Formal Analysis of XACML Policies using SMT

Fatih Turkmen[a,*], Jerry den Hartog[b], Silvio Ranise[c], Nicola Zannone[b]

[a] *University of Amsterdam, Amsterdam, Netherlands*
[b] *Eindhoven University of Technology, Eindhoven, Netherlands*
[c] *Fondazione Bruno Kessler, Trento, Italy*

Abstract

The eXtensible Access Control Markup Language (XACML) has attracted significant attention from both industry and academia, and has become the de facto standard for the specification of access control policies. However, its XML-based verbose syntax and rich set of constructs make the authoring of XACML policies difficult and error-prone. Several automated tools have been proposed to analyze XACML policies before their actual deployment. However, most of the existing tools either cannot efficiently reason about non-Boolean attributes, which often appear in XACML policies, or restrict the analysis to a small set of properties. This work presents a policy analysis framework for the verification of XACML policies based on SAT modulo theories (SMT). We show how XACML policies can be encoded into SMT formulas, along with a query language able to express a variety of well-known security properties, for policy analysis. By being able to reason over non-Boolean attributes, our SMT-based policy analysis framework allows a fine-grained policy analysis while relieving policy authors of the burden of defining an appropriate level of granularity of the analysis. An evaluation of the framework shows that it is computationally efficient and requires less memory compared to existing approaches.

[*] Corresponding author

Email addresses: `f.turkmen@uva.nl` (Fatih Turkmen), `j.d.hartog@tue.nll` (Jerry den Hartog), `ranise@fbk.eu` (Silvio Ranise), `n.zannone@tue.nl` (Nicola Zannone)

1. Introduction

Data and other digital resources have become a valuable asset for most organizations. Their protection is thus of utmost importance. Access control is a widely adopted technology for information security and, in particular, to ensure that sensitive information can only be accessed by authorized users.

In the last decades several access control models and languages have been proposed for the specification and enforcement of access control policies. Among these languages, the eXtensible Access Control Markup Language (XACML) [1] provides an expressive and extensible syntax in XML for the specification of attribute-based access control policies as well as means to combine policies possibly specified by independent authorities. XACML has been widely used in academia and adopted by many enterprises such as IBM [2], becoming the de facto standard for access control.

However, due to its rich set of constructs and XML-based verbose syntax, policy specification in XACML is known to be difficult and error-prone [3, 4]. For instance, when a policy is updated to address new requirements, it becomes difficult to determine whether the revised policy works as intended. Even small errors can lead to large data breaches. Ensuring the correctness of access control policies, especially in the error-prone setting of XACML policy specification, is thus a crucial task for protecting sensitive data.

To assist security administrators in the definition of their policies, several methods and tools have been developed for the verification of access control policies at design time using formal reasoning [3, 4, 5, 6, 7, 8]. These tools aim to verify whether an access control policy (or a set of policies) satisfies certain properties. A property can vary from checking the (types of) access requests that should be allowed (or denied) by a policy to the analysis of the relation between two policies such as being as permissive/restrictive as another policy (i.e., policy refinement [5]). However, many of the existing approaches can only analyze a restricted set of security properties due to limits of the expressiveness of the policy formalization used. Moreover, exiting policy analysis tools often do not naturally support reasoning over non-Boolean variables and functions, which often appear in XACML policies. As a consequence, they are not able to analyze access control policies at a fine level of granularity or the performance of the analysis deteriorates very quickly.

To address these issues, in a previous work [9], we have introduced a framework that employs SAT modulo theories (SMT) [10] as the underlying reasoning method for the formal analysis of XACML policies. SMT is a natural extension to propositional satisfiability (SAT) [11] in which SMT solvers employ tailored reasoners when solving non-Boolean predicates in the input formula. In particular, SMT enables the use of background theories, such as linear arithmetic and equality, to reason about the satisfiability of many-sorted first order formulas. In [9] we provided the intuition of how XACML policies can be encoded into SMT formulas and presented a powerful query language that allows the specification and analysis of a vast range of security properties that have been proposed in the literature. However, given the complex syntax of XACML, it is desirable to have an automated translation of XACML policies into SMT formulas while preserving the semantics of the original policy. Moreover, although SMT provides a powerful approach to problem verification including policy analysis, the problem of checking the satisfiability of arbitrary many-sorted first order logic formulas can be undecidable.

In this paper, we extend [9] by providing the following contributions:

- We provide a complete procedure for the automated translation of XACML policies into SMT formulas for policy analysis. Specifically, we present an encoding of XACML policies that flattens the hierarchical structure of a policy. To support the translation, we provide an encoding of XACML combining algorithms and a mapping between the most common XACML functions and the available SMT background theories.

- We provide a proof of the correctness of the proposed encoding, thus guaranteeing that the semantics of the original policy is preserved.
- We confirm the expressive power of our query language by encoding a new set of properties, namely separation of duty constraints.
- We study under which conditions SMT solvers are capable of tackling policy analysis problems. To the best of our knowledge, this is the first work that provides a detailed study of the complexity of policy analysis in SMT.
- We complement the study of the complexity with an evaluation of the framework through a more extensive set of experiments compared to [9]. In particular, we compare our SMT-based approach with SAT-based approaches using different SAT solvers, thus providing a more comprehensive comparison between the two approaches. Moreover, we evaluate our framework using additional realistic policies.

The paper is structured as follows. The next section provides background about XACML and SMT. Section 3 presents our encoding of XACML policies as SMT formulas. Section 4 introduces a query language for the specification of security properties and demonstrates this language by encoding a number of well-known security properties from the literature. Section 5 discusses the complexity of policy analysis in SMT. Section 6 presents an experimental evaluation of our framework. Finally, Section 7 discusses related work, and Section 8 concludes the paper providing directions for future work.

2. Preliminaries

This section introduces the basic notions underlying XACML and SMT.

2.1. XACML

XACML is an OASIS standard for the specification of access control policies. It provides an attribute-based language that allows the specification of composite policies. In this work, we focus on the core specification of XACML v3 [1] (without obligations) unless it is indicated otherwise.

XACML policies are expressed using three policy elements: *policysets*, *policies* and *rules*. A policyset consists of policysets and policies; policies in turn consist of rules. If a policy element p_1 is nested (i.e., textually contained) in a policy element p_2 we say that p_1 is a *child policy element* of p_2 or, equivalently, that p_2 is the *parent policy element* of p_1. Each policy element has a (possibly empty) *target* which defines (restricts) the applicability of the policy element, i.e. the set of access requests that the policy element applies to. In addition, rules specify an *effect* element that defines whether an access request should be allowed (*Permit*) or denied (*Deny*), and can be associated with a *condition* to further restrict their applicability. A condition is a predicate that must be satisfied for the rule to be applicable.

The evaluation of an access request against a policy element results in an access decision. XACML v3 uses two access decision sets: a four-valued decision set (i.e., Permit, Deny, Indeterminate and NotApplicable) and six-valued decision set in which the Indeterminate decision is extended to record the decision that would have been returned if an error in the policy evaluation did not occur (the so-called *Indeterminate extended set*). In particular, the Indeterminate extended set consists of decisions Indeterminate{P}, Indeterminate{D} and Indeterminate{PD} where Indeterminate{P} indicates an Indeterminate decision from a policy element which could have evaluated to Permit (but not Deny), Indeterminate{D} indicates an

Indeterminate decision from a policy element which could have evaluated to Deny (but not Permit) and Indeterminate$\{$PD$\}$ indicates an Indeterminate decision from a policy element which could have evaluated to Permit or Deny. These decision sets have different contexts of use. The six-valued decision set is used when a more fine-grained decision is desired. For instance, it is used for the evaluation of rules as atomic decision units or within certain combining algorithms (see below). The four-valued decision set is always used to express the final decision of an XACML policy.

To combine decisions obtained from the evaluation of different applicable policy elements, XACML provides a number of combining algorithms: permit-overrides (pov), deny-overrides (dov), deny-unless-permit (dup), permit-unless-deny (pud), first-applicable (fa) and only-one-applicable (ooa). These algorithms define procedures to evaluate composite policies based on the order of the policy elements and priorities between decisions. We refer the reader to [1] for their definition.

Here, it is worth mentioning that permit-overrides and deny-overrides are defined over the six-valued decision set, whereas the other combining algorithms are defined over the four-valued decision set. Thus, depending on the context of use and the combining algorithm specified in a policy element, decisions have to be mapped from one decision set to in the other [12]. Intuitively, decisions in the Indeterminate extended set are mapped to the Indeterminate decision when the four-valued decision set has to be used, and the Indeterminate decision is mapped to Indeterminate$\{$PD$\}$ when the six-valued decision set has to be used; the other decisions (i.e., Permit, Deny and NotApplicable) are mapped to the same decision.

Hereafter, we use a concise notation to represent XACML policies, which is defined as follows:

$$\begin{array}{rll}
PS = & (ca, \mathit{Target}, PL) & \text{// PolicySet} \\
P = & (ca, \mathit{Target}, RL) & \text{// Policy} \\
R = & (\mathit{Effect}, \mathit{Target}, \mathit{Condition}) & \text{// Rule} \\
PL = & [PS] \mid [P] \mid PL \circ [PS] \mid PL \circ [P] & \text{// Policy List} \\
RL = & [R] \mid RL \circ [R] & \text{// Rule List} \\
\mathit{Effect} = & \mathit{Permit} \mid \mathit{Deny} & \text{// Rule Effect}
\end{array}$$

where ca is used to denote a combining algorithm, $[e]$ the list containing just element e and operator $\circ$ the concatenation of lists. Next we present a sample XACML policy, which is used as a running example throughout the paper.

Example 1 *A user is allowed to create a "transaction" object only if her credit balance (*credit*) is higher than the value of the transaction itself (*value*) plus banking costs (*cost*). Transactions can only be created during working days (i.e., Monday, Tuesday, Wednesday, Thursday, Friday) within the time interval 08:00-18:00. One way to model this policy is to represent (the negation of) these constraints as* Deny *rules. In addition, we introduce a default policy with effect* Permit *and combine these rules using* deny-overrides *(*dov*). According to* dov*, the policy yields decision* Permit *only if none of the* Deny *rules is applicable. This policy is represented in our notation as follows:*

$p:\ (\mathsf{dov}, (\mathsf{resource\text{-}type} = \text{“}\mathit{transaction}\text{”} \wedge \mathsf{action\text{-}id} = \text{“}\mathit{create}\text{”}), [r_1, r_2, r_3])$

$r_1 : (\mathit{Deny}, (\mathsf{value} + \mathsf{cost} > \mathsf{credit}), \mathbf{true})$

$r_2 : (\mathit{Deny}, (\mathsf{current\text{-}day} \notin \{Mo, Tu, We, Th, Fr\} \vee$

$\qquad \mathsf{current\text{-}time} < 08{:}00 \vee \mathsf{current\text{-}time} > 18{:}00), \mathbf{true})$

$r_3 : (\mathit{Permit}, \mathbf{true}, \mathbf{true})$

where **true** *is used to indicate that the target of the policy element (or the condition of a rule) matches every access request. We assume that attributes* value*,* cost*,* credit*,* current-time *and* current-day *are further constrained with function* one-and-only *so that the evaluation of the target of the policy element returns an error (i.e., it evaluates to* Indeterminate*) if multiple values are provided for such attributes.*

2.2. Satisfiability Modulo Theories

SMT [10] is a generalization of SAT in which Boolean variables can be replaced by constraints from a variety of theories. To specify SMT formulas, we follow the SMT-LIB (v2) standard (`http://www.smtlib.org`) which is based on Many-Sorted First Order Logic (MFOL). Hereafter, we assume the usual syntactic (e.g., sort, constant, predicate and function symbols, terms, atoms, literals, Boolean connectives) and semantic (e.g., domain, structure, satisfaction, model, and validity) notions of MFOL; see [13] for formal definitions.

A theory $\mathcal{T}$ consists of a signature and a class of models. The signature fixes the vocabulary to build formulas, and the class of models gives the meaning of the symbols in the vocabulary. As an example, consider the theory of an enumerated data-type: the signature consists of a single sort symbol and n constants corresponding to the elements in the enumeration; the class of models contains all structures interpreting the sort symbol as a set of cardinality n. Another example is Linear Arithmetic over the Integers (LAI), in which the signature consists of numerals (corresponding to integers), binary addition, and the usual ordering relations; the class of models contains the standard model of the integers in which only linear constraints are considered, i.e. constraints can only contain variables multiplied by a constant. For yet another example, consider the theory of uninterpreted functions: the signature consists of a finite set of symbols and the equality sign; the class of models contains all those structures interpreting the equality sign as a congruence relation and the other symbols in the signature as arbitrary constants, functions, or relations.

A formula φ is $\mathcal{T}$*-satisfiable* (or *satisfiable modulo* $\mathcal{T}$) iff there exists a structure $\mathcal{M}$ in the class of models of $\mathcal{T}$ and a valuation ϕ (i.e., a mapping from the variables that are not in the scope of a quantifier in the formula to the elements in the domains of $\mathcal{M}$) satisfying φ (in symbols, $\mathcal{M}, \phi \models \varphi$). A formula φ is $\mathcal{T}$*-valid* (or *valid modulo* $\mathcal{T}$) iff for every structure $\mathcal{M}$ in the class of models of $\mathcal{T}$ and every valuation ϕ, we have that $\mathcal{M}, \phi \models \varphi$. Notice that a formula φ is $\mathcal{T}$-valid iff its negation (i.e., $\neg\varphi$) is $\mathcal{T}$-unsatisfiable. We discuss the (un)satisfiability of some formulas below.

Example 2 *Formula $x \neq c_1 \wedge x \neq c_2 \wedge c_1 \neq c_2$ (where $x \neq c_i$ and $c_1 \neq c_2$ abbreviate $\neg x = c_i$ (for $i = 1, 2$) and $\neg c_1 = c_2$ respectively, $\neg$ is negation, and $\wedge$ denotes conjunction) is unsatisfiable modulo the theory of an enumerated data-type containing only two elements since there does not exist a valuation satisfying it. Actually, this formula is only satisfiable in a domain with at least three elements as only in a such a domain x can be given a value distinct from both c_1 and c_2.*

Formula $x < y \wedge y < z \wedge x \geq z$ is unsatisfiable modulo LAI since the less-than relation $<$ is transitive and thus $x < z$ when $x < y$ and $y < z$.

Formula $x = y \wedge f(x) \neq f(y)$ is unsatisfiable modulo the theory of uninterpreted functions since $=$ is interpreted as a congruence relation and thus the values of f at x and y must be equal when the values of x and y are so.

Solving the satisfiability modulo theories of quantified formulas is only possible when these belong to some classes (see, e.g., [14]) as it is the case of the one considered in the example below.

Example 3 *Formula $\neg Q(c) \wedge P(f(c)) \wedge \forall x.P(f(x)) \rightarrow Q(x)$ is unsatisfiable modulo the theory of uninterpreted functions because of the following observations. The universally quantified variable x can be instantiated to a constant c, obtaining the quantifier-free formula $\neg Q(c) \wedge P(f(c)) \wedge P(f(c)) \rightarrow Q(c)$. This is* *obviously unsatisfiable since the second and third conjuncts imply $Q(c)$, which is in contradiction with the first conjunct $\neg Q(c)$.*

In general, the satisfiability modulo the theory of uninterpreted functions of arbitrary first-order formulas is equivalent to the one of arbitrary first-order formulas, which is well-known to be undecidable [13]. The satisfiability modulo the theory of uninterpreted function becomes decidable when considering only arbitrary Boolean combinations of atoms, i.e. first-order formulas not containing quantifiers [10]. This observation can be generalized to other theories because of the following observation. It is always possible to transform arbitrary Boolean combinations of atoms into disjunctions of constraints, i.e. conjunctions of atoms or their negations. Thus, the decidability of the satisfiability of an arbitrary Boolean combinations of atoms modulo a theory T reduces to the one of the satisfiability of constraints modulo T.

While decidability of satisfiability modulo theories transfers from constraints to arbitrary Boolean combinations of atoms, the same does not hold for complexity. It turns out that the complexity of solving the satisfiability modulo theories problem for arbitrary Boolean combination of atoms is NP-hard as it subsumes SAT solving. Therefore, even though the complexity of checking the satisfiability of constraints modulo a theory T is polynomial (this is the case, for instance, when T is the theory of uninterpreted functions [10]), this is not the case when considering arbitrary Boolean combinations (checking the satisfiability modulo the theory of uninterpreted functions of arbitrary Boolean combinations of atoms is NP-complete [10]).

Combining theories is another dimension with respect to which transferring complexity is not possible. In several verification problems (we will see that policy analysis is one of them), the theory T modulo which we need to check for satisfiability, it is the combination of two (or more) simpler theories T_1 and

T_2. This is because it is often the case that different kinds of information have to be taken into account; for instance, XACML policies may have conditions involving attributes whose values range over several data-types such as integers, enumerations, and so on. Being able to infer the decidability and the complexity of the combined theory T from its components T_1 and T_2 is indeed desirable. The widely used Nelson-Oppen combination method makes it possible to do this under reasonable assumptions. For decidability [15], it is sufficient that the component theories share only variables (but no functions or relations) and are stably infinite, i.e. it is always possible to find an infinite model for a satisfiable formula. For complexity [16], the situation is more complex and depends on the convexity (a generalization of the well-known geometric notion) of the component theories; for instance, the theory of uninterpreted functions is convex while LAI is not. The crux is that convex theories do not require case-splitting whereas non-convex ones do and this is more expensive from a computational point of view; for instance, the constraint $1 \leq x \wedge x \leq 100$ may require to consider 100 cases, one for each case $x = i$ with $i = 1, ..., 100$. When the satisfiability problem for constraints modulo component theories is solvable in polynomial-time, it is possible to show that the same problem modulo their combination is also solvable in polynomial-time if the component theories are convex. When they are non-convex, the satisfiability modulo the combined theories is no longer in the polynomial class even if the satisfiability problem of each component theory is polynomial-time solvable.

We conclude this brief introduction to SMT solving with an example illustrating the ideas underlying the Nelson-Oppen method.

Example 4 *Let us consider the constraint*

$$1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

and the problem of showing its un-satisfiability modulo the combination of LAI and the theory of uninterpreted functions. A modular way of reasoning is as follows. Introduce two additional variables w_1 and w_2 and rewrite the constraint above into the (equi-satisfiable) conjunction of

$$\begin{aligned}\varphi_1 &:= f(x) \neq f(w_1) \wedge f(x) \neq f(w_2)\\ \varphi_2 &:= 1 \leq x \wedge x \leq 2 \wedge w_1 = 1 \wedge w_2 = 2\end{aligned}$$

It is easy to see that φ_1 is satisfiable modulo the theory of uninterpreted functions and that φ_2 is satisfiable modulo LAI. However, to be entitled to conclude that $\varphi_1 \wedge \varphi_2$ is satisfiable modulo the combination of the two theories, we need to perform an additional check concerning the shared (i.e., occurring in both φ_1 and φ_2) variables x, w_1 and w_2, namely that there exist valuations ϕ_1 and ϕ_2 such that $\mathcal{M}_1, \phi_1 \models \varphi_1$, $\mathcal{M}_2, \phi_2 \models \varphi_2$, $\phi_1(x) = \phi_2(x)$, $\phi_1(w_1) = \phi_2(w_1)$, and $\phi_1(w_2) = \phi_2(w_2)$ where $\mathcal{M}_1$ is a model of the theory of uninterpreted function and $\mathcal{M}_2$ is a model of LAI. It is not difficult to see that such ϕ_1 and ϕ_2 do not exist by considering all possible equalities and disequalities over the variables x, w_1 and w_2. For the sake of space, we only consider two cases here: (a) assume $x = w_1 = w_2$: $\varphi_1 \wedge x = w_1$ is unsatisfiable modulo the theory of uninterpreted function symbols ($x = w_1$ is in contradiction with $f(x) \neq f(w_1)$) and (b) assume $w_1 = w_2 \wedge x \neq w_1 \wedge x \neq w_2$: $\varphi_1 \wedge x = w_1$ is unsatisfiable modulo LAI (from $w_1 = 1$, $w_2 = 2$, and $w_1 = w_2$, we derive $1 = 2$, a contradiction). The other three cases are similar.

We further elaborate on these and related issues in Section 5.

3. Encoding XACML policies in SMT

This section presents our encoding of XACML policies into SMT formulas. We first present a transformation of XACML policies that flattens the hierarchical structure of an XACML policy while preserving the semantics of the original policy. We then discuss the encoding of the transformed policies into SMT formulas.

3.1. Transformation of XACML Policies

XACML policies are defined within a schema $\langle Att, Dom \rangle$ where Att is a set of attributes and Dom represents the attribute domains Dom_a for all attributes $a \in Att$. A schema acts as a vocabulary from which policies can be constructed. We refer to product $\prod_{a \in Att} 2^{Dom_a}$ as the *policy space* and to its elements as *attribute assignments*. An attribute assignment denotes a policy expression in extensional form and maps each attribute to a (possibly empty) set of values in its domain. An *access request* $\langle a_1 = v_1, \ldots, a_k = v_k \rangle$ (with $a_i \in Att, v_i \in Dom_{a_i}$ for $i = 1, \ldots, k$) specifies an attribute assignment representing the context in which access is requested. In an access request, we omit the attributes for which a value has not been explicitly defined (i.e., it is assigned to the empty set), and multiple attribute/value pairs with the same attribute (note that $a_i = a_j$ with $i \neq j$ is allowed) indicate that multiple values are assigned to that attribute. Hereafter, $\mathcal{R}$ denotes the set of all possible access requests, i.e. the policy space.

To encode XACML policies into SMT formulas, we use the notions of *applicability space* and *decision space* introduced in [9]. However, compared to the work in [9], we extend these notions with an additional component to make the set of requests for which a policy element is not applicable explicit. This choice is to simplify the specification of policy analysis problems (see Section 4).

Every policy element in XACML (i.e., policyset, policy and rule) specifies a (possibly empty) set of constraints in their targets. Rules may additionally have conditions providing additional constraints. These constraints, hereafter referred to as *applicability constraints*, are used to determine to which access requests a policy element applies. In particular, given an access request, the applicability constraints of a policy element can be evaluated to either *Match* (i.e., the policy element is applicable), *NoMatch* (i.e., the policy element is not applicable) or *Indeterminate* (i.e., it is not possible to evaluate applicability constraints due to an error or missing information). Conditions similarly evaluate *True* (rule applies), *False* (rule does not apply) or *Indeterminate*.

Definition 1 (Applicability Space) *For applicability constraints in the form of a target (or condition), the induced applicability space AS is a triple $\langle AS_A, AS_{IN}, AS_{NA} \rangle$ where AS_A and AS_{NA} contains the access requests for which the target (condition) evaluates to Match (True) and NoMatch (False) respectively, and AS_{IN} contains the access requests for which evaluating the constraints produces an error.*

The applicability space induced by the target of a policy element is defined in [1, Section 7.7] and for the condition of a rule in [1, Section 7.9].

The evaluation of an access request against a policy element results in an access decision (see Section 2.1). The decision space of a policy element partitions the policy space into classes of access requests that are evaluated to the same decision.

Definition 2 (Decision Space) *The decision space DS of an XACML policy element is a tuple $\langle DS_P, DS_D, DS_{IN}, DS_{NA} \rangle$ such that each element of the tuple is the set of access requests that evaluate to* Permit, Deny, Indeterminate *and* NotApplicable, *respectively. The Indeterminate decision space DS_{IN} is a triple $(DS_{IN(P)}, DS_{IN(D)}, DS_{IN(PD)})$ representing decisions* Indeterminate{P}, Indeterminate{D} *and* Indeterminate{PD} *respectively.*

In the definition of decision space, we explicitly represent the Indeterminate decision space as a triple. Recall from Section 2.1 that XACML v3 uses two decision sets and decisions might have to be mapped from one set to the other. In particular, decision Indeterminate in the four-valued decision set is mapped to decision Indeterminate{PD} in the six-valued decision. We encode this mapping by using $DS_{IN(PD)}$ to represent the set of access requests that return decision Indeterminate when reasoning over the four-valued decision set. In other words, the Indeterminate space has the form $DS_{IN} = (\emptyset, \emptyset, DS_{IN(PD)})$ when a decision has to be expressed in the four-valued decision set. On the other hand, when a decision is mapped from the six-valued decision set to the four-valued decision set, the Indeterminate extended set is mapped to a single Indeterminate decision: given the Indeterminate space $(DS_{IN(P)}, DS_{IN(D)}, DS_{IN(PD)})$ in the six-valued decision set, this space is mapped into $(\emptyset, \emptyset, DS_{IN(P)} \cup DS_{IN(D)} \cup DS_{IN(PD)})$ in the four-valued decision set. It is easy to observe that this exactly captures the mapping described in Section 2.1. Hereafter, we use notation DS_{IN} or notation $(DS_{IN(P)}, DS_{IN(D)}, DS_{IN(PD)})$ depending on the context of use. In particular, we sometimes abuse the notation and write $DS_{IN} = X$ (instead of $DS_{IN} = (\emptyset, \emptyset, X)$) to indicate the set of access requests X that evaluate to Indeterminate within the four-valued decision set.

Example 5 provides an illustration of these spaces after we show how to capture the decision spaces in formulas that translate easily to SMT. First, however, we need to address the XACML combining algorithms.

We define function applyCA to take as arguments a combining algorithm and a list of decision spaces and returns the result of combining those spaces. Our policy analysis framework supports the analysis of policies expressed in both XACML v2 [17] and XACML v3 [1]. Fig. 1 presents the encoding of the combining algorithms as defined in XACML v3. For the sake of simplicity, they are presented as binary operators and can be extended to an arbitrary number of policy elements in a straightforward way. The encoding of the combining algorithms as defined in XACML v2

Deny-overrides

$$\begin{aligned}
DS^{\mathsf{dov}}_{D} &= DS^{p_1}_{D} \cup DS^{p_2}_{D}\\
DS^{\mathsf{dov}}_{IN(PD)} &= \{(DS^{p_1}_{IN(PD)} \cup DS^{p_2}_{IN(PD)}) \cup [DS^{p_1}_{IN(D)} \cap (DS^{p_2}_{IN(P)} \cup DS^{p_2}_{P})] \cup [DS^{p_2}_{IN(D)} \cap (DS^{p_1}_{IN(P)} \cup DS^{p_1}_{P})]\} \backslash DS^{\mathsf{dov}}_{D}\\
DS^{\mathsf{dov}}_{IN(D)} &= (DS^{p_1}_{IN(D)} \cup DS^{p_2}_{IN(D)}) \backslash (DS^{\mathsf{dov}}_{D} \cup DS^{\mathsf{dov}}_{IN(PD)})\\
DS^{\mathsf{dov}}_{P} &= (DS^{p_1}_{P} \cup DS^{p_2}_{P}) \backslash (DS^{\mathsf{dov}}_{D} \cup DS^{\mathsf{dov}}_{IN(PD)} \cup DS^{\mathsf{dov}}_{IN(D)})\\
DS^{\mathsf{dov}}_{IN(P)} &= (DS^{p_1}_{IN(P)} \cup DS^{p_2}_{IN(P)}) \backslash (DS^{\mathsf{dov}}_{D} \cup DS^{\mathsf{dov}}_{IN(PD)} \cup DS^{\mathsf{dov}}_{IN(D)} \cup DS^{\mathsf{dov}}_{P})\\
DS^{\mathsf{dov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Permit-overrides

$$\begin{aligned}
DS^{\mathsf{pov}}_{P} &= DS^{p_1}_{P} \cup DS^{p_2}_{P}\\
DS^{\mathsf{pov}}_{IN(PD)} &= \{(DS^{p_1}_{IN(PD)} \cup DS^{p_2}_{IN(PD)}) \cup [DS^{p_1}_{IN(P)} \cap (DS^{p_2}_{IN(D)} \cup DS^{p_2}_{D})] \cup [DS^{p_2}_{IN(P)} \cap (DS^{p_1}_{IN(D)} \cup DS^{p_1}_{D})]\} \backslash DS^{\mathsf{pov}}_{P}\\
DS^{\mathsf{pov}}_{IN(P)} &= (DS^{p_1}_{IN(P)} \cup DS^{p_2}_{IN(P)}) \backslash (DS^{\mathsf{pov}}_{P} \cup DS^{\mathsf{pov}}_{IN(PD)})\\
DS^{\mathsf{pov}}_{D} &= (DS^{p_1}_{D} \cup DS^{p_2}_{D}) \backslash (DS^{\mathsf{pov}}_{P} \cup DS^{\mathsf{pov}}_{IN(PD)} \cup DS^{\mathsf{pov}}_{IN(P)})\\
DS^{\mathsf{pov}}_{IN(D)} &= (DS^{p_1}_{IN(D)} \cup DS^{p_2}_{IN(D)}) \backslash (DS^{\mathsf{pov}}_{P} \cup DS^{\mathsf{pov}}_{IN(PD)} \cup DS^{\mathsf{pov}}_{IN(P)} \cup DS^{\mathsf{pov}}_{D})\\
DS^{\mathsf{pov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Only-one-applicable

$$\begin{aligned}
DS^{\mathsf{ooa}}_{D} &= (DS^{p_1}_{D} \cap DS^{p_2}_{NA}) \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{D})\\
DS^{\mathsf{ooa}}_{P} &= (DS^{p_1}_{P} \cap DS^{p_2}_{NA}) \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{P})\\
DS^{\mathsf{ooa}}_{IN} &= (DS^{p_1}_{P} \cap DS^{p_2}_{P}) \cup (DS^{p_1}_{P} \cap DS^{p_2}_{D}) \cup (DS^{p_1}_{D} \cap DS^{p_2}_{P}) \cup\\
&\quad (DS^{p_1}_{D} \cap DS^{p_2}_{D}) \cup DS^{p_1}_{IN} \cup DS^{p_2}_{IN}\\
DS^{\mathsf{ooa}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

First-applicable

$$\begin{aligned}
DS^{\mathsf{fa}}_{D} &= DS^{p_1}_{D} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{D})\\
DS^{\mathsf{fa}}_{P} &= DS^{p_1}_{P} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{P})\\
DS^{\mathsf{fa}}_{IN} &= DS^{p_1}_{IN} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{IN})\\
DS^{p}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Deny-unless-permit

$$\begin{aligned}
DS^{\mathsf{dup}}_{D} &= \mathcal{R} \backslash DS^{\mathsf{dup}}_{P}\\
DS^{\mathsf{dup}}_{P} &= DS^{p_1}_{P} \cup DS^{p_2}_{P}\\
DS^{\mathsf{dup}}_{IN} &= \emptyset\\
DS^{\mathsf{dup}}_{NA} &= \emptyset
\end{aligned}$$

Permit-unless-deny

$$\begin{aligned}
DS^{\mathsf{pud}}_{D} &= DS^{p_1}_{D} \cup DS^{p_2}_{D}\\
DS^{\mathsf{pud}}_{P} &= \mathcal{R} \backslash DS^{\mathsf{pud}}_{D}\\
DS^{\mathsf{pud}}_{IN} &= \emptyset\\
DS^{\mathsf{pud}}_{NA} &= \emptyset
\end{aligned}$$

Figure 1: Encoding of XACML v3 combining algorithms

is presented in Appendix A. As discussed in Section 2.1, combining algorithms are defined over either the four-valued decision set or the six-valued decision set. Thus, function applyCA applies the mappings described above if a conversion between decision spaces is required by the given combining algorithm. Moreover, function applyCA maps the obtained decision into the four-valued decision set if the policy element is the root element of the XACML policy.

Proposition 1 *Function* applyCA *(as in Figure 1) correctly combines the decision spaces, i.e. it returns the decision space for the 'Specified by the rule-combining algorithm' value (also called 'Combining Algorithm Value') of [1].*

The construction of the combining algorithms directly follows the XACML specification [1, Section C]. As such the proof is a laboursome but straightforward check of the different cases. A sketch of the proof is given in the Appendix. With applyCA encoding the combining algorithms' behavior in place, we can now capture the evaluation yielding the decision spaces of policy elements.

Theorem 1 (Policy Evaluation) *We can recursively compute the decision space of any policy element as follows: For a rule r with target T, condition C and effect $e = P$ for Permit or $e = D$ for Deny:*

$$\begin{aligned}
DS^r_x &= \begin{cases} AS^T_A \cap AS^C_A & \textit{if } e = x \\ \emptyset & \textit{otherwise} \end{cases} \\
DS^r_{IN(x)} &= \begin{cases} AS^T_{IN} \cup AS^C_{IN} & \textit{if } e = x \\ \emptyset & \textit{otherwise} \end{cases} \\
DS^r_{IN(PD)} &= \emptyset \\
DS^r_{NA} &= AS^T_{NA} \cup (AS^T_A \cap AS^C_{NA})
\end{aligned}$$

where $x \in \{P, D\}$ and AS^T and AS^C are the induced applicability spaces of T and C respectively.

For a policy(set) p with target T, combining algorithm ca and $DS^{q_1}, \ldots, DS^{q_n}$ being the decision spaces of its children:

$$\begin{aligned}
DS^p_x &= AS^T_A \cap DS^{ca}_x \\
DS^p_{IN(x)} &= \big((AS^T_A \cup AS^T_{IN}) \cap DS^{ca}_{IN(x)}\big) \cup (AS^T_{IN} \cap DS^{ca}_x) \\
DS^p_{IN(PD)} &= (AS^T_A \cup AS^T_{IN}) \cap DS^{ca}_{IN(PD)} \\
DS^p_{NA} &= AS^T_{NA} \cup DS^{ca}_{NA}
\end{aligned}$$

where $x \in \{P, D\}$, $DS^{ca} = \mathsf{applyCA}(ca, DS^{q_1}, \ldots, DS^{q_n})$ and AS^T is the induced applicability spaces of T.

The equations in Theorem 1 capture XACML policy evaluation in a form that easily translates into SMT formulas. These equations follow the evaluation requirements for different XACML policy elements to compute their decision spaces. If the policy element is a rule, the induced applicability spaces of its target and condition together determine whether the effect of the rule applies. If the policy element is a policy or a policyset, the spaces of its children are first determined, combined with the combining algorithm, resulting in space DS^{ca}, and finally the applicability space induced by the target of the policy element is taken into account according to the XACML specification [1, Section 7.12-7.14]. As the evaluation directly follows the specification the proof is straightforward. A sketch of the proof is given in Appendix B.

Example 5 *Consider the policy in Example 1. Below we represent the applicability constraints ac_i defined from the target of every policy element:*

ac_0 : "*transaction*" $\in$ resource-type

ac_1 : "*create*" $\in$ action-id

ac_2 : $\bigwedge_{d\in\{Mo,Tu,We,Th,Fr\}} d \notin$ current-day

ac_3 : $\forall v \in$ current-time $v > 18{:}00$

ac_4 : $\forall v \in$ current-time $v < 8{:}00$

ac_5 : $\forall v_1 \in$ credit, $v_2 \in$ cost, $v_3 \in$ value $(v_1 < v_2 + v_3)$

$ac_6, ..., ac_{10} : att = \emptyset \ \vee \ \exists v_1, v_2 \in att.(v_1 \neq v_2 \wedge v_1 \in att \wedge v_2 \in att)$

where the meta-variable att represents current-day, current-time, credit, cost *and* value *in ac_6, ac_7, ac_8, ac_9, and ac_{10}, respectively. Constraints $ac_6, ..., ac_{10}$ are used to express (the negation of) XACML function* one-and-only *by requiring att to be either empty or to contain at least two distinct elements (denoted by v_1 and v_2).*

In the specification of the decision spaces, we use notation $\overline{X}$ to denote the complement of a set X and we represent sets of access requests as the applicability constraints that render them. Thus, the permit space DS_P^{r1} of rule $r1$, for example,

	DS_P	DS_D	$DS_{I(P)}$	$DS_{I(D)}$	$DS_{I(PD)}$	DS_{NA}
r_1	$\emptyset$	$ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}$	$\emptyset$	$(ac_8 \cup ac_9 \cup ac_{10})$	$\emptyset$	$\overline{ac_5} \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}$
r_2	$\emptyset$	$(ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)}$	$\emptyset$	$(ac_6 \cup ac_7)$	$\emptyset$	$\overline{(ac_6 \cup ac_7)} \cap \overline{(ac_2 \cup ac_3 \cup ac_4)}$
r_3	$\mathcal{R}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
ca	$\overline{(ac_8 \cup ac_9 \cup ac_{10} \cup ac_6 \cup ac_7 \cup S)}$	$S = (ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}) \cup ((ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)})$	$\emptyset$	$\emptyset$	$(ac_8 \cup ac_9 \cup ac_{10} \cup ac_6 \cup ac_7) \cap \overline{S}$	$\emptyset$
p	$ac_0 \cap ac_1 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10} \cup ac_6 \cup ac_7 \cup S)}$	$ac_0 \cap ac_1 \cap S$	$\emptyset$	$\emptyset$	$ac_0 \cap ac_1 \cap (ac_8 \cup ac_9 \cup ac_{10} \cup ac_6 \cup ac_7) \cap \overline{S}$	$\overline{(ac_0 \cap ac_1)}$

Table 1: Decision spaces for r_1, r_2, r_3 and p

is equal to $ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}$ as the target of r_1 will be applicable if ac_5 holds, unless value, cost or credit is not assigned a single value (causing an error). The decision spaces of the three rules are shown in Table 1. The deny-override *algorithm of p combines these into space DS^{ca} also shown in the table. Finally, the target of p is then applied to obtain DS^p given as the last row of the table.*

3.2. Encoding into SMT Formulas

The applicability constraints used in the construction of decision spaces (through applicability spaces) are atomic expressions that determine which background theories are needed for analyzing a given XACML policy. They are defined upon XACML functions that can be modeled and interpreted in certain theories. Table 2 presents the association between classes of XACML functions and available theories. In the first and second columns, the class of functions and example instances from the XACML specification are provided. The third column presents the theory (or combination of theories) that can be used to model them, and the fourth column shows the complexity of checking the satisfiability of constraints modulo the respective theories or their combination. While a detailed discussion of the complexity of checking the satisfiability of arbitrary formulas deriving from policy analysis problem is in Section 5, the last column of Table 2 reports the complexity of reasoning in the theory without considering the Boolean structure (remember

XACML Function Type	**(Example) Functions**	**Theory**	**Complexity**
Logical	*or*, *and*, *not*		
	n-of	Cardinality Constraints on Sets	NP-complete
XPath-based	*xpath-node-count*	Equality with Uninterpreted Functions (EUF)	Polynomial
Regular-expression-based	*ipAddress-regexp-match*		
Arithmetic	*integer-add*, *integer-subtract*	Linear Arithmetic over the Integers (LAI)	NP-complete
Numeric comparison	*integer-equal*, *integer-greater-than*		
Date and time arithmetic	*dateTime-add-dayTimeDuration*		
Non-numeric comparison	*time-equal*, *time-in-range*		
	string-equal, *string-greater-than*	Theory of strings	NP-hard
String conversion	*string-normalize-space*		
Set	*type-intersection*, *type-union*	Theory of Arrays **and** Cardinality Constraints on Sets	NP-complete
Bag	*type-one-and-only*, *type-bag-size*		

Table 2: XACML Functions and Background Theories

that constraints are conjunctions of atoms or their negations). This is a hint of the difficulty of reasoning modulo background theories, as considering arbitrary Boolean combinations of atoms makes satisfiability checking NP-hard while adding quantifiers may make it undecidable (see Section 2.2).

The encoding of XACML functions in Table 2 is an adaptation of the rich literature about representing data-types in various programming languages or software systems as SMT theories [10]. Since this kind of encoding is well understood and to keep technicalities to a minimum, we do not discuss the details of the encoding for XACML functions and operators but we just sketch the main ideas. Most of the logical functions of XACML (i.e., *or*, *and*, *not*) do not require any specific background theory and are modeled by the corresponding Boolean connectives. Some applicability constraints involving equality predicates of attributes with finite domains can be modeled by the theory of enumerated data types in which attribute values are represented as constants within an appropriate signature. Constraints defined using arithmetic and numeric comparison functions require LAI. Constraints defined over strings, bags, or sets may require dedicated theories; e.g., constraints defined using string comparison and string conversion functions can be modeled with the theory of strings [18]. Constraints defined over bags and sets can be modeled with the theories of arrays [19] and cardinality constraints on sets [20]. For some constraints, such as those involving regular or XPath expressions, there is no available theory that reflects their semantics. In these cases, an option is to consider their functions and operators as symbols of the theory of uninterpreted functions. The idea (which is well-known in the SMT literature since the seminal paper [21]) is to abstract away the particular properties of the data structures that the functions and relations manipulate and consider only the properties that these share with any other functions and relations. This is an over-approximation: if we can prove the un-satisfiability of a formula φ modulo the theory of uninterpreted functions, then φ evaluates to false for all possible interpretations of the function and predicate symbols; thus also in those with the right interpretation. In case φ is satisfiable, we are left with the problem if φ evaluates to true in some model with the precise interpretation of the symbols. In our experiments (see Section 6), the approximation was always precise enough for detecting the unsatisfiability of the formula under consideration. The reason for this is that, in most cases, taking an access control decision only requires shallow reasoning about data structures. Intuitively, the properties required to perform this kind of reasoning are those of substituting equals for equals, i.e. those for checking satisfiability modulo the theory of uninterpreted functions. The following example discusses how some of the theories listed in Table 2 are used to encode the applicability constraints of the policy in Example 5.

Example 6 *Constraint ac_0 can be represented by* $\mathsf{resource\text{-}type}[\text{“}transaction\text{”}] = tt$ *in the theory of arrays, where* resource-type *is an array constant, the mixfix binary operator* $\_[\_]$ *is the operator for reading the value stored at a certain position (second argument) of an array (first argument) of the theory of arrays, and tt is a constant representing the Boolean value true. The intuition is that the set* resource-type *can be replaced with its characteristic function (abusing notation, we denote the* *function with the identifier of the set), which is such that if the element "$transaction$" belongs to it, the value returned by* resource-type *at "$transaction$" is true.*

As another example, ac_3 is encoded as $\forall v.(\textsf{current-time}[v] = tt \to v > n_{18:00})$ where the set current-time *is represented by its characteristic function, v is a variable of sort integer, and $n_{18:00}$ is an integer encoding the time value $18{:}00$.*

For constraint ac_0, we need to consider the theory of arrays as the mixfix binary operator $\_[\_]$ occurs in the constraint whereas for ac_3 we need to reason modulo the combination of LAI and the theory of arrays since it mentions the ordering relation $>$ of LAI besides the operator $\_[\_]$ of the theory of arrays. As already observed in Section 2.2, the capability of reasoning modulo combinations of theories is required by many verification problems, including those arising from policy analysis. As an other use of combination, consider constraint ac_5. In Example 1, we have assumed that the cost of a transaction is independent of its value. In some situations, this is an oversimplification that can be refined by using an uninterpreted function f that takes as input the value of the transaction and returns its cost. We use the uninterpreted function f precisely because we are not interested in specifying exactly how to compute the cost of the transition as these details are not important for taking an access control decision; it is enough to model the fact the cost of the transaction depends on its value. In this case, the constraint ac_5 can be written as follows: $\textsf{credit} < f(\textsf{value}) + \textsf{value}$*, that requires the capability of reasoning modulo the combination of LAI and the theory of uninterpreted functions as it contains f and the arithmetic symbols $+$ and $<$.*

At this point, it should be clear that the decision spaces of a policy can be straightforwardly translated to MFOL formulas over the attributes in Att and a theory $\mathcal{T}$ (possibly obtained as a combination of several simpler theories) specifying the algebraic structures of the values of Att in Dom. The justification for this is two-fold. First, as discussed above and shown in the example, the applicability constraints can be encoded as atomic expressions in MFOL with attributes from Att as variables and attribute domains from Dom as sorts. Since the theories to reason about these expressions are those listed in Table 2, the encoding of decision spaces in MFOL follows an immediate translation to respective MFOL constructs.

Definition 3 *Given an XACML policy p with decision space $\langle DS_P, DS_D, DS_{IN}, DS_{NA}\rangle$ and a background theory $\mathcal{T}$, the representation of p in SMT is a tuple $\langle \mathcal{F}_P, \mathcal{F}_D, \mathcal{F}_{IN}, \mathcal{F}_{NA}\rangle$ where $\mathcal{F}_P$, $\mathcal{F}_D$, $\mathcal{F}_{IN}$ and $\mathcal{F}_{NA}$ are many sorted first-order formulas encoding DS_P, DS_D, DS_{IN} and DS_{NA} respectively, and their terms interpreted in $\mathcal{T}$.*

For the analysis of XACML policies, we are interested in the final decision, which is expressed in the four-valued decision set. Thus, formula $\mathcal{F}_{IN}$ encodes the Indeterminate decision space within that decision set, i.e. $\mathcal{F}_{IN}$ encodes $DS_{IN(P)} \cup DS_{IN(D)} \cup DS_{IN(PD)}$. Hereafter, when talking about deciding satisfiability of a policy p in SMT, we refer to $\mathcal{T}$-satisfiability of the formulas $\mathcal{F}_P$, $\mathcal{F}_D$, $\mathcal{F}_{IN}$ and $\mathcal{F}_{NA}$.

4. XACML Policy Analysis

The previous section presented an encoding of XACML policies as SMT formulas. In this section we use this encoding to represent *policy analysis problems*. We first present a query language for the specification of policy properties; then we give example query formulas for various policy properties.

4.1. Query Language

A policy analysis problem aims to verify a policy (or a set of policies) against a given property. Policy properties are expressed in so called *queries*.

Definition 4 *Let $\langle Att, Dom\rangle$ be an access control scheme, $a_1, ..., a_n$ attributes in Att and $\mathcal{T}$ a background theory with signature Σ. A* policy analysis problem *is a tuple $\langle Q, (p_1, \ldots, p_n)\rangle$ where $p_1, \ldots, p_n$ are policies expressed in SMT with respect to $\mathcal{T}$ and Q is a* (policy) query. *A query Q is an expression of the form*

$$Q = P_i \mid D_i \mid IN_i \mid NA_i \mid g(t_1, \ldots, t_k) \mid \neg Q \mid Q_1 \vee Q_2 \mid \ldots$$
$$\mid (\forall x : \sigma\ Q) \mid (\exists x : \sigma\ Q) \mid \nu x.Q \mid Q\langle a_1 = v_1, \ldots, a_k = v_k\rangle$$

where P_i, D_i, IN_i and NA_i (for $i = 1, \ldots, n$) are new symbols representing the Permit, Deny, Indeterminate *and* NotApplicable *spaces of policy p_i (they thus represent $\mathcal{F}_P^{p_i}$, $\mathcal{F}_D^{p_i}$, $\mathcal{F}_{IN}^{p_i}$ and $\mathcal{F}_{NA}^{p_i}$ respectively, see also query semantics in Def. 5), g is a Σ-atom over terms $t_1, \ldots, t_k$ such that each term t is either a variable denoting attributes from Att or built using function symbols in Σ, and logical operators are defined as usual where Q_1 and Q_2 are queries. In quantified formulas, i.e. $(\forall x : \sigma\ Q)$ and $(\exists x : \sigma\ Q)$, σ ranges over sort symbols in the theory $\mathcal{T}$. $\nu x.Q$ represents the restriction of variable x in Q, $Q\langle a_1 = v_1, \ldots, a_k = v_k\rangle$ (with $v_1 \in Dom_{a_1}, \ldots, v_k \in Dom_{a_k}$) represents the instantiation of a policy with a request.*

The basic query P_i encodes (inclusion in) the Permit space of policy p_i; accordingly, P_i is satisfiable by any request permitted by p_i. (If only a single policy is used we omit the index, writing P rather than P_i.) Similarly, D_i, IN_i and NA_i represent the Deny, Indeterminate and NotApplicable spaces of p_i respectively. Note that the query language includes a single symbol for Indeterminate. This is because we are interested in the final decision of an XACML policy, which is expressed in the four-valued decision set. Construct $\nu x.Q$ is used to restrict the scope of the substitution of a variable x to a subformula Q of the query (i.e., $\nu x.Q \equiv Q[x/y]$ with y a fresh variable). This construct allows us to encode properties comparing a number of policies, in which some policies are instantiated with a request while other policies are instantiated with a different request. Construct $Q\langle a_1 = v_{1_j}, \ldots, a_k = v_k\rangle$ is logically equivalent to $Q \wedge v_1 \in a_1 \wedge \ldots \wedge v_k \in a_k$.

Example 7 *Let p_1, p_2 be two policies, and P_1, D_1, IN_1, NA_1 and P_2, D_2, IN_2, NA_2 their query symbols respectively. Below we present some example queries:*

- $(P_1 \wedge P_2)$*: some request is allowed by both p_1 and p_2.*
- $P_1\langle \text{subject-id} = \mathbf{Alice}, \text{resource-type} = \mathbf{transaction}, \text{action-id} = \mathbf{create}\rangle$*: policy p_1 allows Alice to create a transaction.*
- $(P_1 \wedge D_2)\langle \text{subject-id} = \mathbf{Alice}\rangle$*: some request of Alice is permitted by p_1 but denied by p_2.*

It is easy to see that simple logical manipulations allow one to derive an SMT formula from a query Q; for instance, the elimination of the operator ν requires performing standard variable substitution; similarly for the elimination of the $\langle \cdots \rangle$ operator. Because it is always possible and easy to derive an SMT formula from a query, by abuse of notation, we freely mix queries with SMT formulas.

The semantics of queries follows the classical MFOL semantics. Below we define the satisfiability of a policy analysis problem.

Definition 5 *Let $\langle Q, (p_1, \ldots, p_n)\rangle$ be a policy analysis problem and $\mathcal{T}$ a background theory with signature Σ. Let $\langle \mathcal{F}_P^{p_i}, \mathcal{F}_D^{p_i}, \mathcal{F}_{IN}^{p_i}, \mathcal{F}_{NA}^{p_i}\rangle$ be the encoding of policy p_i (with $i = 1, \ldots, n$) in SMT with terms interpreted in $\mathcal{T}$. We say that $\langle Q, (p_1, \ldots, p_n)\rangle$ is satisfiable with respect to $\mathcal{T}$ if formula*

$$Q \wedge \bigwedge_{i=1}^{n} (P_i \leftrightarrow \mathcal{F}_P^{p_i}) \wedge (D_i \leftrightarrow \mathcal{F}_D^{p_i}) \wedge (IN_i \leftrightarrow \mathcal{F}_{IN}^{p_i}) \wedge (NA_i \leftrightarrow \mathcal{F}_{NA}^{p_i})$$

is $\mathcal{T}$-satisfiable. Otherwise, we say that it is $\mathcal{T}$-unsatisfiable.

The precision of the analysis depends on the background theory $\mathcal{T}$. When it is obtained by a combination of the theories listed in Table 2 without resorting to the use of the theory of uninterpreted functions for abstracting away the semantics of some symbols (as discussed in Section 3.2), the result of the analysis is correct as no over-approximation has been performed. Instead, if one of the component theories of $\mathcal{T}$ is that of uninterpreted function symbols because there is no available theory interpreting the operators on a particular data structure as it is the case of XPath expressions, then the results of the analysis are precise only when the formula derived associated to the analysis problem (according to Definition 5) is found to be un-satisfiable modulo $\mathcal{T}$. Otherwise, we are left with problem of checking that among the interpretations satisfying the formula, there is at least one interpreting all the symbols with the exact semantics. This is indeed a difficult and time consuming problem which requires human intervention. However, as already observed in Section 3.2, in our experiments, it was never necessary to manually eliminate such spurious models. The reason is that reasoning about authorization conditions involves only shallow reasoning (namely, substituting equals for equals) about data structures.

4.2. Security Properties and Their Encoding

In our previous work [9], we have demonstrated the expressiveness of the query language by showing how various types of policy properties from the literature, namely policy refinement [5] and subsumption [4], change impact [3], attribute hiding [6] and scenario finding [22, 23], can be encoded in our framework. In this section, we confirm the expressiveness power of the query language and, in particular, we show how the query language can be used to model and verify a new class of properties, namely separation of duty constraints. We also present a revised version of attribute hiding.

Attribute Hiding. Attribute hiding refers to the situation in which a user obtains a more favorable authorization decision by hiding some of her attributes [6]. Intuitively, this type of attack exploits the non-monotonicity of access control systems such as XACML. We verify whether a policy is vulnerable to attribute hiding by checking if a request that is previously denied is permitted by hiding some attributes or attribute-value pairs. Here, we consider two types of attribute hiding attacks: *partial attribute hiding* and *general attribute hiding*.

Partial attribute hiding refers to the case where a user hides a single attribute-value pair. A policy is vulnerable to partial attribute hiding with respect to attribute-value pair $a = v$ (with $a \in Att, v \in Dom_a$) if the following formula is $\mathcal{T}$-satisfiable:

$$D \wedge a = B \wedge \nu a.(P \wedge a = B \setminus \{v\}) \tag{1}$$

Intuitively, a policy p is vulnerable to attribute hiding with respect to a certain attribute-value pair if a request including such an attribute-value pair is denied by p (i.e., a solution of D) and the request in which the attribute-value pair is suppressed is permitted by p (i.e., a solution of P). Note that we consider the values of a in the request and store these in set B so they are available inside the restriction where the values of a is B with v removed.

General attribute hiding extends partial attribute hiding in that a user completely suppresses information about one attribute. A policy is vulnerable to general attribute hiding with respect to attribute a if the following formula is $\mathcal{T}$-satisfiable:

$$D \wedge \nu a.(P \wedge a = \emptyset) \tag{2}$$

We use an example policy from [6] to discuss the analysis of attribute hiding.

Example 8 *Consider two competing companies, A and B. To protect confidential* *information from competitors, company A defines the following policy:*

$$
\begin{aligned}
p:&\quad (\mathsf{dov}, \mathbf{true}, [r_1, r_2])\\
r_1:&\quad (\mathit{Deny}, (\mathsf{confidential} = \mathit{true} \wedge \mathsf{employer} = B), \mathbf{true})\\
r_2:&\quad (\mathit{Permit}, \mathbf{true}, \mathbf{true})
\end{aligned}
$$

*The first rule (r_1) denies employees of company B to access confidential information while the second rule (r_2) grants access to every requests. The two rules are combined using deny-overrides (*dov*). Now consider the following access requests:*

$$
\begin{aligned}
req_1 =&\langle \mathsf{employer} = A, \mathsf{employer} = B, \mathsf{confidential} = \mathit{true}\rangle\\
req_2 =&\langle \mathsf{employer} = A, \mathsf{confidential} = \mathit{true}\rangle\\
req_3 =&\langle \mathsf{confidential} = \mathit{true}\rangle
\end{aligned}
$$

Rule r_1 is only applicable to request req_1 and thus the request is denied. Rule r_2 is applicable to the remaining requests and thus access is granted for requests req_2 and req_3. Therefore, we can observe that, if the subject can hide some information from the request, then she would be allowed to access confidential information leading to a violation of the conflict of interest requirement. For instance, one can exploit partial attribute hiding and reduce req_1 to req_2 by suppressing element employer $= B$ *from the request. As an alternative, it is possible to reduce req_1 to req_3 by suppressing attribute* employer *from the request (general attribute hiding).*

Separation of Duty. Separation of duties (SoD) constraints have been proposed to prevent IT fraud by imposing restrictions on the users involved in the executions of sensitive tasks [24]. Typically, SoD constraints are specified separately from the access control policies, making their specification easier. XACML, however, does not support the specification of external constraints to restrict the permission that users can have [25]. Therefore, when specifying XACML policies, a policy author is forced to specify policies that satisfy SoD constraints by design. We now show how SoD constraints can be specified in our query language to check whether a XACML policy encodes them correctly.

Here, we consider static SoD constraints that restrict the permissions that a user can have (where a permission is an object-action pair).[1] Given a set of permissions $\{p_1, \ldots, p_n\}$, a SoD constraint is of the form;

$$(\{p_1, \ldots, p_n\}, t)$$

which reads as "each user has less than t permissions from the set $\{p_1, \ldots, p_n\}$ with $t \leq n$". This type of constraint can be encoded in our query language as follows:

$$\bigwedge_{1 \leq i_1 < i_2 \ldots < i_t \leq n} \neg \nu \bar{a}.P\langle p_{i_1} \rangle \vee \ldots \vee \neg \nu \bar{a}.P\langle p_{i_t} \rangle$$

In the encoding above, permissions are attribute assignments $\langle a_1 = v_{1_i}, \ldots, a_m = v_{m_k} \rangle$ such that $a_1, \ldots, a_m$ are attributes characterizing an action and an object. This notation (and its semantics) is the same of the one for access requests in Definition 4. Additionally, $\nu \bar{a}$ is used as a shorthand for $\nu a_1 . \nu a_2 \ldots \nu a_m$.

Example 9 *Assume that the policy in Section 2.1 is extended with three additional permissions to manage transactions, namely "approve", "execute" and "audit". The bank wants to prevent that a single user carries out the management of a*

[1]We restrict our focus to static SoD constraints as our framework aims at policy verification at design time. The evaluation of dynamic SoD constraints can only be performed at execution time.

transaction alone. This requirement corresponds to the following SoD constraint:

$$(\{\overbrace{create}^{p_1}, \overbrace{approve}^{p_2}, \overbrace{execute}^{p_3}, \overbrace{audit}^{p_4}\}, 3)$$

The constraint allows users to have only two permissions out of the four in the given set. Checking whether a policy satisfies this constraint can be performed using the following query:

$$(\neg\nu\bar{a}.P\langle p_1\rangle\vee\neg\nu\bar{a}.P\langle p_2\rangle\vee\neg\nu\bar{a}.P\langle p_3\rangle)\wedge(\neg\nu\bar{a}.P\langle p_1\rangle\vee\neg\nu\bar{a}.P\langle p_2\rangle\vee\neg\nu\bar{a}.P\langle p_4\rangle)\wedge$$
$$(\neg\nu\bar{a}.P\langle p_1\rangle\vee\neg\nu\bar{a}.P\langle p_3\rangle\vee\neg\nu\bar{a}.P\langle p_4\rangle)\wedge(\neg\nu\bar{a}.P\langle p_2\rangle\vee\neg\nu\bar{a}.P\langle p_3\rangle\vee\neg\nu\bar{a}.P\langle p_4\rangle)$$

where $\nu\bar{a}.$ *stands for* ν resource-type. ν action-id. *and* $\langle p_i\rangle$ *for* $\langle$resource-type= $\mathbf{transaction}$,action-id=$\mathbf{create}\rangle$, $\langle p_2\rangle$ *for* $\langle$resource-type=$\mathbf{transaction}$,action-id= $\mathbf{approve}\rangle$*, etc.*

5. Complexity of Policy Analysis with SMT

The previous two sections have described our method to reduce a policy analysis problem into an SMT problem. Our experience shows that the formulas and the background theories in the resulting SMT problems share three common features: (*F1*) the encoding of the XACML combining algorithms discussed in Section 3.1 generates formulas with a complex Boolean structure; solving such problems is already NP-hard as it subsumes SAT solving; (*F2*) the required background theories are those shown in Table 2; for all but one of them, checking the satisfiability of constraints (i.e., disregarding the Boolean structure of formulas) is either NP-hard or NP-complete; and (*F3*) the quantifiers occurring in the formulas of the generated SMT problems have limited scopes, i.e. the formulas are built by Boolean combinations of atoms and “small” quantified sub-formulas; the presence of quantifiers may give rise to the undecidability of the SMT problems. Below, we do not give a theoretical characterization of the complexity of solving policy analysis problems in terms of the generated SMT problems but rather discuss how state-of-the-art SMT solvers successfully tackle them by exploiting the three features identified above. Besides shedding some lights on the internal workings of SMT solvers, the discussion provides insights into the experimental results of Section 6.

*5.1. Exploiting feature (*F1*)*

Since the formulas in the SMT problem obtained by the reduction described above have a complex Boolean structure, it is of paramount importance to efficiently reason about the Boolean operators. We begin by presenting a naïve algorithm for checking the satisfiability of arbitrary Boolean combination, i.e. formulas not containing quantifiers. The main idea is to combine a SAT solver with a *theory solver* for checking the satisfiability of constraints: the SAT solver handles the propositional structure of the formula being checked for satisfiability whereas the theory solver takes care of the reasoning modulo the background theory. From the SAT solver viewpoint, the atoms of the formula being checked for satisfiability are considered as Boolean variables so that it can enumerate the satisfying assignments (if any) by mapping each Boolean variable to either true or false. If no Boolean assignment is found, then the input formula is said to be unsatisfiable. If a Boolean assignment a is found, a constraint c is derived by taking the conjunction of any predicate mapped to true or the negation of any atom mapped to false in a. Then, the theory solver is invoked on c. If the solver finds c to be satisfiable modulo the background theory, then the input formula is reported to be satisfiable. Otherwise (i.e., the unsatisfiability of c is detected by the solver), the next Boolean assignment (if any) is considered. When no more Boolean satisfying assignments are available, the formula is said to be unsatisfiable. The algorithm must terminate since there are at most 2^n Boolean assignments to be enumerated for n the number of predicates occurring in the input formula.

To improve performances, it would be beneficial to reduce the number of invocations to the theory solver by extending it with the capability of deriving a constraint that can be used to prevent the Boolean solver to enumerate some of the 2^n possible Boolean assignments. To do this, the basic idea can be illustrated by considering the following constraint c in the theory EUF: $x = y \wedge y = z \wedge x \neq z \wedge c'$ where c' contains k predicates. One can observe that c is unsatisfiable because of the first three predicates, regardless of the remaining k predicates in c'. Unfortunately, the naïve algorithm above will consider 2^k Boolean assignments containing the first three predicates in c to rediscover unsatisfiability each time. If the theory solver would be able to detect that c is unsatisfiable because of the first three predicates (called conflict set), then the negation of their conjunction can be put in conjunction to the input formula so that the SAT solver would be prevented from enumerating the 2^k Boolean assignments sharing the same first three predicates. Indeed, when k is large (as it is usually the case), the speed-up can be quite substantial. Indeed, the smaller the conflict set, the more substantial the pruning effect on the number of Boolean assignments to be eliminated. Several heuristics have been designed to compute small conflict sets in an efficient way, depending on the background theory (even when it a combination of simpler theories); the interested reader is pointed to [10] for a thorough discussion.

*5.2. Exploiting feature (*F2*)*

Since checking the satisfiability of constraints (i.e. disregarding the Boolean structure of formulas) modulo the background theories used in the SMT problems generated by the reduction above (cf. Table 2) are almost all NP-hard or NP-complete, we explain why SMT solvers perform well in practice on these constraints despite the high theoretical complexity. In many of the formulas generated by our reduction, it is possible to identify certain classes of predicates for which the constraint satisfiability problem has lower complexity than the case for arbitrary predicates. For instance, LAI is decidable and NP-complete but in many of the formulas derived from XACML policy analysis problems, LAI predicates are of the form: $\pm x \pm y \leq c$ with c a constant. Such a class of predicates is called the Unit-Two-Variable-Per-Inequality (UTVPI) fragment for which constraint satisfiability is polynomial [10]. Modern SMT solvers are capable of recognizing if the predicates in the constraints are UTVPI and use a polynomial algorithm to check their satisfiability. For the other theories in Table 2 that are NP-complete, several heuristics have been designed to increase the performances of theory solvers when predicates are of certain forms (see, e.g., [26] for more on this point for the theory of arrays). Fortunately, in many cases, the formulas in the SMT problems generated by our reduction fall in these particular classes so that SMT solvers can efficiently reason modulo these theories.

Theory reasoning is further complicated by the fact that a combination of theories is often needed to solve policy analysis problems for XACML policies. For instance, in the simple policy of Example 2, we have equality functions, numerical comparisons, and operators over sets: the corresponding theories need to be considered in combination in order to check the satisfiability of the various constraints generated by a policy analysis problem. Under suitable assumptions on the component theories, it is possible to build theory solvers capable of checking the satisfiability of constraints modulo the combination of theories by modularly re-using the theory solvers of the component theories [10]. For example, solvers for any combination of the theories in Table 2 can be modularly built by using a standard combination method [15] that has been implemented in many (if not all) available SMT solvers.

The main difficulty in the modular combination of theory solvers is the exchange of (disjunctions of) equalities between variables. To understand why this is needed, recall Example 4: a constraint solver for the theory of uninterpreted function symbols returns satisfiable on $\varphi_1 = f(x) \neq f(w_1) \wedge f(x) \neq f(w_2)$ and one for LAI returns satisfiable on $\varphi_2 = 1 \leq x \wedge x \leq 2 \wedge w_1 = 1 \wedge w_2 = 2$, despite the fact that their conjunction is unsatisfiable modulo the combination of the two theories. The crux is that the theory solver for LAI should be able – besides checking for satisfiability – to derive that φ_2 implies the disjunction $x = w_1 \vee x = w_2$. Considering the conjunction of φ_1 with each disjunct allows the solver for the theory of uninterpreted functions to detect unsatisfiability. This shows that sharing the information necessary to satisfiability checking in combination of theories may be computationally expensive according to the fact that equalities or their disjunctions need to be derived. This depends on a property of the component theories, called convexity, which is a generalization of the well-known geometrical notion. For instance, LAI is non-convex (in the example, it is necessary to derive a disjunction of equalities) while the theory of uninterpreted functions is convex (i.e., it is sufficient to derive implied equalities). Indeed, combining a non-convex theory requires to case-split on the implied disjunction of equalities and this gives rise to higher complexity of the combined theory solver. Approaches to improve, in practice, case-splitting when combining non-convex theories have been proposed in the literature by exploiting an available SAT solver [27].

*5.3. Exploiting feature (*F3*)*

The difficulty of checking the satisfiability of formulas derived from XACML policy analysis problems is substantially increased by the presence of quantifiers. The decidability of quantifier-free formulas does not extend to quantified formulas: checking the satisfiability of quantified formulas modulo the theory of EUF is undecidable [10]. Despite this and other negative results, several efforts have been put forward in identifying classes of quantified formulas whose satisfiability is decidable by integrating instantiation or quantifier-elimination procedures in SMT solvers; see [10] for pointers to relevant works. Fortunately, the formulas derived from policy analysis problems contain quantifiers with limited scopes, i.e. the quantified sub-formulas are small; for instance, Example 5 shows constraints in XACML policies (e.g., ac_3) that contain (universal) quantifiers with quite limited scope. This allows quantifier-instantiation procedures (such as the one described in [14]) to be highly successful (see again [14] for a detailed discussion of how the scope of quantifiers influences the success of the instantiation procedure).

5.4. SAT vs. SMT

One may wonder if SMT tools are really needed to solve the logical problems resulting from XACML policy analysis problems. In fact, for many of the theories mentioned in Table 2, reductions to the Boolean satisfiability problem are available so that state-of-the-art SAT solvers would be sufficient. Differentiating between SMT solvers and SAT solvers invoked after reductions does not seem meaningful from a conceptual point of view. In fact, Kroening and Strichman [28] propose a framework in which the two approaches are instances (called eager and lazy, respectively) of the same reasoning strategy that consists in combining SAT solving with a reduction based on theory solving. A lazy strategy leads to interleaving SAT and theory solving along the lines discussed above. An eager strategy first applies theory reasoning in order to derive a Boolean formula which is equisatisfiable to the input quantifier-free formula and then invokes the SAT solver. It is also observed that eager and lazy strategies are on the same continuum and available implementations of state-of-the-art SMT solvers can freely mix the various approaches. However, to the best of our knowledge, almost all available tools implement the lazy strategy since this has emerged as the most efficient of the two. Our experiments, discussed in Section 6.2, suggest that this is also the case for formulas generated from XACML policy analysis problems.

6. Evaluation

To validate our policy analysis framework, we have developed a prototype and evaluated its performance through two sets of experiments along the line of the experiments conducted in [9]. In the first set of experiment, we compared our approach and SAT-based techniques by analyzing the same policies at varying levels of granularity. In the second set, we analyzed realistic policies using SMT solving. The experiments were performed on a 64-bit machine with 16GB RAM and 3.40GHz quad-core CPU running Ubuntu.

6.1. Prototype Implementation

Our prototype, named X2S[2] [29], has been implemented in Java and allows the analysis of both XACML v2 and v3 policies containing a wide range of XACML functions. X2S employs Z3 [30], an open source SMT solver that allows efficient reasoning over a wide range of background theories, as the underlying reasoner. As required by certain types of queries such as *change impact*, X2S is capable of enumerating models that represent a solution to the policy analysis problem at hand. It does this by incrementally adding the negation of the obtained model as a new constraint to the original formula. The enumeration of the models can be performed until a given threshold is reached or the formula becomes unsatisfiable. It is worth noting that the analysis of some policies/properties may result in an infinite number of models. Several models, however, may be "insignificant" in terms of the attribute assignments they represent. Our tool avoids the generation of these models

[2]The tool is available at `http://security1.win.tue.nl/X2S`.

by employing a simple mechanism when necessary. More specifically, for some types of expressions for which this problem is more prominent (e.g., arithmetic expressions), our tool fixes the assignment of (arithmetic) variables to the first values found. As an example, consider the arithmetic expression $att_1 < att_2$. If the first solution returned by the solver assigns 4 and 5 to attributes att_1 and att_2 respectively, then our tool fixes these assignments by adding new (conjunctive) constraints $att_1 = 4$ and $att_2 = 5$ to the original formula.

6.2. Experiments 1: SAT vs. SMT

The first set of experiments aims to compare our SMT-based approach with SAT-based approaches for policy analysis. In particular, we assess the impact of direct reasoning over non-Boolean attributes and, thus, the granularity of analysis offered by both approaches in terms of memory usage and computation time. The granularity of the analysis denotes at which level of details the analysis is supported by policy analysis tools and is measured with respect to the size of attribute domains.

For the experiments, we considered policy analysis problems based on policy refinement (PR), policy subsumption (PS), change-impact (CI) analysis and scenario finding (SF), which have been presented in [9], as well as on partial (P-AH) and general (G-AH) Attribute Hiding and separation of duty (SoD) as presented in Section 4. To study the effect of the granularity of analysis on the performance, we considered three classes of domains for non-Boolean attributes: a 'small' (S) domain containing 10 values, a 'medium' (M) domain containing 100 values and a 'large' (L) domain containing 500 values.

For a fair comparison, we used the three best performing solvers, namely cominiSatPS, cryptominisat and glucose-syrup, from the 2015 SAT Competition.[3]

[3]http://baldur.iti.kit.edu/sat-race-2015/

		#Vars				**Memory(MB)**									
		SAT			SMT	cominisatps			cryptominisat			glucose-syrup			SMT
P	**Q**	S	M	L		S	M	L	S	M	L	S	M	L	
PR	$\neg\mathcal{F}$	231	591	2191	12	~0	30.4	358.6	~0	61.4	820.2	~0	32.6	360.9	1.9
PS	$\neg\mathcal{F}$	549	909	2509	12	~0	60.8	1628.3	31.7	156.8	1717.5	~0	65.8	1629.8	1.9
CI	$\mathcal{F}$	1049	1409	3009	12	~0	87.6	2571.7	32.7	245.7	2450.2	~0	89.7	2574.4	1.9
P-AH	$\neg\mathcal{F}$	15	15	15	3	~0	19	19	~0	38.1	38.1	~0	~0	~0	1.9
G-AH	$\neg\mathcal{F}$	15	15	15	19	~0	19	38.1	~0	38.1	~0	~0	~0	1.9	~0
SF	$\mathcal{F}$	151	511	1718	12	~0	25.4	247.8	~0	46.3	575	~0	27.6	250.5	1.9
SoD	$\mathcal{F}$	1203	1563	3163	12	~0	191.4	4558.8	~0	382.6	4599.1	~0	182.2	4560.7	2.1

Table 3: SAT vs. SMT: number of variables and memory usage

This is because different solvers may be optimized for certain types of problems and these solvers represent the state-of-the-art. To enable policy analysis using those SAT solvers, the test policies and properties were encoded in SAT according to the eager strategy discussed in Section 5.4 using the API provided by Sugar [31].

The results of the experiments are provided in Table 3 and Table 4. Table 3 presents the number of variable declarations used in the encoding and memory allocation. Table 4 shows the computation time required by the SAT solvers and by our prototype to solve the policy analysis problems. In the tables, the first column specifies the property (**P**) analyzed. The second column (**Q**) gives the class of formula used: finding a counter-example ($\neg\mathcal{F}$) or a satisfying assignment ($\mathcal{F}$).

In terms of variables, SAT encoding contains a large number of Boolean variables (e.g., orders of thousands) mostly due to the mapping of non-Boolean domains to Boolean variables. On the other hand, the encoding of the same policy analysis problems in SMT requires a maximum of 19 many-sorted first order variables. Consequently, the SMT solver requires much less memory than the SAT solvers for the encoding and analysis of the example policies. As expected, the memory required by the SAT solvers grows with the domain size. For instance, the amount of memory was around ~200MBs in the case of SoD with medium domain sizes and went up to ~4.5GBs for large domain sizes with cominiSatPS which allocated the least memory among all SAT solvers in the large domain case. The same problem

		Time (seconds)									
		cominisatps			cryptominisat			glucose-syrup			SMT
P	**Q**	S	M	L	S	M	L	S	M	L	
PR	$\neg\mathcal{F}$	4	0.6	>100	4	5.1	>100	~0	0.9	>100	0.01
PS	$\neg\mathcal{F}$	12	5.7	17.6	16	22.4	10.6	8	5.8	17.2	0.01
CI	$\mathcal{F}$	24	17.4	7	288	12.9	11.6	24	17.5	5.7	0.01
P-AH	$\neg\mathcal{F}$	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.01
G-AH	$\neg\mathcal{F}$	~0	~0	~0	~0	~0	~0	~0	~0	~0	0.01
SF	$\mathcal{F}$	~0	0.1	11.6	~0	0.6	>100	~0	0.1	12.6	0.01
SoD	$\mathcal{F}$	28	48.5	11.1	12	58.8	25.7	32	52.2	9.7	0.02

Table 4: SAT vs. SMT: computation time

required only ~2MBs of memory using SMT solving.

The results in Table 4 show that, in general, the use of SAT solvers for policy analysis performs far worse with the growth of the domain size, whereas the time necessary to prove (or disprove) that a property holds was negligible (less than 20ms) for all SMT cases. An exception is the analysis of attribute hiding for which SAT solvers offer performance similar to SMT. This is due to the fact that our example policy for attribute hiding contains very simple predicates that can be easily represented in propositional logic. Our final observation related to these experiments is that, for some properties (i.e., CI and SoD) SAT solvers perform better over large domains than the medium domains. We believe this is due to the fact that most SAT solvers apply certain optimizations such as unit propagation [10] that may allow them to simplify large formulas. Even though such optimizations may provide significant performance gains, our experiments show that such gains are negligible compared to SMT solving.

These results confirm our previous findings [9], in which we used different SAT solvers for policy analysis. In particular, the results show that, even with relatively simple policies, fine-grained policy analysis can become impractical using SAT solving due to the very large formula that is obtained from the SAT encoding of non-Boolean attributes. In contrast, the experiments shows that our SMT-based approach presents a viable alternative to the policy analysis problem.

6.3. Experiments 2: Realistic Policies

In the second set of experiments we evaluated the performance of our prototype over six realistic policies collected from various sources. IN4STARS [32] consists of a number of policies defined in the context of a project on interoperability between intelligence agencies. In addition to set membership and string equality, this policy contains various user-defined functions to determine the privileges of users according to their clearance, which have been modeled with EUF. KMarket [33] is used to manage authorizations in an on-line trading application and contains simple arithmetic operations such as less-than. GradeMan[4] is a simplified version of the access control policy used to regulate access to grades by students, faculty and alike at Brown University. Continue-a[5] is a policy used to govern the permissions of users such as authors or program committee members in a conference management system. These two policies consist mainly of string equality predicates. SAFAX is a policy used to regulate the permissions of users in a security tool[6] that implements the concept of "authorization as a service" [34]. This policy contains various constraints built upon set membership, arithmetic less-than and nested logical operators. The last policy, CodeMan [7], is used to manage authorizations in a software repository where resources are source codes and reports, and users are designers, developers and so on. This policy is characterized by the presence of temporal constraints built upon XACML function time-in-range, which have been modeled with LAI. All policies are written in XACML v2, except KMarket is specified in XACML v3.

We evaluated these policies against the properties presented in Section 4. Note that we created a modified version of the test policies for the verification of policy

[4]http://www.margrave-tool.org/v1+v2/margrave/versions/01-01/examples/college/

[5]http://www.margrave-tool.org/v1+v2/margrave/versions/01-01/examples/continue/

[6]http://security1.win.tue.nl/SAFAX

Policy	#PS	#P	#R	Translation/Solving Time (ms)													
				PR		PS		CI		P-AH		G-AH		SF		SoD	
IN4STARS	3	4	11	436	66	396	65	429	802	246	106	231	88	413	64	292	140
KMarket	1	3	12	102	52	97	56	107	537	82	46	76	58	99	46	74	81
GradeMan	11	5	5	137	54	134	64	151	644	103	48	107	65	134	54	127	103
Continue-a	111	266	298	1518	478	1483	561	1513	5322	943	289	970	278	1620	382	1067	1043
SAFAX	5	14	27	208	93	199	97	198	110	169	87	165	61	153	60	176	155
CodeMan	1	2	5	143	43	137	48	143	517	102	49	94	47	130	48	840	498

Table 5: Evaluation Results for Real-world Policies

refinement, subsumption and change-impact analysis as these properties aim at the comparison between two policies. The modified policies were obtained by changing the value of a single, randomly chosen attribute in the original policy. For change-impact analysis we limited the number of models returned by the SMT solver to 30. Finally, the query used for the evaluation of scenario finding aims to find an attribute assignment (i.e., a model) that is evaluated to Permit by the analyzed policy.

Table 5 presents the results of our experiments along with the size of policies, where #PS indicates the number of policysets, #P the number of policies and #R the number of rules in the corresponding XACML policy. The table reports the time required to build the SMT formula (*translation time*) as well as the time needed to solve the obtained SMT formula (*solving time*). In particular, translation time includes the time to both parse and encode the XACML policy and build the final SMT formula through substitution of the relevant terms in the query.

As it can be observed in Table 5, translation and solving time depend on the policy size and the property to be verified. In particular, the results show that the computation time (both for translation and solving) increases with the growth of the policy size. For instance, our prototype requires less than 500ms for the analysis of relatively small policies (i.e., IN4STARS, KMarket, GradeMan, CodeMan and SAFAX), except for change-impact analysis (we will discuss this case below). On the other hand, the analysis of Continue-a, which consists of approximately 300 rules, requires approximately 2s for building the final SMT formula and determining its satisfiability. It is worth noting, however, that the solving time does not grow as quickly as the size of policy does. This result is expected, since the analysis of a policy with our approach not only depends on the policy size but also on the type of expressions used to encode constraints (and consequently on the background theory used by the SMT solver).

The type of property to be verified has also an impact on the computation time. For instance, the translation time for policy refinement (**PR**), subsumption (**PS**) and change-impact analysis (**CI**) is in general larger than the one for the other properties. This is because two XACML policies have to be translated for these properties, whereas the other properties require the translation of a single policy. The results also show that the verification of change-impact analysis is computationally more expensive than the verification of the other properties. More specifically, determining the satisfiable models for the SMT formulas (i.e., solving time) required up to 0.8s for IN4STARS, KMarket, GradeMan, CodeMan and SAFAX, and more than 5s for Continue-a. This is due to the fact that change-impact analysis requires enumerating the differences between the two policies. As discussed in Section 6.1, these differences are iteratively computed by incrementally adding the negation of the models obtained in previous iterations as a new constraint to the original formula.

Finally, our experiments revealed no significant performance penalty when dealing with expressions containing non-Boolean attributes. For instance, the results for the policies KMarket, CodeMan and SAFAX show that policy analysis is computationally efficient when dealing with linear arithmetic expressions. Similarly, constraints expressed in the theory of EUF, used to represent user-defined functions in IN4STARS, do not have a significant impact on the performance of analysis.

7. Related Work

Several policy analysis tools have been proposed to assist policy authors in the analysis of XACML policies. Many of these tools (e.g., [7, 35, 36]) use binary decision diagrams (BDD) and multi-terminal binary decision diagrams (MTBDD) as the underlying representation of XACML policies and, in general, access control policies. The nodes of a decision diagram are used to represent Boolean variables encoding the attribute-values pairs in the policy. The terminal nodes represent the possible decisions (i.e., NotApplicable, Permit or Deny). Given an assignment of Boolean values to the variables, a path from the root to a terminal node according to the variable values indicates the result of the policy under that assignment. This policy representation has been exploited for analysis. For instance, an early version of Margrave [35] uses MTBDDs to support two types of analysis: policy querying, which analyzes access requests evaluated to a certain decision, and change-impact analysis, which is used to compare policies. Another policy analysis tool that employs BDDs for the encoding of XACML policies is XAnalyzer [7]. XAnalyzer uses a policy-based segmentation technique to detect and resolve policy anomalies such as redundancy and conflicts. Compared to our approach, BDD- and MTBB-based approaches allow the verification of XACML policies against a limited range of properties and are prone to memory blowups when the complexity of the policy/ies increases [37]. In addition, these approaches are only able to encode a fragment of XACML with simple constraints [4].

An alternative to BDD- and MTDD-based approaches is presented in [4] where policies and properties are encoded as propositional formulas and analyzed using a SAT solver. Similarly, Nelson et al. [3] extend Margrave by modeling policies and queries in first-order logic and employing Kodkod [38] to produce solutions to first-order logic formulas using SAT solving. In particular, Kodkod transforms first-order logic formulas into propositional formulas based on a finite universe-size provided by the user. However, this approach may not be complete as the analysis may miss solutions that require a universe larger than the given size [3]. The use of a finite universe-size relates to the fact that SAT solvers cannot handle non-Boolean variables and most XACML functions involving non-Boolean attributes are left uninterpreted. Thus, approaches relying on SAT solving either cannot achieve the same fine-grained analysis as our framework does or need to trade performance for better granularity. EXAM [39] combines the use of SAT solvers and MTBDD to reason on various policy properties. In particular, EXAM supports three classes of queries: *metadata* (e.g., policy creation date), *content* (e.g., number of rules) and *effect* (e.g., evaluation of certain requests). Policies and queries are expressed as Boolean formulas. These formulas are converted to MTBDDs and then combined into a single MTBDD for analysis.

Other formalisms have also been used for the analysis of XACML policies. Kolovski et al. [40] use description logic (DL) to formalize XACML policies and employs off-the-shelf DL reasoners for policy analysis. The use of DL reasoners enables the analysis on a wide subset of XACML but hinders the performance. Ramli et al. [41] and Ahn et al. [42] present a formulation of policy analysis problems in answer set programming (ASP). However, these approaches have drawbacks due to the intrinsic limitations of ASP. Unlike SMT, ASP does not support quantifiers, and cannot easily express linear arithmetic constraints. Indeed, in ASP the grounding (i.e., instantiation of variables with values) of linear arithmetic constraints either yields very large number of clauses (integers) or is not supported (reals). Crampton et al. [43] analyze PTaCL/XACML policies using PRISM, a probabilistic model-checker. In particular, PRISM is used to simulate the non-determinism of retrieving the attributes forming an access request for the analysis of attribute hiding attacks. In contrast, our work provides a general approach in which a large variety of security properties, including attribute hiding, can be analyzed. Another model checker often used for modeling and analyzing XACML policies is Alloy [44]. In particular, Alloy has been used for policy integration [45] and for detecting inconsistencies in XACML policies [46]. However, as observed in [35], Alloy can lead to intractable analyses of XACML policies. Moreover, Alloy is not able to reason over arithmetic constraints and, thus, unsuitable to deal with the analysis of real access control policies. Indeed, this type of constraints are often used to manage authorizations in online trade applications like in KMarket or to restrict access with respect to temporal constraints like in CodeMan. Arithmetic constraints can also be used to limit the number of objects that a given user can create like in SAFAX.

In summary, the approaches discussed above lack the inherent benefits of SMT: either background theories are not supported so that the attributes used in most XACML functions cannot be analyzed at a fine level of granularity, or the performance of analysis deteriorates very quickly.

The last few years have seen an increasing interest in the use of SMT solvers for the analysis of policies specified in different access control models. An example of application of SMT techniques to policy analysis is presented in [47] in which SMT solvers are used to detect conflicts and redundancies in RBAC. Here, rules specifying constraints on the assignment/activation of roles are encoded as SMT formulas with certain background theories such as enumerated data types and Linear Arithmetic over the reals/integers. Another example is the work in [48], which proposes safety analysis of UCON policies with SMT. This work presents a formalization of UCON policies in SMT along with new decidability results in terms of SMT solving.

The work in [49] shares with our approach the use of SMT solvers to support the analysis of XACML policies. In particular, this work provides an expressive logical framework, which can handle a large range of policy specifications including XACML. Similarly to our work, XACML policies can be automatically translated to expressions of the logical framework to which the available analyses (such as those considered in this paper) can be applied. In particular, a policy is encoded using two mutually exclusive predicates representing the set of access requests that are permitted and denied by the policy respectively. However, this framework does not distinguish the Indeterminate and NotApplicable decisions; as shown in Fig. 1, this distinction is necessary to correctly encode the semantics of XACML combining algorithms. Moreover, the work in [49] only presents an empirical evaluation of the proposed framework using the Continue-a dataset and synthetically generated policy datasets. In addition to testing our approach using the Continue-a dataset and other real-life policy datasets, we also study the intrinsic complexity of using SMT for policy analysis with respect to the (combination of) constraints that can often be found in policy specifications. This study is a necessary step to fully understand the potential and limitations of SMT as the underlying reasoning methods for the analysis of access control policies.

8. Conclusion

In this paper, we have presented a formal framework for the analysis XACML policies that supports the verification of a wide variety of well-known security properties. Our framework differs from existing approaches in that it uses SMT as the underlying reasoning mechanism. In particular, it reduced XACML policy analysis problems to SMT problems and uses a state-of-the-art SMT solver for solving such problems. The main advantage of SMT compared to other reasoning techniques lies in the ability of SMT to reason over non-Boolean attributes, which are often used in access control and, in particular, in XACML policies. This allows a fine-grained analysis, relieving policy authors of a compromise between performance and accuracy of the analysis. We evaluated a prototype implementation of our approach and compared it with off-the-shelf SAT solvers. The experiments show promising results and, in particular, a major improvement (order of magnitude) in terms of both memory usage and computational time compared to SAT solvers. Moreover, the experiments show that our SMT-based policy analysis framework is able to cope with realistic policies.

The analysis capabilities of our framework are bound to the background theories available for the SMT solver. In our encoding of XACML policies, we have used various theories to encode XACML functions. However, a specific background theory is not available for all functions provided by the XACML standard (e.g., XPath-based and regular-expression-based functions). In our work, functions for which a theory is not available, were left uninterpreted and addressed with the theory of equality with uninterpreted functions. The development of new background theories, i.e. being able to reason over these functions, will provide our framework with additional analysis capabilities.

As future work, we are planning to further investigate the expressiveness of our policy analysis framework by encoding additional security properties for the analysis of XACML policies and evaluate the performance of our prototype against a larger set of real-world policies. This experimental analysis will help us optimize the efficiency of our approach.

Acknowledgments. This work has been funded by the ITEA2 project M2MGrids (13011) and the Dutch national program COMMIT under the THeCS project.

References

[1] OASIS XACML Technical Committee, eXtensible Access Control Markup Language (XACML) Version 3.0, 2013.

[2] A. Buecker, C. Forster, S. Muppidi, B. Safabakhsh, IBM Tivoli Security Policy Manager, IBM Red Books (2009).

[3] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, S. Krishnamurthi, The Margrave Tool for Firewall Analysis, in: Proceedings of the 24th International Conference on Large Installation System Administration, USENIX Association, 2010, pp. 1–8.

[4] G. Hughes, T. Bultan, Automated verification of access control policies using a SAT solver, International Journal on Software Tools for Technology Transfer 10 (2008) 503–520.

[5] M. Backes, G. Karjoth, W. Bagga, M. Schunter, Efficient comparison of enterprise privacy policies, in: Proceedings of Symposium on Applied Computing, ACM, 2004, pp. 375–382.

[6] J. Crampton, C. Morisset, PTaCL: A Language for Attribute-Based Access Control in Open Systems, in: Principles of Security and Trust, LNCS 7215, Springer, 2012, pp. 390–409.

[7] H. Hu, G.-J. Ahn, K. Kulkarni, Discovery and Resolution of Anomalies in Web Access Control Policies, IEEE Transactions on Dependable and Secure Computing 10 (2013) 341–354.

[8] F. Turkmen, S. N. Foley, B. O'Sullivan, W. M. Fitzgerald, T. Hadzic, S. Basagiannis, M. Boubekeur, Explanations and relaxations for policy conflicts in physical access control, in: Proceedings of International Conference on Tools with Artificial Intelligence, IEEE, 2013, pp. 330–336.

[9] F. Turkmen, J. den Hartog, S. Ranise, N. Zannone, Analysis of XACML

policies with SMT, in: Principles of Security and Trust, LNCS 9036, Springer, 2015, pp. 115–134.

[10] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability, IOS Press, 2008, pp. 825–885.

[11] C. P. Gomes, H. Kautz, A. Sabharwal, B. Selman, Satisfiability Solvers, in: Handbook of Knowledge Representation, Foundations of Artificial Intelligence 3, Elsevier, 2008, pp. 89–134.

[12] C. Morisset, N. Zannone, Reduction of access control decisions, in: Proceedings of Symposium on Access Control Models and Technologies, ACM, 2014, pp. 53–62.

[13] H. B. Enderton, A Mathematical Introduction to Logic, Academic Press, 1972.

[14] Y. Ge, L. de Moura, Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories, in: Computer Aided Verification, LNCS 5643, Springer, 2009, pp. 306–320.

[15] G. Nelson, D. Oppen, Simplification by cooperating decision procedures, ACM Trans. Programm. Lang. Syst. 1 (1979) 245–257.

[16] D. Oppen, Complexity, convexity and combinations of theories, Theor. Comput. Sci. 12 (1980) 291–302.

[17] OASIS XACML Technical Committee, eXtensible Access Control Markup Language (XACML) Version 2.0, 2005.

[18] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: a Z3-based string solver for web application analysis, in: Proceedings of Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 114–124.

[19] D. Kröning, G. Weissenbacher, A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard, in: Proceedings of International Workshop on Satisfiability Modulo Theories.

[20] P. Suter, R. Steiger, V. Kuncak, Sets with cardinality constraints in satisfiability modulo theories, in: Verification, Model Checking, and Abstract Interpretation, LNCS 6538, Springer, 2011, pp. 403–418.

[21] J. R. Burch, D. L. Dill, Automatic verification of pipelined microprocessor control, in: Computer Aided Verification, LNCS 818, pp. 68–80.

[22] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, S. Krishnamurthi, Aluminum: principled scenario exploration through minimality, in: Proceedings of International Conference on Software Engineering, IEEE, 2013, pp. 232–241.

[23] S. Saghafi, R. Danas, D. J. Dougherty, Exploring Theories with a Model-Finding Assistant, in: Automated Deduction, LNCS 9195, Springer, 2015, pp. 434–449.

[24] Role-Based Access Control, National Institute of Standards and Technology (NIST) (2004).

[25] OASIS XACML Technical Committee, XACML v3.0 Core and Hierarchical Role Based Access Control (RBAC) Profile, 2014.

[26] D. Kroening, O. Strichman, Decision Procedures - An Algorithmic Point of View, Texts in Theoretical Computer Science, Springer, 2008.

[27] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Ranise, R. Sebastiani, Efficient Theory Combination via Boolean Search, Information and Computation 204 (2006).

[28] D. Kroening, O. Strichman, A framework for Satisfiability Modulo Theories, Formal Asp. Comput. 21 (2009) 485–494.

[29] F. Turkmen, J. den Hartog, N. Zannone, Analyzing Access Control Policies with SMT, in: Proceedings of Conference on Computer and Communications Security, ACM, 2014, pp. 1508–1510.

[30] L. M. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Tools and Algorithms for the Construction and Analysis of Systems, LNCS 4963, Springer, 2008, pp. 337–340.

[31] N. Tamura, A. Taga, S. Kitagawa, M. Banbara, Compiling finite linear CSP into SAT, Constraints 14 (2009) 254–272.

[32] A. I. Egner, D. Luu, J. den Hartog, N. Zannone, An Authorization Service for Collaborative Situation Awareness, in: Proceedings of Conference on Data and Application Security and Privacy, ACM, 2016, pp. 136–138.

[33] Balana: Open source XACML 3.0 implementation, `http://xacmlinfo.org/category/balana`, 2013.

[34] S. P. Kaluvuri, A. I. Egner, J. den Hartog, N. Zannone, SAFAX - an extensible authorization service for cloud environments, Front. ICT 2015 (2015).

[35] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, M. C. Tschantz, Verification and Change-impact Analysis of Access-control Policies, in: Proceedings of International Conference on Software Engineering, ACM, 2005, pp. 196–205.

[36] B. Bahrak, A. Deshpande, M. Whitaker, J. M. Park, BRESAP: A Policy Reasoner for Processing Spectrum Access Policies Represented by Binary Decision Diagrams, in: Proceedings of IEEE Symposium on New Frontiers in Dynamic Spectrum, pp. 1–12.

[37] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, Y. Zhu, Symbolic Model Checking Using SAT Procedures Instead of BDDs, in: Proceedings of Annual ACM/IEEE Design Automation Conference, ACM, 1999, pp. 317–320.

[38] E. Torlak, D. Jackson, Kodkod: A relational model finder, in: Tools and Algorithms for the Construction and Analysis of Systems, LNCS 4424, Springer, 2007, pp. 632–647.

[39] D. Lin, P. Rao, E. Bertino, N. Li, J. Lobo, EXAM: a comprehensive environment for the analysis of access control policies, Int. J. Inf. Sec. 9 (2010) 253–273.

[40] V. Kolovski, J. A. Hendler, B. Parsia, Analyzing web access control policies, in: Proceedings of International Conference on World Wide Web, ACM, 2007, pp. 677–686.

[41] C. D. P. K. Ramli, H. R. Nielson, F. Nielson, XACML 3.0 in Answer Set Programming, in: Logic-Based Program Synthesis and Transformation, LNCS 7844, Springer, 2012, pp. 89–105.

[42] G.-J. Ahn, H. Hu, J. Lee, Y. Meng, Representing and reasoning about web access control policies, in: Proceedings of Annual Computer Software and Applications Conference, IEEE, 2010, pp. 137–146.

[43] J. Crampton, C. Morisset, N. Zannone, On Missing Attributes in Access Control: Non-deterministic and Probabilistic Attribute Retrieval, in: Proceedings of Symposium on Access Control Models and Technologies, ACM, 2015, pp. 99–109.

[44] D. Jackson, Alloy: A Lightweight Object Modelling Notation, ACM Trans. Softw. Eng. Methodol. 11 (2002) 256–290.

[45] M. Toahchoodee, I. Ray, Validation of Policy Integration Using Alloy, in: Distributed Computing and Internet Technology, LNCS 3816, Springer, 2005, pp. 420–431.

[46] M. Mankai, L. Logrippo, Access control policies: Modeling and validation, in: Proceedings of the 5th Colloque International sur les Nouvelles Technologies de la Repartition, pp. 85–91.

[47] A. Armando, S. Ranise, Automated and efficient analysis of role-based access control with attributes, in: Data and Applications Security and Privacy XXVI, LNCS 7371, Springer, 2012, pp. 25–40.

[48] S. Ranise, A. Armando, On the automated analysis of safety in usage control: A new decidability result, in: Network and System Security, LNCS 7645, Springer, 2012, pp. 15–28.

[49] K. Arkoudas, R. Chadha, C. J. Chiang, Sophisticated access control via SMT and logical frameworks, ACM Transactions on Information and System Security 16 (2014) 17.

Appendix A. Encoding of Combining Algorithms in XACML v2

Fig. A.2 presents the encoding of the combining algorithms as supported by XACML v2 [17]. These encodings are used by function applyCA to evaluate policies expressed in XACML v2. Differently from XACML v3, all combining algorithms in XACML v2 are defined over a four-valued decision set (i.e., *Permit*, *Deny*, *Indeterminate* and *NotApplicable*). Moreover, XACML v2 only supports combining algorithms permit-overrides, deny-overrides, first-applicable and only-one-applicable. In particular, XACML v3 uses algorithms first-applicable and

Deny-overrides (Rules)

$$\begin{aligned}
DS^{\mathsf{rdov}}_{D} &= DS^{p_1}_{D} \cup DS^{p_2}_{D}\\
DS^{\mathsf{rdov}}_{P} &= (DS^{p_1}_{P} \cup DS^{p_2}_{P}) \setminus (DS^{\mathsf{rdov}}_{D} \cup DS^{\mathsf{rdov}}_{IN(D)})\\
DS^{\mathsf{rdov}}_{IN} &= DS^{\mathsf{rdov}}_{IN(D)} \cup DS^{\mathsf{rdov}}_{IN(P)}\\
DS^{\mathsf{rdov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}\\
DS^{\mathsf{rdov}}_{IN(D)} &= (DS^{p_1}_{IN(D)} \cup DS^{p_2}_{IN(D)}) \setminus DS^{\mathsf{rdov}}_{D}\\
DS^{\mathsf{rdov}}_{IN(P)} &= (DS^{p_1}_{IN(P)} \cup DS^{p_2}_{IN(P)}) \setminus\\
&\quad (DS^{\mathsf{rdov}}_{D} \cup DS^{\mathsf{rdov}}_{IN(D)} \cup DS^{\mathsf{rdov}}_{P})
\end{aligned}$$

Deny-overrides (Policies)

$$\begin{aligned}
DS^{\mathsf{pdov}}_{D} &= DS^{p_1}_{D} \cup DS^{p_2}_{D} \cup DS^{p_1}_{IN} \cup DS^{p_2}_{IN}\\
DS^{\mathsf{pdov}}_{P} &= (DS^{p_1}_{P} \cup DS^{p_2}_{P}) \setminus DS^{\mathsf{pdov}}_{D}\\
DS^{\mathsf{pdov}}_{IN} &= \emptyset\\
DS^{\mathsf{pdov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Permit-overrides (Rules)

$$\begin{aligned}
DS^{\mathsf{rpov}}_{D} &= (DS^{p_1}_{D} \cup DS^{p_2}_{D}) \setminus (DS^{p}_{P} \cup DS^{\mathsf{rpov}}_{IN(P)})\\
DS^{\mathsf{rpov}}_{IN} &= DS^{\mathsf{rpov}}_{IN(P)} \cup DS^{\mathsf{rpov}}_{IN(D)}\\
DS^{\mathsf{rpov}}_{P} &= DS^{p_1}_{P} \cup DS^{p_2}_{P}\\
DS^{\mathsf{rpov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}\\
DS^{\mathsf{rpov}}_{IN(P)} &= (DS^{p_1}_{IN(P)} \cup DS^{p_2}_{IN(P)}) \setminus DS^{\mathsf{rpov}}_{P}\\
DS^{\mathsf{rpov}}_{IN(D)} &= (DS^{p_1}_{IN(D)} \cup DS^{p_2}_{IN(D)}) \setminus\\
&\quad (DS^{\mathsf{rpov}}_{P} \cup DS^{\mathsf{rpov}}_{IN(P)} \cup DS^{\mathsf{rpov}}_{D})
\end{aligned}$$

Permit-overrides (Policies)

$$\begin{aligned}
DS^{\mathsf{ppov}}_{P} &= DS^{p_1}_{P} \cup DS^{p_2}_{P}\\
DS^{\mathsf{ppov}}_{D} &= (DS^{p_1}_{D} \cup DS^{p_2}_{D}) \setminus DS^{\mathsf{ppov}}_{P}\\
DS^{\mathsf{ppov}}_{IN} &= (DS^{p_1}_{IN} \cup DS^{p_2}_{IN}) \setminus (DS^{\mathsf{ppov}}_{P} \cup DS^{\mathsf{ppov}}_{D})\\
DS^{\mathsf{ppov}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

First-applicable

$$\begin{aligned}
DS^{\mathsf{fa}}_{D} &= DS^{p_1}_{D} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{D})\\
DS^{\mathsf{fa}}_{P} &= DS^{p_1}_{P} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{P})\\
DS^{\mathsf{fa}}_{IN} &= DS^{p_1}_{IN} \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{IN})\\
DS^{\mathsf{fa}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Only-one-applicable

$$\begin{aligned}
DS^{\mathsf{ooa}}_{D} &= (DS^{p_1}_{D} \cap DS^{p_2}_{NA}) \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{D})\\
DS^{\mathsf{ooa}}_{P} &= (DS^{p_1}_{P} \cap DS^{p_2}_{NA}) \cup (DS^{p_1}_{NA} \cap DS^{p_2}_{P})\\
DS^{\mathsf{ooa}}_{IN} &= (DS^{p_1}_{P} \cap DS^{p_2}_{P}) \cup (DS^{p_1}_{P} \cap DS^{p_2}_{D}) \cup\\
&\quad (DS^{p_1}_{D} \cap DS^{p_2}_{P}) \cup (DS^{p_1}_{D} \cap DS^{p_2}_{D}) \cup\\
&\quad DS^{p_1}_{IN} \cup DS^{p_2}_{IN}\\
DS^{\mathsf{ooa}}_{NA} &= DS^{p_1}_{NA} \cap DS^{p_2}_{NA}
\end{aligned}$$

Figure A.2: Encoding of XACML v2 combining algorithms. In the figure, p_1 and p_2 denote two policy elements with decision space $\langle DS^{p_1}_{P}, DS^{p_1}_{D}, DS^{p_1}_{IN}, DS^{p_1}_{NA}\rangle$ and $\langle DS^{p_2}_{P}, DS^{p_2}_{D}, DS^{p_2}_{IN}, DS^{p_2}_{NA}\rangle$ respectively.

only-one-applicable as defined in XACML v2. Thus, the encoding of these algorithms in Fig. A.2 is equivalent to the encoding given in Fig. 1. On the other hand, combining algorithms permit-overrides and deny-overrides have been redefined in XACML v3. It is worth noting that XACML v2 provides two variants of these combining algorithms: one variant for rules and one for policies. The variants for rules implicitly record the effect of the rules that are evaluated *Indeterminate*. This is similar to the idea underlying the *Indeterminate* extended set used in XACML v3. In Fig. A.2, we use decision spaces $DS_{IN(D)}$ and $DS_{IN(P)}$ to make the effect of the rules evaluated *Indeterminate* explicit, where $DS_{IN(D)}$ indicates that the effect is *Deny* and $DS_{IN(P)}$ indicates that the effect is *Permit*.

Appendix B. Proof of Theorem 1

Theorem 1 states that the semantics we use to encode policies into SMT formulas correctly captures policy evaluation according to the XACML specification [1]. In this section we provide sketches of the proofs of this Theorem and the prerequisite Proposition 1 that states that the combining algoirthms are correctly captured. As applyCA and our policy evaluation procedure are a direct translation of combining algorithms respectively the evaluation defined in the XACML specification, these proofs consist of rather straightforward case distinctions.

Recall that the we use combining algorithm value to refer to the intermediate that the XACML calls either 'Specified by the rule-combining algorithm' or 'Combing Algorithm Value'.

Proposition 1 *Function* applyCA *(as in Figure 1) correctly combines the decision spaces, i.e. it returns the decision space for the 'Specified by the rule-combining algorithm' value (also called 'Combining Algorithm Value') of [1].*

Proof *The combining algorithms (as in Figure 1) are defined according to their specification in [1, Section C]. Here, we treat combining algorithm* deny-overrides *(with two child policies) and assume it is clear how to extend this to other algorithms (and more child policies).*

The specification of deny-overrides *[1, Section C.2] states seven requirements which are captured by the equations given in Figure 1. The first requirement defines the deny space by stating that "If any decision is* Deny *then result is* Deny*" (and this is the only case where* deny-overrides *gives* Deny*). This is captured by the first equations in Figure 1:* $DS_D^{\mathsf{dov}} = DS_D^{p_1} \cup DS_D^{p_2}$*. The other requirements start with 'otherwise' indicating that previously covered cases should be excluded. This is achieved by using set minus the union of already defined spaces. The second and third requirements give the cases when* Indeterminate{PD} *should be obtained: "if*

Target	**CA Value**	**Policy set Value**
Match	x	x
No-match	Don't care	NotApplicable
Indeterminate	NotApplicable	NotApplicable
Indeterminate	Permit	Indeterminate{P}
Indeterminate	Deny	Indeterminate{D}
Indeterminate	Indeterminate{PD}	Indeterminate{PD}
Indeterminate	Indeterminate{P}	Indeterminate{P}
Indeterminate	Indeterminate{D}	Indeterminate{D}

Table B.6: Policy set evaluation. Table 6 and 7 of [1] merged.

any decision is Indeterminate{PD}*" (captured by $DS^{p_1}_{IN(PD)} \cup DS^{p_2}_{IN(DP)}$) or "if any decision is* Indeterminate{D} *and another decision is* Indeterminate{PD} *or* Permit*"(captured by $DS^{p_1}_{IN(D)} \cap (DS^{p_2}_{IN(P)} \cup DS^{p_2}_{P})$ and visa versa). Thus, also this space is correctly capture by the equations in Figure 1). The other spaces and other algorithms are similar.*

The proof (sketch) above illustrates that it is straightforward to translate the XACML combining algorithms into operations on decision spaces.

Theorem 2 (Policy Evaluation) *The recursive equations given in Theorem 1 correctly compute the decision spaces as prescribed by the XACML policy evaluation.*

Proof *As the equation system is clearly well-defined with a unique solution we only need to check whether the decision spaces satisfy the given equations. This is clear by comparing the equations of Theorem 1 with [1, Section 7.11] for rules and [1, Section 7.12-7.14] for policies and policy sets (see also Table B.6). We treat one case for illustrative purposes:*
According to [1, Section 7.13, 7.14] for a policy set p with target T, a request r evaluates to Indeterminate{P} *(i.e. $r \in DS^{p}_{IN(P)}$) in the following cases: (1) the target evaluates to "Match" and the Combining Algorithm Value is equal to* Indeterminate{P}*, (2) the target evaluates to "Indeterminate" and the Combining Algorithm (CA) Value is equal to* Indeterminate{P}*, or (3) the target evaluates*

to "Indeterminate" and the Combining Algorithm (CA) Value is equal to Permit. *By Proposition 1 this exactly corresponds to (1) $r \in AS^T_A \cap DS^{ca}_{IN(P)}$, (2) $r \in AS^T_{IN} \cap DS^{ca}_{IN(P)}$ or (3) $r \in AS^T_{IN} \cap DS^{ca}_P$ respectively. Thus, the equation $DS^p_{IN(P)} = (AS^T_A \cup AS^T_{IN}) \cap DS^{ca}_{IN(P)} \cup (AS^T_{IN} \cap DS^{ca}_P)$ holds.*