there is no unreachable policy set in these policies. The existence of such large number of unreachable rules and policies is often due to an administrator inserting general rules in the middle of the policy to implement some update requirement. The results also report 16 shadowed policies in Continue-b and eight redundant rules in both Continue-a and Continue-b. In order to resolve the unreachability, redundancy, and shadow anomalies, an administrator can simply remove the elements. Note that our policy analysis tool reports these anomalies along with all details about the elements involved to facilitate the resolution procedure.

In order to evaluate the number of Isomorphic policy sets, we check the isomorphic property (specified in Sect. 4.5) for all possible permutation of the XACML policy with two policy sets. For example, since the number of policy sets in Continue-a (and similarly in Continue-b) is 111, we need to check the isomorphic property for $111 * 110 = 12210$ permutations for this policy. Table 1 reports that around 28% (one can see in table that there are 3486 isomorphic pairs and the isomorphic percentage is $3486/12210 * 100 = 28.5\%$) of the policy sets in Continue-a and Continue-b are mutually isomorphic. This shows that there are many replicated segments in both policies and administrator can reduce the policies' lengths by removing the replicated segments. Finally, the results show that these two policies are incomplete as they have no decision for 72 requests. An administrator can take the advantage of the completeness analysis to improve the policy by defining a clear response for these 72 access requests and consequently complete the policy.

5.5 Efficiency of policy transformation

In this section, we compare the efficiency of our policy translation against two other approaches proposed for translating a XACML policy into ASP. There are several attempts to utilize the expressiveness of ASP for XACML analysis [4,7,22,28,29]. Lee et al. [22] extend the translation method proposed by [4] for XACML 3.0, and they have used a very similar approach for the policy translation. The translation method proposed in [7,28] is technically similar to [29], and the authors only extend the translation for a few policy properties. Therefore, we compare our policy translation method against two methods proposed in [22,29], and we call them *Ramli* and *Lee*, respectively. To this end, we employed the *XACML2ASP*[7] tool which is an implementation of *Lee* published by authors. For the *Ramli* policy translation, there is no implementation proposed by authors. Thus, we have implemented the policy translation and used our implementation in the experiments.

In order to compare the performance of our policy translation against Ramli and Lee, we generate synthetic XACML policies using a similar method presented in Sect. 5.3. After creating each policy, we utilize these three policy translation methods to obtain the corresponding ASP programs for the XACML policy. We then use Clingo to ground and solve the ASP programs. The efficiency metrics we collected during the grounding and solving contain all the time parameters reported by Clingo and some metrics related to the size of the grounded program including the number of atoms and the number of rules. We also measured the translation time and the number of lines in the ASP program generated for each policy translation method.

Figure 6 reports the comparison between the performance results of our algorithm against *Lee* and *Ramli* based on various performance metrics extracted during policy translation, ASP grounding, and solving. As we mentioned, the first step of this experiment is to translate a XACML policy into an ASP program. The performance of this step is evaluated based on the lines of code in the ASP program and the elapsed time of the translation. The performance results of this step based on this two metrics are shown in Fig. 6a, b, respectively. As one can see in the results, our method outperforms both *Lee* and *Ramli* in terms of translation efficiency. Moreover, *Ramli* generates ASP programs with a huge number of lines comparing to other methods. This is because the fact that *Ramli* employs an inefficient method to translate the first-applicable combining algorithm. More specifically, such translation for the first-applicable rule combining algorithm in each policy needs n rules and $O(n)$ terms in the body of each rule, where n is the number of rules in the policy. This generates an enormous number of rules in ASP programs as there are many first-applicable terms in the original XACML policy. We can also observe in Fig. 6b that the translation time for *Lee* dramatically increases as the number of rules increases in the XACML policies. This is due to the fact that *Lee* employs a policy translation algorithm with a quadratic time complexity while both *Ramli* and our translation algorithm are in linear time complexity. It is worth noting that we cannot translate any policy with more than 5000 rules using the *XACML2ASP* tool for *Lee*. Thus, we limit our experiments to XACML policies with less than this number of rules.

[7] http://reasoning.eas.asu.edu/xacml2asp/.

Table 1 Effectiveness of our policy analysis over real-world XACML policies

Base policy	Rules				Policies			Policy sets			
	#	SH	RE	UR	#	RE	UR	#	UR	IS	IC
FreeCS	2	1	0	1	1	0	0	1	0	0	×
KMarket	5	4	6	1	1	0	0	1	0	0	×
AU2EU	6	0	6	4	3	1	1	1	0	0	×
Pluto	21	0	0	0	1	0	0	1	0	0	×
Continue-a	298	0	8	197	266	10	165	111	0	3486	72
Continue-b	306	16	8	205	266	10	165	111	0	3486	72

The number of rules (policies, policy sets) is indicated as #, shadow anomalies (SH), redundancies (RE), unreachable elements (UR), isomorphisms (IS), and incompleteness (IC)

In the next step of this experiment, we used Clingo to ground and solve the ASP programs generated in the first phase (policy translation step). The results reported in the remaining plots in Fig. 6c–i are extracted from the output of the Clingo tool when we ground and solve the ASP programs. As one can see in the figures, our method outperforms both *Lee* and *Ramli* in all efficiency metrics for grounding and solving except for the reading and ground times where our method and *Lee* provides very similar performance values. It is clearly shown that *Ramli* presents an inefficient grounding and solving comparing other methods. This can be explained by the fact that *Ramli* uses a very inefficient policy translation, particularly for the first-applicable combining algorithm. Such issue is also illustrated in Fig. 6a where the lines of code in the ASP programs generated by *Ramli* dramatically increase as the number of rules in the policies raises.

6 Discussion

An important benefit of the proposed approach is to specify a wide range of policy properties. However, some security properties, such safety and availability, need a dynamic analysis framework. An interesting research issue is to investigate the possibility of applying our method for dynamical analysis of XACML policy properties. Since our approach provides a modular specification by decoupling the policy specification from policy evaluation, there is a potential to supplement our approach for such analysis.

As shown in Sect. 4.2, our approach can detect various types of anomalies including redundancies and conflicts by generating all possible pairwise anomalies in a XACML program. However, an administrator still needs a mechanism to resolve such anomalies. One idea to provide such mechanism is to leverage the expressiveness of the ASP language to obtain the optimum and anomaly-free policy from an anomalous XACML policy. It is clear that defining such an optimization problem is the main challenge in developing such a mechanism that we leave for future work.

An intrinsic problem in ASP is to express arithmetic constraints which can lead to generating a large number of clauses. We addressed this challenge by resorting to an external Python implementation. More specifically, we employed a general function implemented in Python for supporting wildcard symbols, the complex composition operators, and various matching functions introduced in XACML 3.0, such as regular expressions and both integer and real linear arithmetic (as explained in Sect. 3.4). One can argue that using such external implementation reduces the formality of our specification and verification model. It is to be noted that such an issue is only in the case a non-deterministic external function is employed in the specification [16]. Moreover, all of our external functions implemented in Python are deterministic in which when each function is called multiple times with the same arguments during grounding, it returns the same values.

7 Related work

Policy analysis including anomaly detection in traditional access control policies, such as firewalls, has been extensively studied in the research community [5,18,34]. Al-Shaer and Hamed present a set of algorithms to discover simple pairwise anomalies in centralized and distributed Firewall rules [5]. Inconsistencies and inefficiencies among multiple rules are treated in [30,34]. Recently we proposed an anomaly-free network policy composition for software-defined networking (SDN) [31]. Due to the complexity of the syntax of the XACML policies, directly applying these approaches to XACML is not suitable [19].

Many research efforts have been devoted to XACML policy analysis [10,14,19,24,26,27]. Fisler et al. [14] proposed a policy analysis tool, called Margrave which represents the policies by multi-terminal binary decision diagrams (MTBDDs) to check two properties: change impact analysis and query evaluation. XAnalyzer [19] is another BDD-based solution to detect and resolve policy anomalies including redundancy and conflicts. Bauer et al. [10] proposed a data mining approach to detecting and resolving inconsistencies in access control policies. EXAM [24] is a policy analyzer tool which integrates the SAT-solver-based and MTBDD-based approaches to checking XACML policy properties such as query analysis and policy similarity. Compared with our approach, these solutions are limited to a subset of policy properties. Moreover, the BDD-based approaches are limited to XACML 2.0, while our approach supports both versions 2.0 and 3.0.

There are several works presenting XACML policy analysis by using different types of formal languages. Turkmen et al. [33] employed satisfiability modulo theories (SMT) as the reasoning mechanism to check various properties in policies. Kolovski et al. [21] specified XACML using description logics to analyze

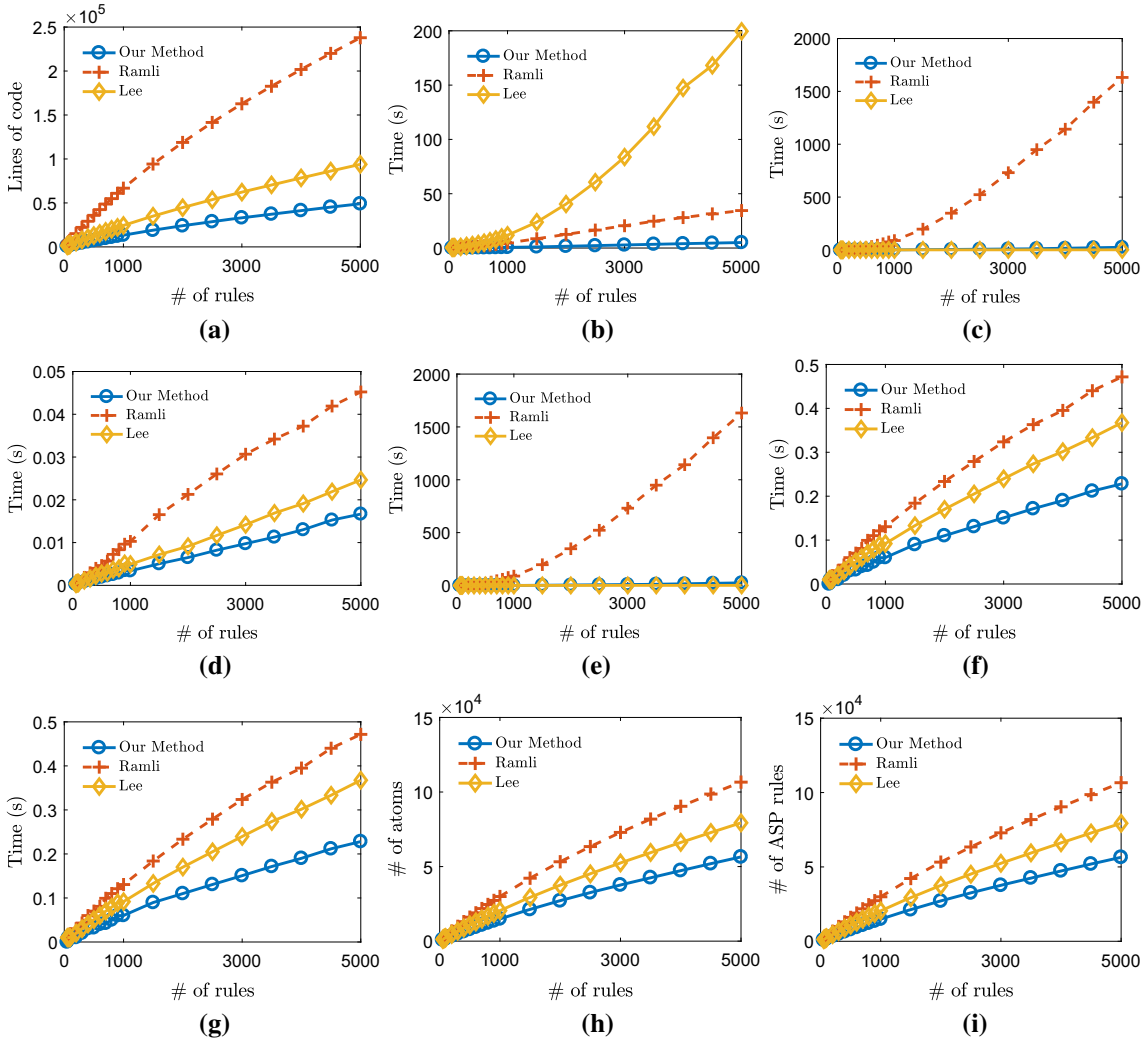

Fig. 6 Performance of the policy translation methods. **a** ASP lines of code. **b** Translation time. **c** Reading time. **d** Preprocessing time. **e** Grounding time. **f** Solving time. **g** Unsat time. **h** Number of atoms. **i** Number of rules

properties such as policy comparison, verification, and querying. Arkoudas et al. [6] proposed an SMT-based framework to specify access control policies, such as XACML. The framework checks various properties in a policy including consistency, coverage, observational equivalence, and change impact. Although these proposals show the potentiality of SMT for XACML analysis, we employed ASP which provides a higher-level mechanism for policy specification and verification.

There are several attempts to utilize the expressiveness of ASP for XACML analysis [4,7,22,28,29]. The pioneering work proposed by Ahn et al. [4] translates a XACML policy to an ASP program. The work is restricted to XACML 2.0 and a few policy properties such as query analysis. Ramli et al. [29] proposed a formal representation of XACML 3.0 in ASP. They also extended this work to analyze policies including completeness, conflicting, and reachability [28]. Ayed et al. [7] provided a concrete implementation of XACML analysis using ASP. Lee et al. [22] also formulated XACML 3.0 in ASP and used ASP solvers to perform automated reasoning about XACML policies. This work is restricted to only query analysis. Our approach is different from these works in four aspects. First, we provided a modular specification for both XACML 2.0 and 3.0, in which the policy specification is decoupled from policy evaluation. Second, we proposed a compact policy transformation approach which largely improves the efficiency of the policy analysis. For example, the translation method proposed in [7,22,28,29] for the first-applicable rule combining algorithm in each policy needs n rules and $O(n)$ terms in the body of each rule, where n is the number of rules in the policy. However, our translation approach needs only two rules with exactly six terms in the body of each rule, as shown in Sect. 3.3. This compact translation provides a promising efficiency for our policy analysis, as presented in Sect. 5. Third, we specified a wide range of policy properties proposed in the literature at the top of our framework. Finally, a concrete implementation of the proposed solution and an extensive evaluation demonstrate the efficiency and effectiveness of our method. Thus, our method makes an additional step to show the applicability of ASP for analyzing the security properties of XACML policies.

8 Conclusions

In this paper, we proposed a novel and formal framework to analyze XACML policies. We first presented a transformation of a policy into an ASP program. Then, we specified a wide range of policy properties using ASP which helps us to verify the properties against a XACML policy specified in ASP. We also showed that our framework is general enough to express various properties in policies, such as query analysis, anomaly (redundancy and conflict) detection, reachability, usefulness, completeness, subsumption, morphism, and disjointness.

For future work, we plan to extend our approach to resolving anomalies in a policy. This can find an optimal subset of the policy which is anomaly-free while keeps the semantics of the original policy. We also would like to extend our framework for analyzing dynamic properties, such as safety and availability. Finally, we plan to construct a formal semantics of XACML 3.0 and prove that our transformation defines the XACML semantics in terms of ASP semantics.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. eXtensible Access Control Markup Language (XACML) Version 3.0 (2013). http://docs.oasis-open.org/xacml/30/xacml-30-core-spec-os-enpdf. Accessed Sept 2018
2. AU2EU: Authentication and authorisation for entrusted unions (2015). http://www.au2eu.eu/. Accessed Sept 2018
3. WSO2 balana: The open source XACML 3.0 implementation (2015). http://xacmlinfo.org/category/balana/. Accessed Sept 2018
4. Ahn, G.J., Hu, H., Lee, J., Meng, Y.: Representing and reasoning about web access control policies. In: Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference, COMPSAC '10, pp. 137–146 (2010)
5. Al-Shaer, E.S., Hamed, H.H.: Discovery of policy anomalies in distributed firewalls. In: INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 4, pp. 2605–2616 (2004)
6. Arkoudas, K., Chadha, R., Chiang, J.: Sophisticated access control via SMT and logical frameworks. ACM Trans. Inf. Syst. Secur. **16**(4), 17:1–17:31 (2014)
7. Ayed, D., Lepareux, M.N., Martins, C.: Analysis of XACML policies with ASP. In: 7th International Conference on New Technologies, Mobility and Security (NTMS) (2015)
8. Basile, C., Cappadonia, A., Lioy, A.: Geometric interpretation of policy specification. In: Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks, POLICY '08, pp. 78–81 (2008)
9. Basile, C., Cappadonia, A., Lioy, A.: Network-level access control policy analysis and transformation. IEEE/ACM Trans. Netw. **20**(4), 985–998 (2012)
10. Bauer, L., Garriss, S., Reiter, M.K.: Detecting and resolving policy misconfigurations in access-control systems. ACM Trans. Inf. Syst. Secur. (TISSEC) **14**(1), 2 (2011)
11. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM **54**(12), 92–103 (2011)
12. Crampton, J., Morisset, C.: PTaCL: a language for attribute-based access control in open systems. In: International Conference on Principles of Security and Trust, pp. 390–409. Springer (2012)
13. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: a primer. In: Reasoning Web. Semantic Technologies for Information Systems, Lecture Notes in Computer Science, vol. 5689, pp. 40–110 (2009)
14. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pp. 196–205 (2005)
15. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, San Francisco (2012)
16. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. CoRR arXiv:1405.3694 (2014)
17. Griffin, L., Butler, B., de Leastar E, Jennings, B., Botvich, D.: On the performance of access control policy evaluation. In: 2012 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY), pp. 25–32 (2012)
18. Hu, H., Ahn, G.J., Kulkarni, K.: Detecting and resolving firewall policy anomalies. IEEE Trans. Dependable Secur. Comput. **9**(3), 318–331 (2012)
19. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. IEEE Trans. Dependable Secur. Comput. **10**(6), 341–354 (2013)
20. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. Int. J. Softw. Tools Technol. Transf. **10**(6), 503–520 (2008)
21. Kolovski, V., Hendler, J., Parsia, B.: Analyzing web access control policies. In: Proceedings of the 16th International Conference on World Wide Web, WWW '07, pp. 677–686 (2007)
22. Lee, J., Wang, Y., Zhang, Y.: Automated reasoning about xacml 3.0 delegation using answer set programming. In: CEUR Workshop Proceedings, CEUR-WS, vol. 1433 (2015)
23. Lifschitz, V.: What is answer set programming? In: Proceedings of the 23rd National Conference on Artificial Intelligence, vol. 3, pp. 1594–1597 (2008)
24. Lin, D., Rao, P., Bertino, E., Li, N., Lobo, J.: EXAM: a comprehensive environment for the analysis of access control policies. Int. J. Inf. Secur. **9**(4), 253–273 (2010)
25. Liu, A.X., Chen, F., Hwang, J., Xie, T.: XEngine: a fast and scalable XACML policy evaluation engine. SIGMETRICS **'08**, 265–276 (2008)
26. Margheri, A., Masi, M., Pugliese, R., Tiezzi, F.: A rigorous framework for specification, analysis and enforcement of access control policies. IEEE Trans. Softw. Eng. **99**, 1–1 (2017)
27. Mejri, M., Yahyaoui, H.: Formal specification and integration of distributed security policies. Comput. Lang. Syst. Struct. **49**, 1–35 (2017)
28. Ramli, C.D.P.K.: Detecting incompleteness, conflicting and unreachability XACML policies using answer set programming. CoRR, arXiv:1503.02732 (2015)
29. Ramli, C.D.P.K., Nielson, H., Nielson, F.: XACML 3.0 in answer set programming. In: Logic-Based Program Synthesis and Transformation, Lecture Notes in Computer Science, vol. 7844, pp. 89–105 (2013)
30. Rezvani, M., Aryan, R.: Analyzing and resolving anomalies in firewall security policies based on propositional logic. In: IEEE 13th International Multi Topic Conference, INMIC (2009)

31. Rezvani, M., Ignjatovic, A., Pagnucco, M., Jha, S.: Anomaly-free policy composition in software-defined networks. In: IFIP Networking 2016 Conference (Networking 2016), Vienna, Austria (2016)
32. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies, SACMAT '06, pp. 160–169 (2006)
33. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Formal analysis of XACML policies using SMT. Comput. Secur. **66**(Supplement C), 185–203 (2017)
34. Yuan, L., Mai, J., Su, Z., Chen, H., Chuah, C.N., Mohapatra, P.: FIREMAN: a toolkit for firewall modeling and analysis. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 199–213 (2006)