Analysis of XACML Policies with SMT

Fatih Turkmen[1], Jerry den Hartog[1], Silvio Ranise[2], and Nicola Zannone[1]

[1] Eindhoven University of Technology, Eindhoven, Netherlands
[2] Fondazione Bruno Kessler (FBK) Trento, Italy

Abstract. The eXtensible Access Control Markup Language (XACML) is an extensible and flexible XML language for the specification of access control policies. However, the richness and flexibility of the language (along with the verbose syntax of XML) come with a price: errors are easy to make and difficult to detect when policies grow in size. If these errors are not detected and rectified, they can result in serious data leakage and/or privacy violations leading to significant legal and financial consequences. To assist policy authors in the analysis of their policies, several policy analysis tools have been proposed based on different underlying formalisms. However, most of these tools either abstract away functions over non-Boolean domains (hence they cannot provide information about them) or produce very large encodings which hinder the performance. In this paper, we present a generic policy analysis framework that employs SMT as the underlying reasoning mechanism. The use of SMT does not only allow more fine-grained analysis of policies but also improves the performance. We demonstrate that a wide range of security properties proposed in the literature can be easily modeled within the framework. A prototype implementation and its evaluation are also provided.

1 Introduction

Access rules governing sensitive data such as patient health records or financial transactions are usually encoded in a policy that is enforced by the authorization system. Correctness of access control policies is crucial for organizations to prevent authorization violations or fraud which can result in serious data leakage and/or privacy violations leading to significant legal and financial consequences (e.g., financial and reputation loss). In this work, we consider policies expressed in eXtensible Access Control Markup Language (XACML) [20]. XACML provides an extensible and flexible language that allows the specification of structured policies in which policies specified by different authorities can be combined together. However, policy specification in XACML is known to be a difficult and error-prone task [10, 13]. This richness and flexibility along with its verbose syntax make it difficult to determine whether policies work as intended. Therefore, automated tools are needed to assist policy authors in analyzing their policies to detect and correct errors before policies are deployed.

This need has spurred the development of several methods and tools for the verification of policy specifications at design time using formal reasoning [5, 8, 10, 12, 13]. The security properties being verified can express requirements on the

policies but also on relations between policies. A requirement on a policy could specify (types of) access requests that should (not) be granted by the policy. An updated policy being compared with the original to ensure the update is 'safe' is an example of a requirement on the relation between policies. Here 'safe' could be expressed in being as permissive/restrictive as another policy as is done in policy refinement [5] and subsumption [13]. Despite a large variety in security properties that one may need to check, existing policy analysis tools often support only a restricted set of properties due to the (lack of) expressiveness of the formalization employed by the tool and the capabilities offered by the underlying reasoner.

Advances in propositional satisfiability (SAT) research [11] make SAT solvers an attractive underlying reasoner in policy analysis [13]. SAT allows efficient reasoning about propositional logic formula and many access control policies and security properties can be naturally modeled in propositional logic. However, SAT solvers do not natively support reasoning on predicates over non-Boolean variables and functions which frequently appear in access control policies and, in particular, in XACML policies. For instance, SAT does not allow a straightforward reasoning on temporal constraints such as *request-time* $>$ 13:20, which can play an important role in the correctness of a policy and thus in the security of the system. Such non-Boolean expressions are usually left uninterpreted [13] which restricts analysis capabilities. Alternatives that support fine-grained policy analysis can lead to excessively large encodings of the policy. The analyst is forced to choose a trade-off between performance and accuracy by introducing bounds on the domains.

In this paper, we consider SAT modulo theories (SMT) [6] as the underlying reasoning method for the analysis of XACML policies. SMT enables the use of theories, such as linear arithmetic and equality, to reason about the satisfiability of first order formulas. SMT is a natural extension to SAT in which SMT solvers employ tailored reasoners when solving non-Boolean predicates in the input formula. The use of SMT makes it possible to perform a more fine-grained analysis than existing SAT-based policy analysis tools allow.

The contributions of this paper are thus as follows:

- A novel policy analysis framework which makes it possible to verify access control policies against a large range of security properties.
- A fine-grained analysis of access control policies by performing reasoning on non-Boolean predicates, e.g. arithmetic functions on numeric attributes.
- A prototype implementation of the framework and its extensive evaluation using a number of well-known security properties taken from the literature.

The remainder of the paper provides an overview of XACML and SMT in Section 2, and an encoding of XACML policies in SMT in Section 3. Our analysis framework that uses this encoding for policy analysis is given in Section 4. Section 5 presents a prototype of our framework with experimental results. Section 6 discusses related work, and Section 7 provides conclusions.

2 Preliminaries

In this section we shortly recall key points of XACML and SMT.

2.1 XACML

XACML [20] 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 [20] (without obligations).

Three policy elements are provided by XACML: *policy sets*, *policies* and *rules*. A policy set consists of policy sets and policies; policies in turn consist of rules. If policy element p_1 is nested in policy element p_2 we say that p_1 is a *child policy element* of p_2 and 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 in terms of attributes characterizing the subject, the resource, the action to be performed on the resource, and the environment. Intuitively, the target identifies the set of access requests that the policy element applies to. In addition, rules specify an *effect* element that defines whether the requested actions should be allowed (*Permit*) or denied (*Deny*), and can be associated with *conditions* to further restrict their applicability.

If an access request matches both the target and conditions of a rule, the rule is applicable to the request and yields the decision specified by its effect element. Otherwise, the rule is not applicable, and a *NotApplicable* decision is returned. If an error occurs during evaluation, an *Indeterminate* decision is returned. XACML v3 also introduces an extended set of *Indeterminate* values to allow a fine-grained combination of decisions: *Indeterminate*$\{P\}$, *Indeterminate*$\{D\}$ and *Indeterminate*$\{PD\}$. Intuitively, these *Indeterminate* decisions indicate the evaluation result of a policy element if the error not occurred.

To combine decisions obtained from the evaluation of different applicable policy elements, XACML provides a number of combining algorithms [20]: permit-overrides, deny-overrides, deny-unless-permit, permit-unless-deny, first-applicable and only-one-applicable.[3] Intuitively, these algorithms define procedures to evaluate composite policies based on the order of the policy elements and priorities between decisions.

Next we present a sample XACML policy in a concise form that we will use as a running example through the paper.

Example 1 *A user is allowed to create an object of type "transaction" only if his credit balance (*credit*) is higher than the value of the transaction itself (*value*) and banking costs (*cost*). Transactions can only be created during working days*

[3] Combining algorithms permit-overrides and deny-overrides are defined over the *Indeterminate* extended set, while the other algorithms are defined over a single *Indeterminate* decision value. Combining algorithm only-one-applicable can only be used to combine policy sets and policies.

(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 and then to combine the resulting rules using* deny-overrides *(*dov*):*

$$
\begin{array}{ll}
p[\mathsf{dov}]: & \mathsf{resource\text{-}type} = \text{“}transaction\text{”} \wedge \mathsf{action\text{-}id} = \text{“}create\text{”} \\
r_1[Deny]: & \mathsf{value} + \mathsf{cost} > \mathsf{credit} \\
r_2[Deny]: & \mathsf{current\text{-}day} \notin \{Mo, Tu, We, Th, Fr\} \vee \\
 & \mathsf{current\text{-}time} < 08{:}00 \vee \mathsf{current\text{-}time} > 18{:}00 \\
r_3[Permit]: & \textbf{true}
\end{array}
$$

where **true** *is used to indicate that the target of the policy element 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 a policy element returns* Indeterminate *if multiple values are provided for them.*

2.2 Satisfiability Modulo Theories

SMT [6] 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 an extended version of the SMT-LIB (v2) standard (`http://www.smtlib.org`) which is based on many-sorted first order logic. In the remainder, we assume the usual syntactic (e.g., sort, constant, predicate and function symbols, terms, atoms, literals, Boolean connectives, quantifiers, and formulas) and semantic (e.g., structure, satisfaction, model, and validity) notions of many-sorted first order logic; see [9] for formal definitions.

A theory $\mathcal{T}$ consists of a signature and a class of models. Intuitively, 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. For Linear Arithmetic over the Integers (LAI), the signature consists of the numerals (corresponding to the 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. For 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 the negation of φ (i.e., $\neg\varphi$) is $\mathcal{T}$-unsatisfiable.

Checking the satisfiability of conjunctions of literals (i.e., atoms or their negations) modulo certain theories – e.g., the theory of uninterpreted functions, theories of enumerated data-types, and Linear Arithmetic over the Integers – is well-known to be decidable [6]. These results imply the decidability of checking the satisfiability of quantifier-free formulas modulo the same theories. This is so as it is always possible to transform arbitrary Boolean combinations of atoms into disjunctive normal form (DNF), i.e. in a disjunction of conjunctions of literals. Unfortunately, the transformation to DNF may be computationally expensive and generate an exponentially larger formula [9]. For this reason, even if checking the satisfiability of conjunctions of literals modulo certain theories is polynomial (as it is the case for the theory of uninterpreted functions), checking the satisfiability of quantifier-free formulas modulo the same theories becomes NP-hard. While these theoretical limitations are unavoidable, modern SMT solvers have developed a wealth of heuristics to scale and handle very large formulas with arbitrary Boolean structures. The interested reader is pointed to [6] for a thorough introduction.

The situation is further complicated by two possible sources of problems. First, several verification problems (such as the XACML policy analysis problems considered in this paper) require to consider more than one theory to model various aspects of the situation under scrutiny. Under suitable assumption on the component theories, it is possible to build theory solvers capable of checking the satisfiability of conjunctions of literals in combinations of theories by modularly re-using the theory solvers of the component theories. However, the complexity of checking the satisfiability of conjunctions of literals in the combination can be much higher than that of modulo the individual theories. For instance, there exists a combination of two theories with polynomial satisfiability problem whose combination becomes NP-complete [22]. The second source of problems is the presence of quantifiers in the proof obligations generated by certain verification tools (as it is the case of some of the policy analysis problems considered in this work). In fact, the decidability of quantifier-free formulas does not extend to quantified formulas. For instance, checking the satisfiability of quantified formulas modulo the theory of uninterpreted functions is undecidable since one can encode the satisfiability problem for arbitrary first-order formulas whose undecidability is well-known [9]. Despite this and other negative results, several efforts have been put in identifying classes of quantified formulas whose satisfiability is decidable by integrating instantiation or quantifier-elimination procedures in SMT solvers; see, e.g., [6] for pointers to relevant work.

3 Encoding XACML policies in SMT

In this section, we first present our formalization of XACML policies that allows us to represent policies in terms of predicates. We then show how the obtained predicates can be used to define SMT formulas.

3.1 XACML Formalization

An access control schema $\langle Att, Dom \rangle$ defines the vocabulary used for specifying access control policies. Here Att is a set of attributes $a_1, \ldots, a_n$, Dom gives the corresponding attribute domains $Dom_{a_1}, \ldots, Dom_{a_n}$ and we refer to set $2^{Dom_{a_1}} \times \ldots \times 2^{Dom_{a_n}}$ as the *policy space* specified within the schema. The elements of the policy space are called *attribute assignments*. An attribute assignment maps attributes to a (possibly empty) set of values in their domains. An *access request* $\langle a_1 = v_{1_i}, \ldots, a_n = v_{n_k} \rangle$ (with $v_{1_i} \in Dom_{a_1}, \ldots, v_{n_k} \in Dom_{a_n}$) specifies an attribute assignment, provided the values for those attributes are not assigned the empty set (multiple attribute/value pairs with the same attribute indicate multiple values are assigned to that attribute). Hereafter, $\mathcal{R}$ denotes the set of all possible access requests, i.e. the policy space.

Each policy element in XACML has a target that specifies *applicability constraints* in terms of attribute assignments. Applicability constraints are used to divide the policy space in three disjoint sub-spaces: the *Applicable* space AS_A, the *Indeterminate* space AS_{IN}, and the *NotApplicable* space AS_{NA}. These sub-spaces respectively represent access requests for which the policy's target matches the request, checking whether the target matches the request produces an error, and the target does not match the request. We represent the applicability space of a policy element as $\langle AS_A, AS_{IN} \rangle$ with an access request req in the set AS_{NA} (in symbols, $req \in AS_{NA}$) iff $req \notin AS_A \cup AS_{IN}$. An access request is evaluated against a policy element only if it matches the target of policy element's parent. Based on this observation, we flatten a XACML policy by propagating its applicability constraints in a top-down fashion from the root policy element to rules.

Definition 1 *Let p be a policy where $\langle AS_A^T, AS_{IN}^T \rangle$ is the applicability space induced by its target. The applicability space of p is inductively given by:*

$$\langle AS_A^p, AS_{IN}^p \rangle = \begin{cases} \langle AS_A^T, AS_{IN}^T \rangle & \text{if } p \text{ is a root policy} \\ \langle AS_A^T \cap AS_A^q, (AS_{IN}^T \cap AS_A^q) \cup AS_{IN}^q \rangle & \text{if } q \text{ is the parent of } p \end{cases}$$

For the root policy (i.e., the policy that does not have a parent policy element), the applicability space is that induced by its target. For policies that do have a parent the applicability space of the parent is also taken into account so the parents applicability is iteratively propagated to all its child policies. Thus, a rule has an applicability space which is determined by the applicability constraints in its target and by the applicability constraints in the target of all its ancestor policy elements. Note that, as for any target AS_A^T and AS_{IN}^T are disjoint, a straightforward inductive arguments shows that AS_A^p and AS_{IN}^p are also disjoint.

Example 2 *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 att is current-day, current-time, credit, cost, *and* value *in* ac_6, ac_7, ac_8, ac_9, *and* ac_{10}, *respectively. Constraints* $ac_6, ..., ac_{10}$ *address* Indeterminate *cases by requiring att to be either empty or to contain at least two distinct elements (denoted by* v_1 *and* v_2*). The applicability space induced by the target of rule* r_i, $\langle AS_A^{T_i}, AS_{IN}^{T_i} \rangle$, *can be represented as follows (for the sake of simplicity, we represent sets of access requests as the applicability constraints that render them):*
$T_1 : \langle ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}, ac_8 \cup ac_9 \cup ac_{10} \rangle$
$T_2 : \langle (ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)}, ac_6 \cup ac_7 \rangle$
$T_3 : \langle \mathcal{R}, \emptyset \rangle$
Policy p has applicability space $\langle ac_0 \cap ac_1, \emptyset \rangle$*; this space has to be propagated to rules. Thus, the applicability space* $\langle AS_A^{r_i}, AS_{IN}^{r_i} \rangle$ *of rule* r_i *is:*
$r_1 : \langle ac_0 \cap ac_1 \cap ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}, (ac_8 \cup ac_9 \cup ac_{10}) \cap (ac_0 \cap ac_1) \rangle$
$r_2 : \langle ac_0 \cap ac_1 \cap (ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)}, (ac_6 \cup ac_7) \cap (ac_0 \cap ac_1) \rangle$
$r_3 : \langle ac_0 \cap ac_1, \emptyset \rangle$

Based on the possible decisions in XACML, the policy space can be partitioned into four disjoint subsets DS_P, DS_D, DS_{IN} and DS_{NA} by using rule effects and applicability constraints. These subsets represent the classes of access requests that evaluate to same access decision: *Permit*, *Deny*, *Indeterminate* and *NotApplicable*, respectively. We denote the decision space of a policy as $\langle DS_P, DS_D, DS_{IN} \rangle$. If an access request does not fall in $DS_P \cup DS_D \cup DS_{IN}$, then it falls in DS_{NA}. The decision space of a rule can be derived from its effect and applicability constraints.

Definition 2 *Let* $\langle AS_A, AS_{IN} \rangle$ *be the applicability space of a rule r and* Effect *its effect. The decision space of r, denoted* $\langle DS_P, DS_D, DS_{IN} \rangle$*, is*

$$DS_P = \begin{cases} AS_A \text{ if Effect} = \text{Permit} \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$DS_D = \begin{cases} AS_A \text{ if Effect} = \text{Deny} \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$DS_{IN} = AS_{IN}$$

In order to obtain the decision space of the root policy element, the decision space of child policy elements have to be recursively combined in a bottom-up fashion according to specified combining algorithms. As noted in Section 2.1 some combining algorithms use an extended decision set in which the *Indeterminate* space is subdivided into three parts. For these we extend the decision space accordingly. Here, we show the decision space of a policy with respect to deny-overrides as an example. The other combining algorithms can be defined in a similar way. Let $\langle DS_P^{p_1}, DS_D^{p_1}, DS_{IN(P)}^{p_1}, DS_{IN(D)}^{p_1}, DS_{IN(PD)}^{p_1} \rangle$ and

$\langle DS_P^{p_2}, DS_D^{p_2}, DS_{IN(P)}^{p_2}, DS_{IN(D)}^{p_2}, DS_{IN(PD)}^{p_2} \rangle$ be the (extended) decision spaces of policy elements p_1 and p_2, respectively. We are interested in the decision space $\langle DS_P^p, DS_D^p, DS_{IN}^p \rangle$ of a policy p which combines policy elements p_1 and p_2 using deny-overrides. The decision spaces induced by deny-overrides can be defined as follows:

$$DS_D^p = DS_D^{p_1} \cup DS_D^{p_2}$$

$$DS_{IN(PD)}^p = \Big((DS_{IN(PD)}^{p_1} \cup DS_{IN(PD)}^{p_2}) \cup \big(DS_{IN(D)}^{p_1} \cap (DS_{IN(P)}^{p_2} \cup DS_P^{p_2})\big) \cup \big(DS_{IN(D)}^{p_2} \cap (DS_{IN(P)}^{p_1} \cup DS_P^{p_1})\big)\Big) \setminus DS_D^p$$

$$DS_{IN(D)}^p = (DS_{IN(D)}^{p_1} \cup DS_{IN(D)}^{p_2}) \setminus (DS_D^p \cup DS_{IN(PD)}^p)$$

$$DS_P^p = (DS_P^{p_1} \cup DS_P^{p_2}) \setminus (DS_D^p \cup DS_{IN(PD)}^p \cup DS_{IN(D)}^p)$$

$$DS_{IN(P)}^p = (DS_{IN(P)}^{p_1} \cup DS_{IN(P)}^{p_2}) \setminus (DS_D^p \cup DS_{IN(PD)}^p \cup DS_{IN(D)}^p \cup DS_P^p)$$

Intuitively, the representation above defines the priorities between decision spaces. The *Deny* space of the parent policy element is the union of the *Deny* space of child policy elements, i.e. the former evaluates to *Deny* if at least one child policy element evaluates to *Deny*. Then, *Indeterminate*$\{PD\}$ has priority over *Indeterminate*$\{D\}$; in turn *Indeterminate*$\{D\}$ has priority over *Permit*, which has priority over *Indeterminate*$\{P\}$. The overall *Indeterminate* space can be obtained as the union of the three *Indeterminate* spaces, i.e. $DS_{IN}^p = DS_{IN(PD)}^p \cup DS_{IN(D)}^p \cup DS_{IN(P)}^p$.

Example 3 *Consider the policy in Example 1 and the applicability space of the rules forming it in Example 2. Decision space of rule* r_i $\langle DS_P^{r_i}, DS_D^{r_i}, DS_{IN}^{r_i} \rangle$ *is*

$r_1 : \langle \emptyset, ac_0 \cap ac_1 \cap ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}, (ac_8 \cup ac_9 \cup ac_{10}) \cap (ac_0 \cap ac_1) \rangle$

$r_2 : \langle \emptyset, ac_0 \cap ac_1 \cap (ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)}, (ac_6 \cup ac_7) \cap (ac_0 \cap ac_1) \rangle$

$r_3 : \langle ac_0 \cap ac_1, \emptyset, \emptyset \rangle$

The decision space of the overall policy $\langle DS_P, DS_D, DS_{IN} \rangle$ *can be obtained by combining the decision space of the rules as shown above (by derivation order):*

$$DS_D^p = ac_0 \cap ac_1 \cap \Big(\big(ac_5 \cap \overline{(ac_8 \cup ac_9 \cup ac_{10})}\big) \cup \big((ac_2 \cup ac_3 \cup ac_4) \cap \overline{(ac_6 \cup ac_7)}\big)\Big)$$

$$DS_{IN}^p = \big(ac_0 \cap ac_1 \cap (ac_8 \cup ac_9 \cup ac_{10} \cup ac_6 \cup ac_7)\big) \setminus DS_D^p$$

$$DS_P^p = (ac_0 \cap ac_1) \setminus (DS_D^p \cup DS_{IN}^p)$$

where notation $\overline{S}$ *is used to denote the complement of set* S.

3.2 Policies as SMT Formulas

The expressions presented in the previous section can be straightforwardly translated to many-sorted first-order formulas over the attributes in *Att* and a theory $\mathcal{T}$ specifying the algebraic structures of the values of *Att* in *Dom*. This allows us to encode the decision space of a policy using SMT formulas.

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

When talking about deciding satisfiability of a policy p in SMT, we refer to $\mathcal{T}$-satisfiability of the formulas $\mathcal{F}_P$, $\mathcal{F}_D$ and $\mathcal{F}_{IN}$. Since decision spaces DS_P, DS_D and DS_{IN} are pair-wise disjoint, their satisfiability is mutually exclusive.

The background theories needed for the analysis of a policy are determined from the policy's applicability constraints. In order to do this, we map classes of common XACML functions to certain background theories that can be used to encode the applicability constraints constructed from them.

Most of the logical functions of XACML (i.e., *or*, *and*, *not*) do not require any specific background theory. 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 0-ary function symbols within an appropriate signature Σ. Other constraints involving equality predicates require the theory of equality with uninterpreted functions. This theory does not impose any constraint on the way the symbols in the signature are interpreted. Thus, the predicates that are not supported by any theory can be left uninterpreted and analyzed using the theory of "uninterpreted functions". The theory of equality with uninterpreted functions can be used to support XACML functions for which a dedicated theory is not available such as XPath-based functions. Constraints defined using arithmetic and numeric comparison functions (e.g., $ac_3, \ldots, ac_5$ in Example 2) require the theory of linear arithmetic. Applicability constraints defined over strings, bag and sets may require dedicated theories. For instance, constraints defined using comparison functions over strings and string conversion functions can be modeled with the theory of strings [26]; constraints defined over bag and set functions (e.g., $ac_6, \ldots, ac_{10}$) can be modeled with the theories of arrays [15] and cardinality constraints on sets [24].

Finally, observe that a background theory can be a combination of different theories as it is the case of Example 2 in which the cost of a transaction depends on its value. This dependence can be represented by a function f which is left uninterpreted since we are not interested in specifying exactly how the cost must be derived from the value of the transaction. By abstracting the actual details of f, the applicability constraint ac_5 can be represented as $(\mathsf{credit} < f(\mathsf{value}) + \mathsf{value})$ and interpreted within a combined background theory of linear arithmetic and uninterpreted functions.

4 XACML Policy Analysis

The previous section describes an encoding of XACML policies as SMT formulas. In this section we use this encoding to represent *policies analysis problems*, i.e. for a collection of policies checking various properties expressed in so called *queries*. We first introduce the query language and then give example query formulas for different policy properties from the literature.

Definition 4 *Let $\langle Att, Dom\rangle$ be the access control scheme 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 a formula of the form*

$$Q = P_i \mid D_i \mid IN_i \mid g(t_1, \ldots, t_k) \mid \neg Q \mid Q_1 \vee Q_2 \mid \ldots \\ \mid (\forall x : \sigma\ Q) \mid (\exists x : \sigma\ Q) \mid \nu x.Q \mid Q\langle a_1 = v_{1_j}, \ldots, a_n = v_{n_k}\rangle$$

where P_i, D_i and IN_i (for $i = 1, \ldots, n$) are new symbols representing the Permit*,* Deny *and* Indeterminate *spaces of policy p_i (they thus represent $\mathcal{F}_P^{p_i}$, $\mathcal{F}_D^{p_i}$ and $\mathcal{F}_{IN}^{p_i}$ respectively, see also query semantics below), 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 also 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 a variable x in Q (i.e., $\nu x.Q \equiv Q[x/y]$ with y a fresh variable), $Q\langle a_1 = v_{1_j}, \ldots, a_n = v_{n_k}\rangle$ represents the instantiation of a policy with a request with $v_{1_j} \in Dom_{a_1}, \ldots, v_{n_k} \in Dom_{a_n}$.*

Note that construct $\nu x.Q$ is used to restrict the scope of the substitution of a variable x to a subformula Q of the query. 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 (see below for examples of such properties). $Q\langle a_1 = v_{1_j}, \ldots, a_n = v_{n_k}\rangle$ is logically equivalent to $Q \wedge v_{1_j} \in a_1 \wedge \ldots \wedge v_{n_k} \in a_n$.

The basic query P_i encodes (inclusion in) the *Permit* space of policy p_i; it is satisfiable if any request is permitted by p_i. Similarly, D_i and IN_i represent the *Deny* and *Indeterminate* spaces of p_i respectively. Constraints such as $\textbf{Alice} \in \mathsf{subject\text{-}id}$, $\forall v \in \mathsf{current\text{-}time}\ v < 18{:}00$ etc., are used to instantiate the subject or the time of the query. The predicates can also capture relations between different policies (see examples below).

Example 4 *Let p_1, p_2 be two policies, and $\langle P_1, D_1, IN_1\rangle$ and $\langle P_2, D_2, IN_2\rangle$ their SMT representation, respectively. Below we present some example queries.*

- $(P_1 \rightarrow P_2)$*: any request permitted by p_1 is also permitted by p_2.*
- $\nu\ \mathsf{subject\text{-}id}.(P_1\langle \mathsf{subject\text{-}id} = \textbf{Alice}\rangle) \wedge \nu\ \mathsf{subject\text{-}id}.(D_1\langle \mathsf{subject\text{-}id} = \textbf{Bob}\rangle)$*: some request is permitted by p_1 for Alice but denied for Bob.*
- $(P_1 \wedge D_2)\langle \mathsf{subject\text{-}id} = \textbf{Alice}\rangle$*: some request of Alice is permitted by p_1 but denied by p_2.*
- $P_1\langle \mathsf{subject\text{-}id} = \textbf{Alice}, \mathsf{resource\text{-}type} = \textbf{transaction}, \mathsf{action\text{-}id} = \textbf{create}\rangle$*: policy p_1 allows Alice to create a transaction.*

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

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

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

In the remainder of this section, we demonstrate that our framework can model various types of policy properties proposed in the literature.

Policy Refinement and Subsumption Organizations often need to update their security policies to comply with new regulations or to adapt changes in their business model. Nonetheless, they might have to ensure that the new policies preserve (refine) the intention of the original policies. Different definitions of policy refinement have been proposed in the literature. Backes et al. [5] propose a notion of policy refinement based on the idea that *"one policy refines another if using the first policy automatically also fulfills the second policy"*. Intuitively, a policy refines another policy if whenever the latter returns *Permit* (or *Deny*) the first policy returns the same decision. This can be formalized in our framework as follows. Let p_1, p_2 be two policies with decision space $\langle P_1, D_1, IN_1\rangle$ and $\langle P_2, D_2, IN_2\rangle$ respectively. Policy p_2 is a refinement of p_1 iff the following formula is $\mathcal{T}$-valid

$$(P_1 \rightarrow P_2) \wedge (D_1 \rightarrow D_2) \tag{1}$$

Hughes and Bultan [13] present a stronger notion of policy refinement called policy subsumption. In addition to constraining *Permit* and *Deny* spaces as in refinement, subsumption also imposes constraints on the *Indeterminate* space. Formally, policy p_1 subsumes policy p_2 iff the following formula is $\mathcal{T}$-valid

$$(P_1 \rightarrow P_2) \wedge (D_1 \rightarrow D_2) \wedge (IN_1 \rightarrow IN_2) \tag{2}$$

Note that our framework is general enough to express other notions of policy refinement, for instance imposing constraints only on the *Permit* space or on the *Deny* space. In the next example, we demonstrate the notion of policy refinement presented in [5] with respect to background theory *linear arithmetic*.

Example 5 *Consider the XACML policy in Example 1. Suppose that the policy is updated by omitting the cost of the transaction in rule r_1:*

$$r_1'[Deny] : \mathsf{value} > \mathsf{credit}$$

We want to check whether the new policy is a refinement of the original policy. It is easy to verify that (1) does not hold if the cost of the transaction is higher than the credit minus the value of the service. Therefore, the new policy is not a refinement of the original policy.

Change-impact Change-impact analysis [10] aims to analyze the impact of changes to policies. Intuitively, change-impact analysis is the counterpart of policy refinement, in which the goal is to extract the differences between two policies. Differently from policy refinement, changes of the *NotApplicable* space should also be considered in change-impact analysis. Let p_1, p_2 be two policies with decision space $\langle P_1, D_1, IN_1\rangle$ and $\langle P_2, D_2, IN_2\rangle$ respectively. We are interested in

finding the access requests for which the decisions returned by p_1 and p_2 are different. This policy analysis problem consists of finding access requests that satisfy the following formula:

$$(P_1 \rightarrow \neg P_2) \vee (D_1 \rightarrow \neg D_2) \vee (IN_1 \rightarrow \neg IN_2) \\ \vee(\neg(P_1 \vee D_1 \vee IN_1) \rightarrow (P_2 \vee D_2 \vee IN_2)) \tag{3}$$

where $\neg(P_1 \vee D_1 \vee IN_1)$ represents the *NotApplicable* space.

Attribute Hiding An attribute hiding attack is a situation in which a user is able to obtain a more favorable authorization decision by hiding some of her attributes [8]. Attribute hiding attack is a threat exploiting the non-monotonicity of access control systems such as XACML. Differently from the previous policy properties that can be expressed solely in terms of *Permit*, *Deny* and *Indeterminate* spaces of the policies, attribute hiding is about changing the request: a request that is previously denied is permitted by hiding some attributes or attribute-value pairs. In particular, we call *partial attribute hiding* attack the situation in which a user hides a single attribute-value pair. Let $req = \langle a_1 = v_{1_i}, \ldots, a_n = v_{n_k} \rangle$ with $v_{1_i} \in Dom_{a_1}, \ldots, v_{n_k} \in Dom_{a_n}$ be a request denied by a policy p (i.e., a solution of D_p), and $a_{j_m} = v_{j_m}$ an attribute-value pair occurring in req ($1 \leq j \leq n$) and $v_{j_m} \in Dom_{a_j}$. A policy is vulnerable to partial attribute hiding attack if the request obtained by suppressing $a_j = v_{j_m}$ from req is permitted by p (i.e., a solution of P_p). The property representing the absence of partial attribute hiding attack can be encoded as follows:

$$\nu a.(D_p\langle a = v \rangle) \rightarrow \neg P_p \tag{4}$$

where we use restriction to ensure that the request is only applied to the left part of the formula. A more generalized version of attribute hiding attack is *general attribute hiding* where a user completely suppresses information about one attribute. The property representing the absence of general attribute hiding attack can be encoded as follows:

$$\nu a.(D_p\langle a = v_1, \ldots, a = v_n \rangle) \rightarrow \neg P_p \tag{5}$$

We use an example policy from [8] to discuss the analysis of attribute hiding.

Example 6 *Consider two competing companies, A and B. To protect confidential information from competitors, company A defines the following policy:*

$$\begin{aligned} p[\mathsf{dov}] &: \textbf{true} \\ r_1[Deny] &: \mathsf{confidential} = true \wedge \mathsf{employer} = B \\ r_2[Permit] &: \textbf{true} \end{aligned}$$

The first rule (r_1) of the policy 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 combining algorithm (*dov*). Now consider the following access requests:*

$$req_1 = \langle \mathsf{employer} = A, \mathsf{confidential} = true \rangle$$
$$req_2 = \langle \mathsf{employer} = A, \mathsf{employer} = B, \mathsf{confidential} = true \rangle$$
$$req_3 = \langle \mathsf{confidential} = true \rangle$$

Rule r_1 is only applicable to request req_2 and thus the request is denied. Rule r_2 is applicable to the remaining requests and thus access is granted for requests req_1 and req_3. However, if the subject can hide some information from the request, for instance, reducing req_2 to req_1 by suppressing element $\mathsf{employer} = B$ *from the request (partial attribute hiding) or to req_3 by suppressing attribute* employer *from the request (general attribute hiding), then she would be allowed to access confidential information leading to a violation of the conflict of interest requirement. Note that we assume that attribute* confidential *is under the control of the system and cannot be hidden by the user.*

Scenario-finding Scenario finding queries [10, 19] aim to find attribute assignments that represent scenarios in which a sought behavior occurs. They are especially useful to obtain request instances of certain decision types (e.g., *permit*) which are otherwise difficult to obtain manually. Examples of scenario finding queries include checking whether a policy ever permits (some) users to perform certain actions or denies certain actions under given circumstances. Scenario finding can also be used to check whether a policy is compliant with well-known security principles. For instance, a XACML policy implementing role-based access control can be checked for the separation of duty principle or a XACML policy implementing Chinese Wall policy can be checked if it correctly implements conflict of interest classes.

Scenario finding does not have a fixed form of encoding as the previous properties since it is formulated by the user according to selected decision space.

Example 7 *In the context of Example 2 a policy author may want to check whether the policy permits any access request before* 18:00 *on* Saturday*. We can encode this query as follows:*

$$P \wedge \mathsf{current\text{-}day} = Saturday \wedge \mathsf{time} < 18{:}00$$

Many types of scenario finding queries can be formulated and analyzed within existing XACML analysis tools. However, most of these tools leave non-Boolean functions (i.e., Σ-terms of form $f(t_1, \ldots, t_n)$) uninterpreted. In contrast, SMT enables to reason on those attributes using a suitable underlying background theory. For instance, an SMT solver can find an assignment for an attribute "age" that satisfies a Linear Arithmetic constraint $\mathsf{age} < 18$.

5 Evaluation

In this section we evaluate our SMT-based policy analysis framework by means of a prototype implementation. In the evaluation we use two sets of experiments, one comparing our SMT-based solution to SAT-based techniques and one showing our prototype can be used on realistic policies. Our experimental testbed consists of a 64-bit (virtual) machine with 16GB of RAM and 3.40GHz quad-core CPU running Ubuntu.

5.1 Prototype Implementation

To support the analysis of XACML policies described in the previous section, we have implemented *X2S* [25], a formal policy analysis tool. *X2S* employs Z3 [18], an SMT-LIB v2 compliant tool that supports efficient reasoning in a wide range of background theories, as the underlying solver. *X2S* accepts both XACML v2 and v3 policies and supports a large fraction of standard XACML functions. It consists of two main components. The first component, the *SMT Translator*, first translates XACML policies provided by the user into SMT formulas using the encoding presented in Section 3. Next the user is prompted to enter a query expressed in the language defined in Section 4 which is also translated and added to SMT specification. The second component, the *Report Generator*, presents the results of the analysis by providing an interface to the SMT solver.

Our prototype can enumerate models as required for certain queries such as *change impact*. We perform this by incrementally adding a new constraint representing the negation of the obtained model to the original formula. However, there may be infinitely many models satisfying a formula with certain expression types. To help alleviate this problem, we try to avoid models that do not "significantly" differ from those already considered with respect attribute assignments. In particular, we do this in the treatment of arithmetic expressions by fixing the assignments of (arithmetic) variables in a model to the first values found. For instance, if the first solution of the arithmetic expression $att_1 < att_2$ assigns 4 and 5 to the attributes respectively, then we fix these assignments by adding new (conjunctive) constraints $att_1 = 4$ and $att_2 = 5$ to the original formula.

5.2 Experiments 1: SAT vs. SMT

Consider a user wanting to validate and possibly update a set of policies collected over time and from different contributors. For example, a building manager wants to verify the policy governing the access to a certain building in which right to enter depend on the current time and date and/or membership of a group; or a bank manager wants to verify the bank policy for transfers which depend on the balance of accounts, size of the transfer, etc. The main advantage of our SMT approach over a SAT based solution is that it allows direct reasoning with non-Boolean values. For example, one can use the background theories for basic sets (i.e., the theory of arrays) and linear arithmetic (LAI). To perform this analysis in SAT one has to encode everything in Boolean values. With some limitations

we can encode LAI constraints in SAT using order encoding [21] where each expression of the form $x \leq c$ is represented by a different Boolean variable. Membership expressions in the set theory can be encoded in SAT using a similar approach where the relation between a variable and a value from its domain is represented with a different Boolean variable for each value.

Ideally, the user's validation tool would be able to give real-time feedback on their edits, or at the very least, respond promptly to a validation query. When analyzing with SAT, users needs to find a suitable trade-off between the precision and the efficiency as well as the scalability of the analysis; for example instead of the time only distinguishing 'morning' from 'afternoon' or hour of the day. Choosing what granularity is suitable for what attribute is a difficult, laborious and error-prone task requiring the user to closely investigate all constraints. Too a low granularity may lead to missing errors in the policies. Yet, the more fine grained the analysis is, the larger the SAT encoding. Our experiment below confirm that increasing the granularity quickly become very costly performance wise. Our SMT-based approach does not needed to restrict the granularity.

To illustrate the effect of granularity on the analysis we distinguish course grained analysis using a 'small' domain (e.g., morning/afternoon for time, and day of the week for date), an analysis with some detail through a 'medium' (M) size domain (e.g., minutes in an hour, days in a month) and a detailed analysis using a 'large' (L) domain (e.g., minute in a day, day of a year). We analyze policies and properties from the examples in Section 4. We check policy refinement (PR), policy subsumption (PS), change-impact (CI) analysis, both partial (P-AH) and global (G-AH) Attribute Hiding, and finally scenario finding (SF). We analyze each with our prototype and three different SAT solvers; zchaff [17], lingeling [7] and Z3 itself to obtain a fair comparison as certain solvers are optimized for certain types of problems. We use size 10 to represent small domains, 100 for medium domains and 500 for large domains (they may need to be much larger but this size already shows the clear advantage of our SMT solution). Note that we aim at a comparison in orders of magnitude rather than an in-depth and comprehensive performance analysis. For small domains all solutions are able to complete the analysis quickly with limited resources; they are fast enough for real-time feedback during editing. For the medium and large domains the results are provided in Table 1. 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 ($\mathcal{F}$). The other columns present the results in terms of number of variables used in the encoding, memory allocation[4] and required computation time for the SAT solvers with M(edium) and L(arge) domain size, and SMT.

Compared to the number of many-sorted first order variables in SMT encoding, the number of Boolean variables in SAT encoding is quite large due to the mapping of non-Boolean domains to Boolean variables. For instance, the SMT encoding of the policy query for verifying policy refinement requires 12 variables. These variables are used to specify the attributes defined in the policy as well

[4] We used a memory profiler for measuring the memory usage.

Table 1: Evaluation Results of Example Properties with SAT vs SMT Encoding

		#Vars			**Memory(MB)**							**Time(s)**						
		SAT		SMT	Z3-SAT		zchaff		lingeling		SMT	Z3-SAT		zchaff		lingeling		SMT
P	**Q**	M	L		M	L	M	L	M	L		M	L	M	L	M	L	
PR	$\neg\mathcal{F}$	591	2191	12	84	459	99	340	23	555	0.3	1.6	99.7	~0	3.3	20.5	>100	~0
PS	$\neg\mathcal{F}$	909	2509	12	303	240	377	1159	82	2054	0.3	3.9	6.1	~0	12.4	65.5	>100	~0
CI	$\mathcal{F}$	1409	3009	12	88	231	650	1087	133	1513	0.5	0.3	9.1	~0	19.1	36.5	45.5	~0
P-AH	$\neg\mathcal{F}$	24	15	3	0.1	0.1	0.1	0.1	~0	~0	0.3	~0	~0	~0	~0	~0	~0	~0
G-AH	$\neg\mathcal{F}$	12	12	4	~0	~0	0.1	0.1	~0	~0	0.3	~0	~0	~0	~0	~0	~0	~0
SF	$\mathcal{F}$	511	1718	12	13	328	92	409	14	356	0.3	~0	1.2	~0	13.4	0.2	7.3	~0

as the Boolean variables representing one-and-only constraints on the arithmetic variables (Example 1). In contrast, 591 Boolean variables are needed to encode the same policy query in SAT when a medium size domain is considered. The memory allocated by the SMT solver needed in analyzing the example policies was always less than 1MB for all properties while SAT solver requires several orders of magnitude more memory. The time necessary to prove (or disprove) that the property holds was negligible (~10ms) for all SMT cases. Analysis with SAT solvers performs far worse with the growth of the domain size as can be noted from the table. For instance, for scenario finding analysis with a large domain, the best performing SAT solver (Z3) took ~1.2s which is several orders of magnitude slower than the analysis with SMT (which took 7ms). The exception is the case of attribute hiding analysis where the SAT solvers offer performance similar to SMT. This is expected since our example policy for attribute hiding does not include complicated predicates and the available predicates can be easily represented in propositional logic. Note that in our experiments for the case of change-impact analysis, we obtained only one model since we prune the uninteresting assignments of arithmetic variables (i.e. value, credit and cost). Finally, we also observe a performance variation between different SAT solvers. We believe this is due to the fact that certain solvers are better tailored to certain types of problems.

In conclusion, even with these relatively simple policies, performance quickly becomes impractical using SAT based solvers while the SMT approach could even be used for real-time feedback while editing a policy. In the next section, we test our approach with some more complex and realistic policies.

5.3 Experiments 2: Real-world Policies

In this second set of experiments, we analyze four realistic policies with our prototype in order to obtain insights about its performance in real-world settings. The policy GradeMan is a simplified version of the access control policy used to regulate access to grades at Brown university and the Continue-a policy is used to manage a conference management system. Both policies are from [10] and consist mainly of string equality predicates. IN4STARS is an in-house policy

Table 2: Evaluation Results for Real-world Policies

Policy	#PSet	#Policy	#Rule	Time(ms)					
				PR	PS	CI	P-AH	G-AH	SF
IN4STARS	3	4	11	24	28	1717	7	7	10
KMarket	1	3	12	36	12	2525	13	12	10
GradeMan	11	5	5	40	30	2424	10	9	17
Continue-a	111	266	298	91	87	2929	33	21	43

defined in the context of a project on intelligence interoperability. It contains various user-defined functions that are used to determine the privileges of users according to their clearance. All these three policies are XACML v2 policies. Our final test policy, KMarket, is a sample policy to manage authorizations in an on-line trading application from [1]. It contains simple arithmetic operations such as less-than and is written in XACML v3.

We performed policy refinement, subsumption and change-impact analysis by modifying the value of a single, randomly chosen attribute in the original policy. The number of models has been limited to 100 during change-impact analysis. For scenario finding, we look for an assignment of attributes (i.e. model) that is permitted by the input policy. Our findings are summarized in Table 2 in which we report the characteristics of policies (e.g., the number of policy elements in the XACML policy) and the time taken by our prototype to answer queries.

Analyzing the policies included in our experiments takes less than 100ms for all properties except Change-impact which makes feedback during policy editing feasible. Change-impact analysis, however, brings the time up to 3s as it requires the enumeration of models in the SMT formula. Another important observation in the experiments is the efficiency of dealing with expressions with non-Boolean attributes; we have not observed a significant performance difference between the analysis of KMarket which contains linear arithmetic expressions and GradeMan which consists of very simple expressions. Finally, the result of Continue-a analysis (a policy with around 300 rules) indicates that the time needed for analysis with SMT of larger policies increases but not necessary as quickly as the policy grows. This result is not surprising since the analysis of a policy with our approach not only depends on the size of the policy but also the type of expressions contained in them.

We believe the experimental results of this and the previous section demonstrate that our approach can be used in practice to analyze realistic policies at a more fine-grained level than the one permitted by the use of SAT solvers with no significant performance penalty.

6 Related Work

When XACML policies grow in number and size, or are updated to address new security requirements, it is difficult to verify their correctness due to XACML's

rich and verbose syntax. To assist policy authors in the analysis of XACML policies, several policy analysis tools have been proposed. One of the most prominent tools for policy analysis is Margrave [10]. Margrave uses multi-terminal binary decision diagrams (MTBDDs) as the underlying representation of XACML policies. The nodes of an MTBDD 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. Margrave 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 [12]. XAnalyzer uses a policy-based segmentation technique to detect and resolve policy anomalies such as redundancy and conflicts. Compared to our approach, BDD-based approaches allow the verification of XACML policies against a limited range of properties. In addition, these approaches encode only a fragment of XACML with simple constraints [13].

An alternative to Margrave, and in general to BDD-based approaches, is presented in [13] where policies and properties are encoded as propositional formulas and analyzed using a SAT solver. However, SAT solvers cannot handle non-Boolean variables; most XACML functions are thus left uninterpreted limiting the capability of the analysis. EXAM [16] 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. For instance, Kolovski et al. [14] 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 in a more expressive manner but also hinders the performance. Ramli et al. [23] and Ahn et al. [2] present a formulation of policy analysis problems similar to ours in answer set programming (ASP). However, these approaches have drawbacks due to intrinsic limitations of ASP. Unlike SMT, ASP does not support quantifiers, and cannot easily express constraints such as Linear Arithmetic. Indeed, in ASP the grounding (i.e., instantiation of variables with values) of Linear Arithmetic constraints either yield very large number of clauses (integers) or is not supported (reals).

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

While the use of SMT for the analysis of XACML policies is new to our knowledge, there are few recent proposals that exploit SMT solvers for the anal-

ysis of policies specified in different access control models. The work in [3] shares with our approach the use of SMT solvers to support the analysis of polycies. The main difference is in the input language: instead of using XACML, Arkoudas et al. [3] adopts a sophisticated logical framework, which can handle XACML policies (such as Continue) indirectly by translating them to expressions of the logical framework to which the available analyses (such as those considered in this paper) can be applied. In contrast, our technique generates proof obligations to be discharged by SMT solvers directly from XACML policies. Another example of SMT techniques supporting the analysis of policies is [4] 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. Although these proposals show the potentiality of SMT for policy analysis, the policy specifications considered in such proposals are rather simple. In this work we make an additional step by showing that SMT is able to deal with real world XACML policies.

7 Conclusions

In this paper, we presented an SMT-based analysis framework for policies specified in XACML. The use of SMT does not only enable wider coverage of XACML compared to existing analysis tools but also presents significant performance gains in terms of allocated memory and computational time. As demonstrated in the paper, several security policy properties found in the literature can be easily encoded and checked within our framework. In our prototype, we use various background theories to encode a large fraction of XACML functions, allowing a fine-grained analysis of XACML policies. SMT function symbols encoding XACML functions for which a specific background theory is not available (e.g., XPath-based and regular-expression-based functions) are left uninterpreted. With the development of new background theories, policy analysis problems using those predicates can be represented and solved efficiently. Our experiments show that our framework enables efficient policy analysis and can be used in practice. As future work, we plan to extend the performance analysis of our prototype against a larger set of real-world policies.

Acknowledgments This work has been partially funded by the EDA project IN4STARS2.0, the EU FP7 project AU2EU, the ARTEMIS project ACCUS, and the Dutch national program COMMIT under the THeCS project.

References

1. Balana: Open source xacml 3.0 implementation (Jan 2013), `http://xacmlinfo.org/category/balana/`
2. Ahn, G.J., Hu, H., Lee, J., Meng, Y.: Representing and reasoning about web access control policies. In: COMPSAC. pp. 137–146 (2010)

3. Arkoudas, K., Chadha, R., Chiang, C.J.: Sophisticated access control via SMT and logical frameworks. ACM TISSEC 16(4), 17 (2014)
4. Armando, A., Ranise, S.: Automated and efficient analysis of role-based access control with attributes. In: DBSec. pp. 25–40 (2012)
5. Backes, M., Karjoth, G., Bagga, W., Schunter, M.: Efficient comparison of enterprise privacy policies. In: SAC. pp. 375–382 (2004)
6. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885. IOS Press (2008)
7. Biere, A.: Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In: POS. p. 88 (2014)
8. Crampton, J., Morisset, C.: PTaCL: A Language for Attribute-Based Access Control in Open Systems. In: POST. pp. 390–409 (2012)
9. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
10. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE. pp. 196–205 (2005)
11. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability Solvers. In: Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 89–134. Elsevier (2008)
12. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and Resolution of Anomalies in Web Access Control Policies. TDSC 10(6), 341–354 (2013)
13. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. STTT 10(6), 503–520 (2008)
14. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing web access control policies. In: WWW. pp. 677–686 (2007)
15. Kröning, D., Weissenbacher, G.: A Proposal for a Theory of Finite Sets, Lists, and Maps for the SMT-Lib Standard. In: Pro. International Workshop on Satisfiability Modulo Theories (2009)
16. Lin, D., Rao, P., Bertino, E., Li, N., Lobo, J.: Exam: a comprehensive environment for the analysis of access control policies. Int. J. Inf. Sec. 9(4), 253–273 (2010)
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535 (2001)
18. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS. pp. 337–340 (2008)
19. Nelson, T.: First-order Models For Configuration Analysis. Ph.D. thesis, Worcester Polytechnic Institute (2013)
20. OASIS XACML Technical Committee: eXtensible Access Control Markup Language (XACML) (2013)
21. Petke, J., Jeavons, P.: The Order Encoding: From Tractable CSP to Tractable SAT. In: SAT. pp. 371–372 (2011)
22. Pratt, V.R.: Two easy theories whose combination is hard. Tech. rep., MIT (1977)
23. Ramli, C.D.P.K., Nielson, H.R., Nielson, F.: XACML 3.0 in Answer Set Programming. In: LOPSTR (2012)
24. Suter, P., Steiger, R., Kuncak, V.: Sets with cardinality constraints in satisfiability modulo theories. In: VMCAI. pp. 403–418 (2011)
25. Turkmen, F., den Hartog, J., Zannone, N.: Analyzing Access Control Policies with SMT. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 1508–1510. ACM (2014)
26. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a Z3-based string solver for web application analysis. In: ESEC/SIGSOFT FSE. pp. 114–124 (2013)