





Efficiently Characterizing the Undefined Requests of a Rule-Based System

Zheng Cheng¹(✉) , Jean-Claude Royer² , and Massimo Tisi²

¹ Inria Rennes - Bretagne Atlantique, LS2N (UMR CNRS 6004), Rennes, France
zheng.cheng@inria.fr

² IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
{jean-claude.royer,massimo.tisi}@imt-atlantique.fr

Abstract. Rule-based systems are used to define complex policies in several contexts, because of the flexibility and modularity they provide. This is especially critical for security systems, which may require to compose evolving policies for privacy, accountability, access control, etc. The inclusion of conflicting rules in complex policies, results in the inability of the system to unambiguously answer to certain requests, with possibly unpredictable effects. The static identification of these undefined requests is particularly challenging for unconstrained rule-based systems, including quantifiers, computations and chaining of rules. In this paper we introduce a static method to precisely characterize the set of all undefined requests for a given unconstrained rule-based system, providing the user with a global view of the rule conflicts. We propose an enumerative approach, made usable in practice by two key performance optimizations: a finer classification of the rules and the resort of the topological sorting. We demonstrate its application on a well-known policy with more than fifty rules.

1 Introduction

Rule-based systems are widely used in very different contexts, ranging from knowledge representation and reasoning to system configuration, from logic programming to databases. Among these contexts, security systems are especially witnessing a significant growth in production of critical, complex and rapidly evolving rule-based policies aiming to offer strong guarantees (of privacy, accountability, etc.) in modern networking environments (including Internet of Things, Software-Defined Networks, etc.). A rule expresses in a concise and natural manner the link between some conditions and a conclusion. This **if then else** semantics is familiar to many software stakeholders, and allows for the definition of modular systems and their flexible evolution.

In this paper we consider a strict logical context, where a rule system is a finite set of logical implications and in conjunction with a request it ensures a reply. Efficient methods for verifying the correctness of such systems in practice is an important research subject [1]. We are particularly interested in one type of error, namely rule conflicts, that cause some requests to be *undefined*, i.e. to have

several incompatible replies. The risk of rule conflicts is very relevant in modern security systems, that often compose independent and evolving policies. We are here interested in a general notion of conflicts entailing the system execution and leading to a runtime bug. We are not focusing on redundancies, misconfigurations or other similar problems which can be considered either as simplifications or holes in the rule system.

There are already several verification methods and algorithms for rule-based policies, in expert systems and databases [2], for Web policies and contracts [3], and in the security domain [4–6]. In this paper we will focus on formal methods and assume a formal policy written in a decidable logic. Moreover we will consider unconstrained systems with complex conclusions and *chaining* of rules (*i.e.* the conclusion of one rule can be used to match other rules and produce new derivations). For these systems we want to provide a *precise characterization* of the set of all undefined requests the user can present, that is in general an infinite set. This characterization constitutes a global view of the conflicts in the system, and a valuable aid in debugging extensive rule sets.

Techniques that have the potential to check for conflicts in unconstrained systems with chaining, divide in two categories. The *testing method* (e.g., [7,8]) computes the set of undefined requests by generating large sets of ground sentences as test requests, and checking the unsatisfiability of each of them in conjunction with the rule system. The testing method suffers from two main drawbacks: (*i*) the global cost of the request generation and evaluation is very high and (*ii*) the test set has finite coverage over the infinite set of undefined requests. The *verification method* (e.g., [9–12]) considers consistency properties (e.g., it is not possible to deny and permit an access at the same time) and tries to prove these properties. It has similar drawbacks since it is generally costly, and does not aim at an exhaustive view of all possible conflicts.

Our contribution is to provide an enumerative method based on symbolic manipulation of the rules and a satisfiability procedure to exhaustively find the precise set of undefined requests in a rule system. The computation complexity is exponential, but we provide two optimization steps to enhance its practical applicability: an iterative method with rule classification and a sorting algorithm. Finally, we evaluate our approach on a well-known case study in XACML, translated into FOL. The experimentation shows that our method is suitable for the verification of rule systems of this size (*i.e.*, 47 rules), where in less than one hundred seconds it produces and analyzes less than one thousand new rules (summarizing the analysis of 2^{47} rule combinations).

The content of this paper is structured as follows. Section 2 describes related work in the area of rule-based systems and checking for conflicting rules. Section 3 presents a motivating example. Section 4 provides the necessary background and definitions to understand our approach. Section 5 illustrates our enumerative method and optimizations. In Sect. 6 we evaluate our method, based on a well-known case study. Lastly, In Sect. 7 we conclude and sketch future work.

2 Related Work

An extensive literature studies the management of rule-based systems. The survey [1] shows that verification and validation of rule-based systems in practice is mainly based on testing or code review which are of course not sufficient to prove that a system is free from bugs. Validation and verification techniques for various kinds of rule-based systems have also been discussed in [2] for expert systems and database management or [3] for Web policies and contracts. In the domain of security policies, the problem of conflicts has been intensively studied. In surveys on security [4–6], conflict detection is a central problem but it is typically treated together with other tasks like finding bugs, redundancies, misconfigurations, etc. We propose a specific solution for static conflict detection, that we believe to be one of most critical issues in modern rule-based systems.

There are already some efficient algorithms to statically detect conflicts in access control policies [13–18]. The method is generally to look for conflict in all combinations of a number of rules (often only two rules). However, most of these methods support only policies with discrete conclusions (like permit and deny) and even only handle discrete attributes as conditions (ABAC policies). Our approach is more general because, except decidability of satisfiability, we do not assume constraints on the rule system, as we could have conditions, functions, any kind of conclusions, possibly with quantifiers and free variables. Even with unbounded rule systems we claim to generate all the undefined requests of the system.

In case of complex systems with predicates in conclusions and allowing the chaining of rules, the existing solutions are not numerous. There are approaches based on testing, used in different contexts, e.g. [7, 8, 16, 19]. A few studies rely on checking satisfiability as in [20], but it is too weak since satisfiability is always a system requirement. Other formal methods and verification, for instance [9–12], use manual proof or derivation tools and are able to prove expected properties of the system (e.g., an access is never both permitted and denied). In practice these properties concern a finite set of authorizations, but in case of unrestricted obligations and chaining of rules the enumeration of the property of interest can be an issue. We do not expect to compete with these approaches on the side of efficiency in detecting a single conflict. Differently from these works, our method automatically computes all the conflicting requests of the system, even if it is not a finite set.

In the domain of constraint solving, we think that techniques for extracting *minimal unsat core* and *maximal sat core* are closely complementary to our work (e.g. [21, 22]). We concretely show in this work how to encode the exhaustive conflict detection problem of security policies as a satisfiability problem. To make sure that the problem can be solved within reasonable time, we introduce performance optimizations as symbolic manipulations, before providing the problems to the constraint solver. We do not directly use *minimal unsat core* nor *maximal sat core* techniques to solve our problem. However, our manipulations can be also adapted for enhancing alternative solutions for conflict detection that rely on sat/unsat core (e.g. [23]).

3 Motivating Example

To exemplify our approach, we refer to a very simple example of rule-based policy for a medical center, inspired from [24]. The example is illustrated in Listing 1.1 using the syntax of the Z3 solver, where we represent implication as an infix operator. This example is a first-order example with predicates and free variables but our approach, detailed in the next section, is applicable to any decidable logic extending propositional logic. Here h , p are universally quantified variables (respectively for hospital personnel and patients) and other terms are predicates or boolean operators. The system decides about read and write access rights to the information of a patient (`pread`, `pwrite`). Hospital personnel comprises the `doctor`, `nurse` and `chief` roles. Personnel can be assigned to the ward of a patient (`sameward`). Despite representing a simple policy, user-defined rules in Listing 1.1 contain conflicts, missing information and redundancies. Briefly, hospital employees with multiple roles can not be assigned to the ward of any patient (rule 1), doctors have read and write access to any patient data (rule 2), nurses can not get information on patients from other wards (rule 3), a doctor can access data for the patients of his ward (rule 4), a chief has read access to any patient data (rule 5).

Listing 1.1. Input for the hospital example

```

1 And(doctor(h), nurse(h)) => Not(sameward(h, p))
2 doctor(h) => And(pread(h, p), pwrite(h, p))
3 And(nurse(h), Not(sameward(h, p))) => Not(pread(h, p))
4 And(doctor(h), sameward(h, p)) => pread(h, p)
5 chief(h) => pread(h, p)

```

We provide an automatic prototype that, given a system like the one in Listing 1.1, identifies all *undefined requests*, e.g. requests having multiple ambiguous answers because of rule conflicts. When provided with Listing 1.1, our prototype produces the output in Listing 1.2.

Listing 1.2. Prototype output for the example

```

[0, 1, 0, -1, -1]
  And(doctor(h), Not(nurse(h))) => And(pread(h, p), pwrite(h, p))
[0, 0, 1, 0, 0]
  And(Not(doctor(h)), nurse(h), Not(sameward(h, p)), Not(chief(h)))
    => Not(pread(h, p))
[0, 0, 0, 0, 1]
  And(Not(doctor(h)), Or(Not(nurse(h)), sameward(h, p)), chief(h))
    => pread(h, p)

# ----- Unsafe -----
[1, 1, 1]
  And(doctor(h), nurse(h), Not(sameward(h, p))) => False
[1, 1, 0]
  And(doctor(h), nurse(h), sameward(h, p)) => False
[0, 0, 1, 0, 1]
  And(Not(doctor(h)), nurse(h), Not(sameward(h, p)), chief(h)) => False

```

The rule system in Listing 1.2 is logically equivalent to the original rule system and composed of pair-wise exclusive rules¹. Furthermore, each rule is prefixed by a sequence of digits indicating the combination of the original rules which produces this rule. For instance, the first rule in Listing 1.2 comes from the combination of the original rule 2 and negation of rule 1 and 3; -1 is a don't care digit for rule 4 and 5. The rules tagged as unsafe denote indeed set of undefined requests, for instance `And(doctor(h), nurse(h), Not(sameward(h, p)))` is undefined which results from the combination ([1, 1, 1]) of the three first rules. That means that its conjunction with the rule system is unsatisfiable. The other rules denote defined and undefined requests. On one side, the defined requests should intersect the conjunction of the rule condition and the rule conclusion. On the other side, undefined requests are included, by implication, in the conjunction of the rule condition and the negation of the rule conclusion. Related to the first not unsafe rule, `And(doctor(h), Not(nurse(h)))` is a defined request and `And(doctor(h), Not(nurse(h)), Not(pread(h, p)))` is an undefined one.

With such information the user can exactly know what are all the conflicting problems, and then what are the requests that this rule system can handle. Furthermore, he can localize the problems in the rule system, by knowing what is the rule combination leading to undefined requests. For instance, the first unsafe rule comes from the combinations of rules 1, 2 and 3. It states that any request about a personnel member that is at the same time a doctor and a nurse will lead to a rule conflict, even if the person is not assigned to the ward of the patient. The conflict involves the first three rules, since chaining rule 1 with respectively rule 2 and 3 leads to contradictory answers, where the data from the patient can and can't be read at the same time (`pread(h, p)` and `Not(pread(h, p))`).

The next section will introduce the required concepts and Sect. 5 will present the principles of our algorithm.

4 Background and Definitions

In this section, we define a terminology that will be consistently used in the rest of the paper. Our focus are systems that process requests and produce replies. One important problem is due to requests leading to evaluation failures and often called conflict in the literature. Here we formalize the notion by defining the term *undefined request*. We focus on *rule systems*: a conjunction of rules, a rule is $A \Rightarrow B$ with A and B in some logical language. Considering that satisfiability decision has made important progress, we expect to build a new management method of rule-based systems by reusing efficient tool support (e.g., Z3, SMT, SPASS, TSPASS).

Let R be a satisfiable policy system with its *set of expected requests* REQ . REQ is a finite set of satisfiable logical formulae that we are interested in answering. It represents the set of expected inputs to the policy system. Requests and

¹ To improve readability, we simplify the actual output from our prototype in Listing 1.2. The complete output and our prototype can be found in <https://github.com/atlanmod/ACP>.

replies are satisfiable logical expressions, they could be ground (without variables) or containing free variables (implicitly universally quantified) or quantified variables. Indeed all the expressions (requests, replies, conditions, conclusions, ...) are assumed to be written in a given logical language which may allow variables, quantifiers, modal operators, and so on, providing it has a satisfiability procedure.

Definition 1 (Rule-based system). *A rule-based system (R) consists of a finite conjunction of rules noted $R = r_{1 \leq i \leq n}$, where each rule r_i takes the format of $D_i \Rightarrow C_i$, where D_i stands for the premise/condition of the rule, and C_i stands for conclusion of the rule. In addition we assume that R is satisfiable and does not contain tautological rules.*

We should note that a rule-based system is interesting if it is satisfiable and not valid. This means that it can derive non trivial facts from the request. Thus we assume that R is satisfiable and it does not contain tautological rules, that is $D_i \wedge \neg C_i$ is satisfiable for all i . A request will be a satisfiable logical expression submitted to a rule-based system and leading to a reply, another logical expression. Note that a request should trigger at least one rule, otherwise its conjunction with the system is satisfiable but does not infer a reply. Thus we make the natural hypothesis that our requests are satisfiable and imply $\bigvee_{1 \leq i \leq n} D_i$. We will say that a set of rules is *exclusive* if their conditions are pair-wise disjoint.

Definition 2 (Request evaluation in rule-based system). *Let req be a logical expression, evaluating it against a given rep , called the reply, means that $req \wedge R \Rightarrow rep$ is valid or equivalently $req \wedge R \wedge \neg rep$ is unsatisfiable.*

We are interested in the evaluation problem raised due to undefined requests.

Definition 3 (Undefined requests). *A req request is said undefined if and only if req and R are both separately satisfiable but $req \wedge R$ is unsatisfiable.*

An undefined request in fact leads to a problem since its evaluation leads to multiple incompatible replies. This definition is stricter than satisfiability and covers the usual notion of conflicts we found for instance in security policies [6, 25]. This has two simple consequences: (i) any useful system has undefined requests and (ii) these undefined requests are included in $\bigvee_{1 \leq i \leq n} D_i$. Furthermore any rule ($D_i \Rightarrow C_i$) which is satisfiable and not valid has *1-undefined* requests, that is requests invalidating this rule alone (in other words requests which imply $D_i \wedge \neg C_i$). In this work, we aim to study a complete and efficient algorithm to ensure the safety of a given rule-based system (Definition 4).

Definition 4 (Safety of rule-based system). *A rule-based system with its set of expected requests is safe if and only if it does not contain any undefined requests, that is for all satisfiable request $req \in REQ$ implies $req \wedge R$ is satisfiable.*

As we can see this property assumes that the input system is satisfiable, and it is different from many approaches looking for logical inconsistencies in a system, for instance [9, 10, 20].

5 Characterizing Undefined Requests

The management of policies and requests requires to consider several activities: looking for the existence of one undefined request, checking a request, finding all the undefined requests, localizing the conflicting rules, resolving the undefined requests and evaluating a request. We here focus on finding all the undefined requests since this global knowledge is necessary to understand the failures in the system and to globally fix them. We do not study the fixing process but we will give a few related hints later. As we will see, one additional benefit of our method is to get localization of conflicting rules for free.

This section shows a decision procedure for the safety property (if satisfiability is decidable), its theoretical complexity is EXPSPACE. Our approach is based on transformations of the original rule-based system. It is important to preserve the conditions and conclusions of rules as they represent the expected requests and replies. Hence we will build new rules by mixing the conditions on the left hand side and conclusions on the right hand side.

5.1 The Enumerative Method with Exclusive Rules

Our enumerative method to compute undefined requests is based on rewriting the original rule system into its equivalent form in terms of exclusive rules.

A minor point of the method is to allow requests with variables. By considering requests with variables rather than ground requests, we aim to cover the whole (sometimes infinite) set of requests. For example, in Listing 1.1, a ground request like `And(doctor(Jack), nurse(Jack), Not(sameward(Jack, May)))` with constants `Jack`, and `May` is undefined. However, there are many other different ground requests that expose this problem. To capture the essence of these problems, we use the requests `And(doctor(h), nurse(h), Not(sameward(h, p)))`, which is a short hand for $(\forall h, p \cdot (doctor(h) \wedge nurse(h) \wedge \neg sameward(h, p)))$.

Furthermore, an undefined request is *unsafe* in the sense that it contains (i.e., it is deducible by logical implication from) only undefined requests. However, defined requests are not *safe* in general, in the sense they are not maximally defined.

Definition 5 (Safe request). *Let R be a rule system, a satisfiable request req is safe if and only if any satisfiable request which implies req is defined.*

Second, Lemma 1 establishes that we can rewrite the original rule system into an equivalent form, namely exclusive rules. The lemma can be proved by recurrence on the size of rule system n .

Lemma 1 (Exclusive rules). *Let R be a rule system, we have the equivalence*

$$\bigwedge_{1 \leq i \leq n} (D_i \Rightarrow C_i) \Leftrightarrow \bigwedge_I ((\bigwedge_{i \in I} D_i \wedge \bigwedge_{j \notin I} \neg D_j) \Rightarrow \bigwedge_{i \in I} C_i)$$

where I is any non-empty subset of $\{1, \dots, n\}$.

The rewriting of a rule system with n rules generates $2^n - 1$ exclusive rules. The rule system is composed only of pair-wise disjoint rules, and preserves the original set of defined and undefined requests.

For example, the rewriting applied to the rule system shown in Listing 1.1, results in a total of 31 exclusive rules (partially shown in Listing 1.3). As shown in Listing 1.3, first, we abbreviate each rule by its appearance order in Listing 1.1, e.g. $R1$ is $\text{And}(\text{doctor}(h), \text{nurse}(h)) \Rightarrow \text{Not}(\text{sameward}(h, p))$. Second, we use a D function to get the condition part of a rule, and a C function to get the conclusion part of a rule, e.g. $D(R1)$ is $\text{And}(\text{doctor}(h), \text{nurse}(h))$ and $C(R1)$ is $\text{Not}(\text{sameward}(h, p))$. For instance, in Listing 1.4 we show the non-abbreviated form of the first exclusive rule, corresponding to the first line of Listing 1.3. The condition is obtained by conjunction of the first original rule $R1$'s condition, with the negation of the conditions of the other rules ($R2$ – $R5$). The conclusion is simply the conclusion of $R1$.

Listing 1.3. Part of abbreviated exclusive rules generated by our enumerative method for the example shown in Listing 1.1

```

1 And( D(R1), Not(D(R2)), Not(D(R3)), Not(D(R4)), Not(D(R5)) ) => C(R1)
2 And( D(R1), D(R2), Not(D(R3)), Not(D(R4)), Not(D(R5)) ) => And( C(R1), C(R2) )
3 And( D(R1), D(R2), D(R3), Not(D(R4)), Not(D(R5)) ) => And(C(R1), C(R2), C(R3))
4 ... Another 28 rules

```

Listing 1.4. The first exclusive rule of Listing 1.3 in non-abbreviated form

```

1 [1, 0, 0, 0, 0]
2 And(doctor(h), nurse(h)), Not(doctor(h)),
3 Not(And(nurse(h), Not(sameward(h, p))))),
4 Not(And(doctor(h), sameward(h, p))), Not(chief(h))
5 => Not(sameward(h, p))

```

As we previously saw, a rule (if not valid) has always 1-undefined requests and the transformation above builds a system whose undefined requests are disjoint unions of 1-undefined requests.

Lemma 2 (Undefined requests of exclusive rules). *req is an undefined request of an exclusive rule system R if and only if it is a disjoint sum, $req = \bigoplus_{1 \leq j \leq m} req_j$, where \bigoplus is the accumulative exclusive-or operator, and each req_j is satisfiable and 1-undefined for a given rule j .*

Lemma 2 simply results from the exclusive rules and the partition of req into disjoint parts related to the conditions D_i . Thus the set of safe requests is defined as the disjoint union of all the safe requests associated to each rule. For each rule the conjunction of the condition and the conclusion defines a safe request. The maximal safe request is $safe(R) = \bigvee_I \wedge_{i \in I} D_i \wedge_{j \notin I} \neg D_j \wedge_{i \in I} C_i$. A request is safe if included in $safe(R)$, it is defined if it intersects $safe(R)$, and else it is an undefined request. From that the safety property can be checked by the satisfiability of elements in REQ against $safe(R)$.

Discussion. While existing testing methods for rule systems are generally able to show the existence of ground undefined requests, they can't prove that a given (unbounded) rule system is safe. The enumerative way has three main benefits:

- it works with the same complexity in case of finite or infinite set of ground requests while the testing approach is not suitable with infinite sets,
- it does not depend on the set of requests thus it outperforms the testing approach in many non trivial examples,
- it may produce undefined requests with variables, that represent an abstraction of the system problems, with localization for free.

Compared to the verification approaches, our enumeration does not require a large set of interesting properties to prove. Moreover, if the property is not satisfied, in the best cases previous work on verification generates a counter-example. In general, it is not possible to produce a characterization of all the counter-examples as we did here.

On the performance side, the enumerative method implies the computation of all the rule combinations, which has an expensive cost. Furthermore, we check each rule for validity and valid rules are discarded as they do not contribute to the set of undefined requests. The enumerative method can have worse performance than the testing methods in case of propositional rule systems. However, we expect our method to perform better than exhaustive testing of rule systems with variables. For instance, with systems expressed in FOL, the exhaustive testing requires a maximal number of 2^{P*D^V} test cases, where P is the number of predicate occurrences in R , D the size of the domain of arguments and V the number of arguments. The set of test cases is growing quickly, while the number of exclusive rules is only dependent on the number of rules in the input system.

Work on *minimal unsat core* and *maximal sat core* are closely related to our enumerative method [21,22]. For example, after our partition in order to identify all the undefined requests of a rule-based system, we could send each exclusive rule to a solver to extract all the minimal unsat cores. However, for the sake of performance, we only check exclusive rules for satisfiability, since theoretically, the problem of extracting minimal unsat core is harder than checking satisfiability.

5.2 The Iterative Method

We present a first improvement of the enumerative method, called the *iterative method*. It is based on adding iteratively a new rule at each step and eliminating some rules as soon as possible. The principle is based on the following property. Assuming that R' is a list of exclusive rules computed with the enumerative method and r_j a new rule. Note that each rule in R' can be uniquely described by its *binary characteristic* which is a binary integer of length n (where n is the rule size of the initial rule system). Let b a binary integer we can construct a rule of R' in the following manner: the condition is the conjunction of the conditions D_i (respectively $\neg D_i$) if the i^{th} digit of b is 1 (respectively 0). The conclusion is the conjunction of the conclusions of R for which digit is 1 in b . Thus speaking of an exclusive rule or its binary characteristic is equivalent. We will say that a rule i of the input system is active in an exclusive rule if the binary digit at position i is 1.

Proposition 1 (Iterative principle). *Let R' built from the enumerative method, then the enumerative result for $R' \wedge r_j$ is obtained from the binary characteristics of rules in R' and for each b we will get two new binary characteristics $2 * b$ and $2 * b + 1$ plus one single last characteristic equal to 1.*

This results from a simple recursive analysis of the binary characteristics of R' compared to that of $R' \wedge r_j$. That means that if R' has m rules we will get $2 * m + 1$ rule in $R' \wedge r_j$.

Furthermore, the main loop of the iterative method is based on the above principle and two optimizations.

Definition 6 (Obvious rule). *A rule $D \Rightarrow C$ is an obvious rule iff D is unsatisfiable.*

The first optimization is to discard obvious rules during the iterative method (Definition 6). Obvious rules are one specific kind of tautologies. Adding them during the iterative method offers no value, since they will just generate two new obvious exclusive rules and make no difference to the last generated exclusive rule that in negation form. Notice that we keep exclusive rules that are tautologies but not “obvious” during the iterative method. The reason is that they have an impact on the iteration, e.g. affect the last generated rule. An additional optimization is possible here but it is postponed to future work.

Definition 7 (Unsafe rule). *A satisfiable rule $D \Rightarrow C$ is an unsafe rule iff it is equivalent to $D \Rightarrow false$.*

The second improvement of the iterative method is to separate unsafe rules. An unsafe rule implies no defined request matching its condition. Checking it efficiently depends from the logical language used and should take care of quantifiers in case of free variables. If a rule is not unsafe then $(D \wedge C)$ is a safe request and the rule has only 1-undefined requests in $(D \wedge \neg C)$. A consequence of this classification is that now the maximal safe request ($safe(R)$) is computed only from rules that are not unsafe.

The above definition paves the way to a further optimization, which is to check for unsafe rules and to store them separately from the others. Hence we stop processing unsafe rules and present them in the final result (possibly with a shorter binary characteristic w.r.t. not unsafe rules, e.g. see the first two unsafe rules of Listing 1.2). With the iterative method and the classification (obvious, unsafe and not unsafe) we expect to decrease the number of generated rules which is a critical factor of the enumerative approach.

For example, by applying our iterative method on the rule system shown in Listing 1.1, the first rule $R1$ is checked for its obviousness and unsafeness, and the checks succeed. Then, $R1$ becomes the first exclusive rule and the iteration starts. We show in Listing 1.5 the result of first iteration by adding $R2$ to the first exclusive rule. Since $R2$ also passes obviousness and unsafeness checks, three exclusive rules are built along with iterative method. Their binary characteristics are $[1, 1]$, $[1, 0]$ and $[0, 1]$ respectively.

Listing 1.5. Exclusive rules generated by our iterative method for the example shown in Listing 1.1 at the 1st iteration

1	[1, 1]	And(D(R1), D(R2)) => And(C(R1), C(R2))
2	[1, 0]	And(D(R1), Not(D(R2))) => C(R1)
3	[0, 1]	And(Not(D(R1)), D(R2)) => C(R2)

At the final iteration, we get a total of 6 exclusive rules (3 unsafe, 3 not unsafe, plus 4 eliminated as obvious), which results in a total of 16 unsat checks. Clearly, the iterative approach performs better than the enumerative approach (that in this example would generate 31 rules, by 31 unsat checks), and we anticipate that the effectiveness of the iterative approach would be more visible on larger examples. Note that the iterative method is a quite general solution which requires a logic extending propositional logic with a decision procedure for satisfiability.

5.3 The Sorting Method

The analysis of relations between rule conclusions can lead to significant performance improvements. We argue that in many practical examples, rule conclusions are built on a set of finite predicates, and several rule systems have some pairs of rules whose conclusions are related by inclusion. If we add in the iterative process a new rule $D_j \Rightarrow C_j$ and if we know that it exists $i < j$ such that $C_i \Rightarrow C_j$ is valid then we can simplify the process using the following principle. Let $cond \wedge D_i \Rightarrow conc \wedge C_i$ be the exclusive rule where rule i is active, in the step before the addition of rule j . By adding rule j , the iterative process would generate two rules $cond \wedge D_i \wedge D_j \Rightarrow conc \wedge C_i \wedge C_j$ and $cond \wedge D_i \wedge \neg D_j \Rightarrow conc \wedge C_i$. Given the inclusion relation between C_i and C_j , instead of these two rules, the two rules simplify in $cond \wedge D_i \Rightarrow conc \wedge C_i$, that was already present before adding rule j . This rule will still be exclusive w.r.t. all the other rules generated during the addition of rule j . In the binary characteristic of this exclusive rule, we insert a -1 at position j , to indicate that the condition of rule j does not matter in this combination. This optimization eliminates two satisfiability checks, but most of all it decreases the number of generated exclusive rules.

With this optimization the size of the result depends on the order of rules. Thus, to take the maximal benefit from these relations, we sort the rules from minimal conclusions to maximal conclusions (w.r.t. the implication relation). We achieve this reordering by a simple adaptation of topological sorting, with a complexity in $O(n^2)$ (where n being number of rules). The result of our topological sorting is that each rule is preceded by all the rules with a smaller conclusion than its proper conclusion. For example, the sorting result for our example shown in Listing 1.1 is R1, R3, R2, R4, R5.

5.4 The Algorithm

To sum up, in Algorithm 1 we sketch our process to efficiently compute exclusive rules and classify them as unsafe and not unsafe. Notations used in the algorithm require some explanation.

Algorithm 1. Our algorithm to efficiently compute not unsafe and unsafe requests

```

1: NotUnsafes, Unsafes, Negs, Zeros  $\leftarrow$  dict(), dict(), True, []
2: procedure MAIN( $R$ )
3:    $R \leftarrow$  sort( $R$ )
4:   BUILD_EXRULE(D(hd( $R$ )), C(hd( $R$ )), [1])
5:   Negs  $\leftarrow$  Negs  $\wedge$   $\neg$ D(hd( $R$ ))
6:   Zeros  $\leftarrow$  Zeros ++ [ 0 ]
7:   for each rule  $r_i \in$  tl( $R$ ) do
8:     ITERATIVE_TABLECTR( $r_i$ )
9:   end for
10: end procedure
11:
12: procedure ITERATIVE_TABLECTR( $r$ )
13:   prevNotUnsafes  $\leftarrow$  NotUnsafes
14:   NotUnsafes  $\leftarrow$  dict()
15:   for each  $b_i \in$  prevNotUnsafes.keys() do
16:      $exrule_i \leftarrow$  prevNotUnsafes[ $b_i$ ]
17:     if C( $exrule_i$ ) implies C( $r$ ) then
18:       NotUnsafes  $\leftarrow$  NotUnsafes.update({  $b_i$  ++ [-1] :  $exrule_i$  })
19:     else
20:       BUILD_EXRULE(D( $exrule_i$ )  $\wedge$  D( $r$ ), C( $exrule_i$ )  $\wedge$  C( $r$ ),  $b_i$  ++ [1])
21:       BUILD_EXRULE(D( $exrule_i$ )  $\wedge$   $\neg$ D( $r$ ), C( $exrule_i$ ),  $b_i$  ++ [0])
22:     end if
23:     BUILD_EXRULE(Negs  $\wedge$  D( $r$ ), C( $r$ ), Zeros ++ [1])
24:     Negs  $\leftarrow$  Negs  $\wedge$   $\neg$ D( $r$ )
25:     Zeros  $\leftarrow$  Zeros ++ [ 0 ]
26:   end for
27: end procedure
28:
29: procedure BUILD_EXRULE( $d$ ,  $c$ ,  $b$ )
30:    $rule \leftarrow$  Rule( $d$ ,  $c$ )
31:   if not isObvious( $d$ ) then
32:     if isUnsafe( $d$ ,  $c$ ) then
33:       Unsafes  $\leftarrow$  Unsafes.update({  $b$  :  $rule$  })
34:     else
35:       NotUnsafes  $\leftarrow$  NotUnsafes.update({  $b$  :  $rule$  })
36:     end if
37:   end if
38: end procedure

```

First, we use D and C to get the condition and conclusion of a rule respectively. Second, we inherit traditional list operators hd , tl and $++$ for head, tail and concatenation of lists. Third, we use *dict* to initialize a dictionary, and use *update* to add a pair to a dictionary, and overload $a[b]$ for element access in a dictionary (i.e. accessing element whose key is b in a dictionary a). Finally, three predicates specific to our algorithm are *sort*, *isObvious* and *isUnsafe*. They perform a topological sort of a list of rules w.r.t. implication between conclusions (Sect. 5.3), check whether the rule is an obvious tautology (Sect. 5.2), and check whether the rule is unsafe (Sect. 5.2), respectively.

The input of our algorithm is a list of rules, the output are classified exclusive rules. To compute from input to output, we use 4 global variables in our computation (line 1). *NotUnsafes* and *Unsafes* are dictionaries, which are used to keep track of the rules detected so far, that are unsafe and not unsafe. Notice that the key of these two dictionaries is the binary characteristic of each exclusive rule, and the value is the exclusive rule itself. *Negs* is a conjunction of formula that represents negated conditions for rules that previously iterated on. *Zeros* is synchronized with *Negs* to record binary characteristics. These two global variables are used when building exclusive rules in negative form on line 23.

Our iterative construction is performed to categorize rules that are unsafe or not unsafe (lines 2–27). It internally uses our *sorting method* to optimize its efficiency (lines 17–18), and uses *BuildExRule* to interact with the solver to check whether a newly constructed exclusive rule is unsafe (lines 29–38). The whole process offers no surprise w.r.t. what we described in Sects. 5.2 and 5.3.

Once the two lists of rules *NotUnsafes* and *Unsafes* are computed by Algorithm 1, the set of undefined requests is precisely characterized as:

- each unsafe exclusive rule denotes a set of undefined requests, i.e. all the requests included by implication in the rule condition,
- each exclusive rule that is not unsafe denotes a set of undefined requests, i.e. all the requests included by implication in the conjunction of the rule condition and the negation of the rule conclusion,
- no other undefined requests exist for the original system.

6 Evaluation and Discussion

We implement the three previous methods in Python 3 and we use the `z3py` interface to interact with the Z3 solver [26]. Our input rules are defined using a class of rules reusing the logical expression defined by Z3. The code is available on our on-line repository² with a set of examples. We prove, using the solver, that the original rule system is equivalent to the new generated one.

Our evaluation objective is to experiment with middle size examples to observe if the concrete performances are according to our expectations and suitable for a practical use. In our approach, we rely on the Z3 solver, but the solver

² Efficiently characterizing the undefined requests of a rule-based system (on-line). <https://github.com/atlanmod/ACP.git>.

can be changed as long as it provides a decision procedure for satisfiability suitable for the input rule system. Our method is not limited to pure predicates with free variables, however its success and efficiency depend on the ability of the solver to check such construction.

6.1 The CONTINUE Example

We describe here the case of the CONTINUE A policy already used in [20, 27] and dedicated to conference management. This policy³ is specified in 25 XACML files containing 44 rules. Our objective was not to exactly encode this policy but rather to validate that our optimizations are effective in case of a non trivial example. We deviate from the original CONTINUE example in several ways. First, we consider a pure logical translation and this introduces a difference as already discussed in [10]. We do not take into account the combining algorithms since our objective is to observe the undefined requests occurring in the business rules, not to provide an ad-hoc automatic resolution. We also handle free variables and predicates (unary and binary), while this is not the case in the XACML language.

Our initial rule system has 47 rules, with two types, two free variables, 6 binary predicates and 29 unary ones. We start with this example without additional relations. Figures 1, 2, and 3 depict the results we get with a MacBookPro under El Capitan, 2.5 GHz Intel Core i7 and 16 Go 1600 MHz DDR3 RAM. The resulting times were computed from an average of 10 runs.

In the figures we report the number of rules in the system and a few curves: *correct* is the number of non tautology in the system, *time* the time in second, *exclusive*, *not unsafe*, *unsafe* the number of exclusive, not unsafe and unsafe rules. We process the example by taking the first n rules from an arbitrary ordering. The *correct* curve shows that there is no tautology in this system.

The enumerative experiment (Fig. 1) shows clearly an exponential growth in the number of generated rules and in the processing time. While the performances of the iterative method (Fig. 2) are much higher, we still cannot process the full example in a reasonable time. The sorting method (Fig. 3) provides more interesting results in such a case. CONTINUE has many inclusion relations among conclusions (653 relations), which explains the good performances by topological sorting. While this is not a universal property, in our experience, this is often the case in security systems. This is obvious when conclusions are only permit and deny as in some simple access control policies. More generally, in security we expect to control the possible outcomes of the rules, thus defining a limited set of replies. Each rule can then combined these outcomes and thus revealing relations between conclusions. We easily observe this on several of our examples, but a statistical analysis should be perform to validate this assumption.

We should also note that some rules have a great impact on the behavior of the iterative and sorting methods. This is the case with predicate exclusivity

³ <http://cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/>.

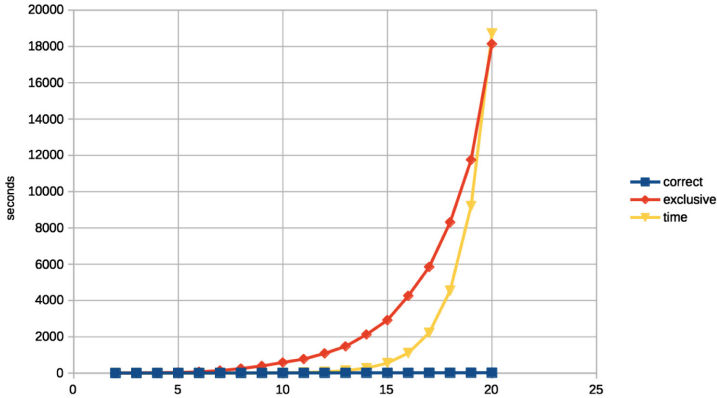


Fig. 1. Enumerative experiments

which is often implicit. Indeed XACML, due to the use of the combining algorithms, does not make hypothesis about the disjunction of roles or permissions of different actions. To observe the stability of our performances we experiment adding a few new rules about roles in the system. For instance, it makes sense to consider that the PC chair is also a PC member and a subreviewer is not a PC member. There are also some relations related to resources, for example there are several different kinds of information about papers. These resources appears only in conditions and alone but never in a conjunction, thus we may consider these resources as disjoint. Adding these rules increases the size of the original system to 57 rules. As shown in the curves in Fig. 4 these disjunctions decrease the number of rules that are not unsafe. With this setting we generate 776 rules (535 unsafe and 241 not unsafe) in nearly 100s, including the verification of the equivalence with the original system which takes 10 s.

Listing 1.6. An example of conflicting rules

```
// An unsafe case
UNSAFE [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]
// The four active 10-13th rules (other negative rules are omitted)
pcchair(X) => pcmember(X)
Or(admin(X), pcchair(X), pcmember(X), subreviewer(X)) => subject(X)
And(PaperAssignments(R), subject(X), isConflicted(X))
=> And(Not(Pread(X, R)), Not(Pwrite(X, R)), Not(Pcreate(X, R)))
And(PaperReviewContent(R), pcmember(X), isEQuserID(X))
=> And(Pcreate(X, R), Pwrite(X, R), Pdelete(X, R))
```

As an example of output in this scenario, Listing 1.6 shows the binary characteristic of one of the first unsafe rules detected (the 10th). As shown by the binary characteristic, this rule denotes a set of undefined requests coming from the composition of the four active rules listed underneath. The four rules are conflicting in a non-trivial way. The next challenge of our work is the automated analysis of each group of undefined requests (like the one in Listing 1.6), for aiding in the resolution of the conflict.

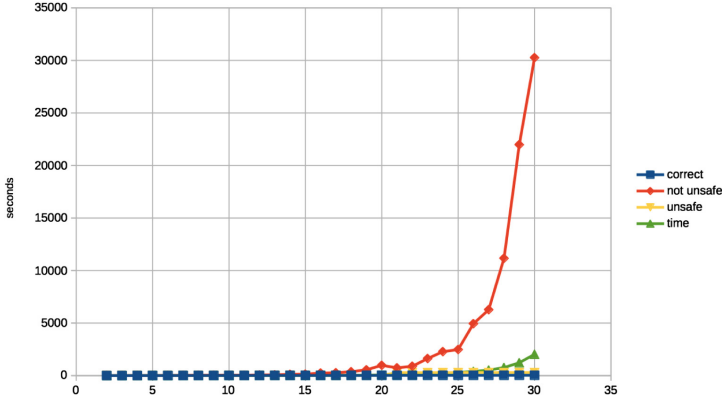


Fig. 2. Iterative experiments

Another usage of this computation is to check if a request is safe, defined or undefined. Starting from the rule classification, our prototype is able to compute the maximal safe request as defined in Sect. 5 then we can check the request against it. Listing 1.7 shows two examples: After the computation of Algorithm 1, both of these examples are processed in less than one second.

Listing 1.7. Two safety tests

```

// Intersecting the maximal safe request it is a defined request
ForAll([X, R], And(Not(PcMember(R)), PaperReviewContent(R), pcmember(X),
                  Not(subreviewer(X)), isEQuserID(X)))
// It is contained in the negation of the maximal safe request
// thus it is an undefined request
ForAll([X, R], And(PaperAssignments(R), subject(X), isConflicted(X),
                  PaperReviewContent(R), pcmember(X), isEQuserID(X)))
    
```

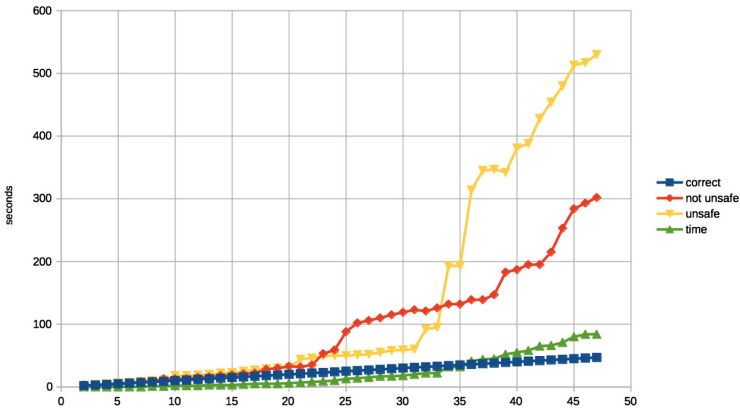


Fig. 3. Sorting implementation

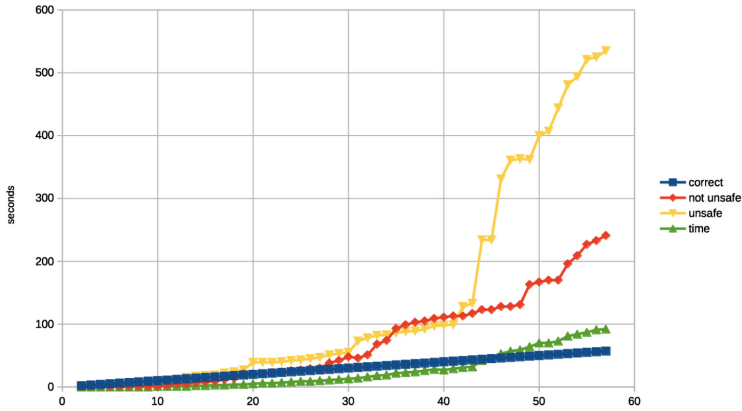


Fig. 4. Sorting test with 57 rules

6.2 Discussion

In summary, through our evaluation, we experiment our approach on a middle-size example, and observe its performances are according to our expectations and suitable for a practical use. There are also some lessons we learned.

Correctness. Decidability of the satisfiability is required for our approach to get an optimal rule classification and best performance. For example, we interact with solver to check conclusion implication, and whether a rule is obvious, unsafe or not unsafe. If the solver cannot give a reply to these questions within a given timeout, an unknown will be returned as result. This would degrade the precision of rule classification result and the performance of our approach. However, when unknown results occur, we always defensively categorize them as not unsafe, and thus will not give incorrect answers to the user.

Usability. It is important to simplify as much as possible the output, to facilitate the inspection of rule conflicts. Some simple cases are already handled in the iterative and sorting steps. For instance, two rules with equivalent conditions are simply merged into one rule with this condition and a conjunction of each conclusion. However, more aggressive simplifications are complex and time consuming. In our current solution, we think that it is better to first produce a result which alerts the user on the presence of undefined requests. In a second step, if the user wants to fix some problems we should provide a simplified version of the rules, and perhaps some hints for resolution of conflicts.

While we do not think that automatic resolution will always match users' expectation, our approach can be extended to suggest some automatic fixes to the user. For instance, the user may want to restrict its set of expected requests to the maximal set of safe requests.

Another idea is to add conditions occurring in the unsafe rules as extra conditions of the not unsafe rules. One approach is to introduce the input rules

one by one and to resolve the generated unsafe cases. If the system has no chaining of rules then there is no further problem and this way will produce a safe system. In the more general chaining case, the user should always cope with 1-undefined requests.

Generalization. While the number of rules is related to the complexity of the rule system, they do not necessarily compromise the generalization of our approach, e.g. the algorithm could take advantage of more obvious rules, or there could have more implications between conclusions.

However, we do agree that more case studies are needed to confirm the generalization for the performance and practicability of our approach. We processed another example⁴, consisting of 61 rules for managing resources, hierarchy of roles, permissions and revocation of permissions. The encoding of these rules are more complex than the CONTINUE example, e.g. predicates to represent discrete time. While the relationship between rules are more sparse (259 inclusion relations among conclusions), our sorting method is still much more efficient than the iterative one. For example, during the evaluation process, we observe that iterative method takes about 13000s to analysis 40 rules in this example. Our sorting method only takes 735s on the same set of rules. In our preliminary result on this example, we also observe a reasonable growth in its analysis time (5736s), and find 4 unsafe rules and 17085 not unsafe ones. The last unsafe rule reveals an unexpected conflict due to the hierarchy of resources and not seen in the original description.

Optimization. Currently, our approach produces a logically equivalent system for the input during its analysis. However, we think this restriction can be relaxed, e.g. a new system that is stronger than the original one could still be acceptable for analysis since it guarantees the behavior of the original. Our future work will explore new optimizations based on this kind of relaxation.

Another track of optimization could be decompose input system into sub-systems, and pave its way for a map-reduce-flavor algorithm.

7 Conclusion

In this paper we provide a new way to compute all the conflicting problems occurring in a rule based system with chaining of rules. Our methods are rather general since they require a logic extending propositional logic and a decision procedure for its satisfiability. Existing methods rely either on testing or on formal verification but they are not suitable to find the exhaustive set of potential problems. Mixing symbolic manipulations and satisfiability, we provide a decidable enumerative approach to solve this problem but due to its exponential complexity we must provide optimizations. We study two optimizations in order to reduce the number of generated rules: an iterative method with a classification of rules and the use of the topological sorting to take the maximal advantage of

⁴ RBAC and ARBAC policies for a small health care facility. <http://www3.cs.stonybrook.edu/~stoller/ccs2007/>.

relations between rule conclusions. As an evaluation we successfully apply our algorithm to a FOL rule system with more than forty rules. With this instance, rather than computing 2^{47} new rules, we produce less than 1000 rules in less than 100 s. Note that when the rule system is complex, it contains many relations between the predicates, and increases the risk of undefined requests. However, in this case our method, especially the sorting optimization, is particularly efficient. Our automatic method is able to handle middle size examples and more improvements are needed to solve larger examples in reasonable time. In our future work we expect to explore other practical optimizations, for instance by relaxing the relation of equivalence that we impose between the original rule system and its implementation. Another important research line will be to enrich our method with automatic or assisted ways to fix the detected problems.

References

1. Zacharias, V.: Development and verification of rule based systems - a survey of developers. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2008. LNCS, vol. 5321, pp. 6–16. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88808-6_4
2. Coenen, F., Eaglestone, B., Ridley, M.J.: Verification, validation, and integrity issues in expert and database systems: two perspectives. *Int. J. Intell. Syst.* **16**(3), 425–447 (2001)
3. Paschke, A.: Verification, validation and integrity of distributed and interchanged rule based policies and contracts in the semantic web. In: Semantic Web, 2nd International Semantic Web Policy Workshop (SWPW 2006). CEUR-WS.org (2006)
4. Han, W., Lei, C.: A survey on policy languages in network and security management. *Comput. Netw.* **56**(1), 477–489 (2012)
5. Hanamsagar, A., Jane, N., Borate, B., Wasvand, A., Darade, S.: Firewall anomaly management: a survey. *Int. J. Comput. Appl.* **105**(18), 1–5 (2014)
6. Aqib, M., Shaikh, R.A.: Analysis and comparison of access control policies validation mechanisms. *I.J. Comput. Netw. Inf. Secur.* **7**(1), 54–69 (2015)
7. 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)
8. Hwang, J., Xie, T., Hu, V.C.: Detection of multiple-duty-related security leakage in access control policies. In: Secure Software Integration and Reliability Improvement, pp. 65–74. IEEE Computer Society (2009)
9. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. *Fundamantae Informatica* **89**(4), 511–538 (2008)
10. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. *ACM Trans. Inf. Syst. Secur.* **11**(4), 1–41 (2008)
11. Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E.C., Bandara, A.K.: Expressive policy analysis with enhanced system dynamicity. In: Li, W., Susilo, W., Tupakula, U.K., Safavi-Naini, R., Varadharajan, V. (eds.) Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, pp. 239–250. ACM (2009)
12. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Analysis of XACML policies with SMT. In: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 115–134. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_7

13. Ni, Q., et al.: Privacy-aware role-based access control. *ACM Trans. Inf. Syst. Secur.* **13**(3), 24:1–24:31 (2010)
14. Neri, M.A., Guarneri, M., Magri, E., Mutti, S., Paraboschi, S.: Conflict detection in security policies using semantic web technology. In: *Satellite Telecommunications (ESTEL)*, pp. 1–6. IEEE (2012)
15. Armando, A., Ranise, S.: Automated and efficient analysis of role-based access control with attributes. In: Cuppens-Bouahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) *DBSec 2012. LNCS*, vol. 7371, pp. 25–40. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31540-4_3
16. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. *IEEE Trans. Dependable Sec. Comput.* **10**(6), 341–354 (2013)
17. Shaikh, R.A., Adi, K., Logrippo, L.: A data classification method for inconsistency and incompleteness detection in access control policy sets. *Int. J. Inf. Sec.* **16**(1), 91–113 (2017)
18. Deng, F., Zhang, L.Y.: Elimination of policy conflict to improve the PDP evaluation performance. *J. Netw. Comput. Appl.* **80**, 45–57 (2017)
19. Xia, X.: A conflict detection approach for XACML policies on hierarchical resources. In: *Conference on Green Computing and Communications, Conference on Internet of Things, and Conference on Cyber, Physical and Social Computing*, pp. 755–760. IEEE Computer Society (2012)
20. Royer, J.-C., Santana De Oliveira, A.: AAL and static conflict detection in policy. In: Foresti, S., Persiano, G. (eds.) *CANS 2016. LNCS*, vol. 10052, pp. 367–382. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48965-0_22
21. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) *CPAIOR 2013. LNCS*, vol. 7874, pp. 160–175. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_11
22. Previti, A., Marques-Silva, J.: Partial MUS enumeration. In: *27th AAAI Conference on Artificial Intelligence*, Bellevue, Washington, pp. 818–825. AAAI Press (2013)
23. Wu, H.: Finding achievable features and constraint conflicts for inconsistent meta-models. In: Anjorin, A., Espinoza, H. (eds.) *ECMFA 2017. LNCS*, vol. 10376, pp. 179–196. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_11
24. Adi, K., Bouzida, Y., Hattak, I., Logrippo, L., Mankovskii, S.: Typing for conflict detection in access control policies. In: Babin, G., Kropf, P., Weiss, M. (eds.) *MCETECH 2009. LNBIP*, vol. 26, pp. 212–226. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01187-0_17
25. Dunlop, N., Indulska, J., Raymond, K.: Methods for conflict resolution in policy-based management systems. In: *Enterprise Distributed Object Computing Conference*, pp. 98–111. IEEE Computer Society (2003)
26. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008. LNCS*, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
27. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: *International Conference on Software Engineering* (2005)