Removing Problems in Rule-Based Policies

Zheng Cheng[1], Jean-Claude Royer[2(✉)], and Massimo Tisi[2]

[1] ICAM, LS2N (UMR CNRS 6004), Nantes, France
zheng.cheng@icam.fr
[2] IMT Atlantique, LS2N (UMR CNRS 6004), Nantes, France
{jean-claude.royer,massimo.tisi}@imt-atlantique.fr

Abstract. Analyzing and fixing problems of complex rule-based policies, like inconsistencies and conflicts, is a well-known topic in security. In this paper, by leveraging previous work on enumerating all the problematic requests for a rule-based system, we define an operation on the policy that removes these problems. While the final fix remains a typically manual activity, removing conflicts allows the user to work on unambiguous policies, produced automatically. We prove the main properties of the problem removal operation on rule-based systems in first-order logic. We propose an optimized process to automatically perform problem removal by reducing time and size of the policy updates. Finally we apply it to an administrative role-based access control (ARBAC) policy and an attribute-based access control (ABAC) policy, to illustrate its use and performance.

Keywords: Conflict · Inconsistency · Policy · Problem · Removing · Rule

1 Introduction

Analyzing inconsistent and conflicting situations in security policies is an important area of research and many proposals exist. Several approaches are focused on detecting specific kinds of problems [2,7,9–11], while others are interested in fixing these problems [5,6,8,12,14]. We consider inconsistencies, or conflicts, or undefined requests, called *problems* here, as they lead to bugs or security leaks. Fixing these problems is difficult because of the policy size, the number of problems, their complexity and often the right fix needs human expertise. Our purpose in this paper is to suggest a new method in order to assist specifiers in fixing discovered problems in their policies. To make the conflicts explicit, we reuse a general and logical method defined in [2]. This method, in addition to revealing the problems, provides some information which can be exploited to cure those problems. In our current work we focus on *removing* some problems in the specification. Removing the problem means modifying the policy so that

Published by Springer Nature Switzerland AG 2019
G. Dhillon et al. (Eds.): SEC 2019, IFIP AICT 562, pp. 120–133, 2019.
https://doi.org/10.1007/978-3-030-22312-0_9

the problem disappears and no new problems are added. Of course we need to preserve, as far as possible, the original behavior of the policy while minimizing the time and the size of the policy update. While the final fix (i.e. obtaining a system that gives the right reply for every request) remains a typically manual activity, removing conflicts allows the user to work on unambiguous policies, produced automatically. We also believe that this step is a useful basis for future work on assisted policy fixing. We first provide a naive approach to remove the problem but its time complexity becomes exponential if we try to minimize the size of the modifications. Exploiting the enumerative method of [2] we are able to provide an optimized version of this process.

Our first contribution is the formal approach to remove a problem while minimizing the rule modifications in a first-order rule-based logical context. The second contribution is an experiment on an ARBAC policy, illustrating how to apply the approach in practice. Regarding the performance of the problem removal process, we confirm our results on another case study, based on an XACML policy, and we demonstrate getting the same minimal modifications in dividing the global time by a factor of 4.

The content of this paper is structured as follows. Section 2 describes related work in the area of fixing conflicting problems. Section 3 provides the necessary background and a motivating ARBAC example. Section 4 describes the general process to remove problems. Section 5 provides the formal results regarding the optimization of the removing process. In Sect. 6 we evaluate our method on our initial use case and another ABAC use case. Lastly, in Sect. 7 we conclude and sketch future work.

2 Related Work

Our work is under the umbrella of automatic software/system repair, which aims at automatically finding a solution to software/system bugs without human intervention. There exists extensive work under this broad topic, and we refer to [8] for a review. In this section, we discuss some of related work on automatic repairing of bugs in rule-based systems. Researchers try to find automatic fixes for various kinds of bugs, e.g. redundancies [6], misconfigurations [5]. In this work we focus on conflicts and inconsistencies in rule-based systems. These problems may lead to runtime failures in the sense that sending a request to the system, may return several incompatible replies. This separates our work from efforts that address other kinds of bugs. Meta-rules are one of the most common ways to handle conflicts in rule-based systems. The general idea is that when conflicts occur, pre-defined meta-rules (e.g. first-applicable, prioritization [3]) will govern rule applications to have compatible replies. However, the problem persists to resolve potential conflicts in meta-rules. Hu et al. propose a grid-based visualization approach to identify dependency among conflicts, which aims to guide user in defining conflict-free meta-rules [6]. Son et al. [12] repair access-control policies in web applications. They first reverse engineer access control rules by examining user-defined annotations and static analysis. Then, they encapsulate domain specific knowledge into their tool to find and fix security-sensitive operations that are not protected by appropriate access-control logic. Wu focuses

on detecting inconsistency bugs among invariant rules enforced on UML models [14]. The author presents a reduction from problem domain to MaxSMT, and then proposes a way to fix bugs by solving the set cover problem. Our approach is specific in targeting FOL rule systems and providing an automatic fix, without considering additional information, while minimizing the time to proceed and the size of the modifications.

3 Background

In this section, we introduce concepts/notations that will be consistently used in the rest of the paper. To facilitate our introduction, we illustrate on a variation of an ARBAC policy given by [13][1]. This is a middle size example with roles and a hierarchy of roles, role exclusivity constraints, permissions assignment and revocation. The original example contains 61 rules. In this work, we rewrite them in FOL, and modularize them into 4 modules, and parametrize appropriate rules with one integer for discrete time. The four modules are: roles, hierarchy and exclusion rules (11 rules), permissions (24 rules), assignment (13 rules) and assignment revocation (13 rules). In this section, we show only the role module for illustration purpose in Listing 1.1.

Listing 1.1. Rules of roles for an administrative RBAC policies

```
1  And(Patient(T, X), PrimaryDoctor(T, X)) => False    %first rule
2  And(Receptionist(T, X), Doctor(T, X)) => False
3  And(Nurse(T, X), Doctor(T, X)) => False
4  Nurse(T, X) => Employee(T, X)                       %4th rule
5  Doctor(T, X) => Employee(T, X)
6  Receptionist(T, X) => Employee(T, X)
7  MedicalManager(T, X) => Employee(T, X)
8  Manager(T, X) => Employee(T, X)
9  Patient(T, X) => PatientWithTPC(T, X)
10 Doctor(T, X) => ReferredDoctor(T, X)
11 Doctor(T, X) => PrimaryDoctor(T, X)                 %last rule
```

A *rule* is a logical implication, taking the form of D => C, with D being the condition and C the conclusion of the rule, expressed in a logical language (in our case FOL). For example, line 4 specifies that at any given time T, if X is a nurse, it is also an employee. A *rule system* (R) is simply a conjunction of rules. *Requests* are FOL expressions. When they are sent to a rule system at runtime, they will be evaluated against all rules in that system to generate *replies* (which are also FOL expressions). For example, when a request `Nurse(1, Jane)` is sent to the system shown in Listing 1.1, `Employee(1, Jane)` is implied as a reply. A request is called *undefined request*, if it is satisfiable by itself, but unsatisfiable when in conjunction with R. The phenomenon caused by an undefined request is that when it is evaluated, R would give contradictory/unsatisfiable replies, therefore making the system unrealizable.

We previously propose in [2] an optimized method to enumerate and classify all undefined requests in a rule system. The method translates the original rule

[1] The original example with comments is available at http://www3.cs.stonybrook.edu/~stoller/ccs2007/.

system into an equivalent system made of exclusive rules. Each *exclusive rule* abstracts what kind of replies will be generated, provided that a certain set of rules in the original rule system is applied. We call *1-undefined request* a request which, in conjunction with one rule alone, is unsatisfiable. One result of our approach is that any undefined request is a union of 1-undefined requests associated to exclusive rules. The exclusive rules we generate are analyzed by the Z3 SMT solver[2]. Based on the result from the solver, we separate exclusive rules into two categories:

- Unsafe exclusive rules. These are exclusive rules that under request will always return *unsat* by the solver. They abstract undefined requests that are certain to cause conflicts.
- Not unsafe exclusive rules. These are exclusive rules that under request return *sat* or *unknown* by the solver. Therefore, undefined requests, that are abstracted by not unsafe exclusive rules, are uncertain to cause conflicts (conservatively, we also pick them up to rise developer's attention).

A special representation called *binary characteristic* links each (not) unsafe exclusive rule to the original rule system. Each binary characteristic is a list of values, where the position i represents a rule at the corresponding position in the original system. Values have enumeration type with three possibilities, i.e. $0/1/-1$, indicating that the condition of the rule is negatively/positively/not presented in the exclusive rule. We call a binary characteristic *complete* if it does not contain -1, and has the length equal to the total number of rules in the original system (*incomplete* otherwise).

Listing 1.2. The roles module analysis

```
 ----------- UNSAFE --------------
[0, 0, 0, 1, 1, 1, -1, -1, -1, -1, -1]
  And(Not(Nurse(T, X)), Doctor(T, X),
      Not(PrimaryDoctor(T, X)),
      Not(Receptionist(T, X)), Patient(T, X)) => False
[1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
  And(Patient(T, X), PrimaryDoctor(T, X)) => False

... another 2 unsafe exclusive rules
 ----------- NOT UNSAFE --------------
[0, 0, 0, 1, 0, 0, 1, -1, -1, -1, -1]
  And(Nurse(T, X), Not(Doctor(T, X)), Patient(T, X), Not(PrimaryDoctor(T, X)))
     => And(PatientWithTPC(T, X), Employee(T, X))

... another 9 not unsafe exclusive rules
```

Applying the method given in [2] on the example shown in Listing 1.1, a total of 4 unsafe and 10 not unsafe exclusive rules are generated. Listing 1.2 shows a snippet of these rules. The shown rules are three typical kinds of exclusive rules, that in our experience, help in identifying problems in a rule system:

- Implicit unsafe rules, which are not contained in the original system, are the primary source of problems. They usually imply some overlapping condition

[2] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

that would cause conflicts. For example, from lines 2 to 5, the unsafe rule points out a problem, as several undefined requests. For instance in the original system `Doctor(T, X)` implies `PrimaryDoctor(T, X)` (last rule 11) which clashes with `Patient(T, X)` by rule 1. Interestingly, this problem is not mentioned under the radar of [13].
- Explicit unsafe rules (e.g. lines 6 and 7), are unsafe rules contained in the original system and whose conclusion is unsatisfiable.
- Not unsafe rules. As discussed before, they are uncertain sources of problems. It is a problem if the request implies the conjunction of the condition and the negation of the conclusion else it is an admissible request.

4 The Removing Process

Undefined requests detected by our optimized enumeration method may identify problems in the original rule system. However, fixing all identified undefined requests is not desirable, since it would result in a tautological and rather useless rule system: the system would not be able to infer new facts from the logical context. Also fixing a subset of the undefined requests is not trivial. It is difficult to guarantee that the iterative fixing process terminates, since it is hard to tell that there will be no new undefined requests after a fix.

Therefore, we propose an alternative solution, which is similar to the "quick fix" feature that appears in most of integrated develop environments. First, by analyzing the result of our enumeration method, the rule developers select a set of critical undefined requests to fix. Next, our solution performs a "quick fix" to remove the selected undefined requests. In the process, we strive to pinpoint a minimal set of rules in the original system for removing the selected undefined requests. By doing so, we minimize modifications that need to be made, and preserve the semantic of the original rule system as much as possible. Another important property of our solution is that it is effective. It means that regardless of choosing which undefined requests to fix first, applying our solution on each iteration will completely remove the selected undefined requests while not introducing new problems.

For example, let us consider the removal of the undefined requests characterized by the unsafe rule shown on lines 2 and 5 of Listing 1.2. Our approach takes its binary characteristic as input, and produces Listing 1.3 as output. The output states that one rule (i.e. the 11th rule) in the original system needs to be changed, and what it should be changed to (lines 2 and 5). Identified problems are existentially quantified (that explains the `Exists` quantifier) and composed by union with the selected rule conclusion.

Listing 1.3. Removing the Problem

```
1 Target rule: [11],
2 Suggested fix: Doctor(T, X) =>
3 Or(PrimaryDoctor(T, X), Exists([T, X], And(Not(Nurse(T, X)), Doctor(T, X),
4         Not(PrimaryDoctor(T, X)), Not(Receptionist(T, X)), Patient(T, X))))
```

Once the change has been made, applying our optimized enumeration method again will result in 10 not unsafe rules as before, but the selected unsafe rule has been removed and only the 3 explicit unsafe rules remain. At this stage, rule developers can choose once again undefined requests to remove, or stop if the rule system satisfied their expectation. In our final version of this module we simply forget this rule (the 11th) as it seems an error in the roles specification.

In what follows, we present an overview of our removing process (Sect. 4.1), and its main properties (Sect. 4.2).

4.1 Overview of Removing Process

Let U be the undefined request (represented by exclusive rules) that rule developer choose to fix. One way to remove U is by adding a new rule in the original rule system, which takes the shape of `U => False`. It explicitly alerts the rule developer on some undefined cases. However, this rule is redundant and the resulting system contains more and more rules when fixes iterate, which compromises understandability and maintainability. Another way is to globally restrict the set of input requests by adding a condition to all the defined requests. But this condition is far from readable in case we have several problems in the rule system. Therefore, for understandability and maintainability, we design a removing process that aims at automatically modifying a selected set of rules in the original system without the need to understand the real nature of the problem to fix. The principle of our removing process is to exclude U from the original rule system, i.e. $\neg U => R$ which is equivalent to

$$\forall * \bigwedge_{1 \leq i \leq n} (D_i => (C_i \vee U)) \tag{1}$$

All free variables in (1) are denoted by *, and i is the index of a rule in the rule system of size n. Moreover, U is the selected problem to fix, and represented by a FOL existential expression without free variables. Obviously modifying all the rules is not always required, and sometimes the modification of one rule alone $D => (C \vee U)$ could be sufficient. Thus we will consider how to do optimal modifications rather than modifying all the rules, that is to modify F, a subset of all the rules. We denote $R/F/U$ the system resulting by this modification.

4.2 Properties

We have illustrated how to remove selected undefined requests by pinpointing a minimal set of rules in the original system to modify. In this section, we discuss the effectiveness and semantics preservation of this approach.

Effectiveness. By effectiveness, we mean that removing the problem U does not add new problems. From now on, *all* represents the set of all the rules in R. We know that we have $R => R/F/U => R/all/U$, and $R/all/U = (R \vee U)$. It is easy to see that $R/F/U = R \vee (U \wedge R_{\neg F})$, where $R_{\neg F}$ is the subsystem of

the rules in R which are not modified. We should note that if U is a problem at least one rule should be fixed and we know that fixing all the rules may be required (if U is 1-undefined for all the rules in R). If we fixed a problem U in R and get $R/F/U$, we expect that U is not a problem of $R/F/U$ and also this does not add new problems. But if U' is a problem after the fix we have $U' => \neg(R/F/U)$ then $U' => \neg R \wedge \neg U \wedge \neg R_{\neg F}$ which implies that $U' => \neg U$ and $U' => \neg R$. Thus it means that U' was not a new problem but an already existing one for R but not included in the fix U. Note also that we do not have $R/F/U => R$ to be valid (provided that U is satisfiable) meaning that we have strictly less problems after removing U. This shows that the removing process is effective whatever the removing ordering is.

Behaviour Preservation. What behaviours of the original rule system could rule developers expect to preserve after the removing process applied. Let a given request $req = (req \wedge \neg R) \vee (req \wedge R)$, applying it to R we get $req \wedge R$. But applied to $R/F/U$ we get $(req \wedge R) \vee (req \wedge U \wedge R_{\neg F})$. Thus preserving is equivalent to $req \wedge U \wedge R_{\neg F} => req \wedge R$ which is equivalent to $req \wedge U \wedge R_{\neg F}$ unsatisfiable. If req intersects both R and its negation we get $(req \wedge R) \vee (req \wedge U \wedge R_{\neg F})$ meaning that the new reply widens the original reply. Behaviour preservation is not possible for all requests.

Property 1 (Behaviour Preservation). Let req a satisfiable request thus $req => \neg U \vee \neg R_{\neg F}$ is equivalent to $req \wedge R = req \wedge R/F/U$.

The above property states that behaviour is strictly preserved after removing U if and only if the request satisfies $req => \neg U \vee \neg R_{\neg F}$. If $req => \neg U$ the behaviour is preserved for any selection F and if $req => R$ the behaviour is preserved for any problems and any selection. The next section explain how to make the process more efficient by pinpointing a minimal set of rules in the original system for removing the selected undefined requests (Sect. 5).

5 Finding the Minimal Selection

Let U be a problem and R a set of rules, our goal is to modify R in order to make the requests in U defined. The challenge is to do that efficiently and minimizing the modifications in the rule system. We will get a new system $R/F/U$ where the rules in F are modified in order to avoid U to be undefined. The fixing principle is either to add $\neg U$ in the selected rule conditions or to add U in the rule conclusions. Modifying conclusions is simpler since we have the same enumerative decomposition for R and $R/F/U$ only the conclusions are different.

Definition 1 (Correct Fix of a Rule System). *Let R, F, U be a closed and satisfiable sentence, F is not empty, $R/F/U$ is a correct fix for R with F and U if each rule in F has its conclusion enlarged with U and U is not a problem for $R/F/U$.*

A simple fact to observe is: If F is a correct fix then any G such that $F \subset G$ is also a correct fix. Then our challenge is to find a selected set F to fix, smaller than all the rules. Thus we need to show that if U is a problem for R it is defined for $R/F/U$. It means that a direct, called here *naive*, solution is to check this property with a SAT solver.

Definition 2 (Naive Check). *$R/F/U$ is a correct fix if and only if $U \wedge R_{\neg F}$ is satisfiable.*

This comes from the fact that U is a problem, $R/F/U = R \vee (U \wedge R_{\neg F})$ and the definition of a correct fix.

It is easy to see that if U is a problem for R with a set of conditions $D_{1 \leq i \leq n}$ then $U => \bigvee_{j \in J} \exists * D_j$, where J is a subset of $1 \leq i \leq n$. Exclusive rules as built by the enumerative method have some interesting properties, particularly because $U => (\forall * D_j)$ means that only one rule (the j^{th}) applies. A *single* problem is associated to a complete binary characteristic, while a *complex* problem has an incomplete binary characteristic. From [2] we know that any undefined request U satisfies $U => \exists * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j)$, where I_1 (respectively I_0) is the set of positive (respectively negative) rules in the binary characteristic.

Property 2 (Application to Exclusive Rules). Let R an exclusive rule system if $U => \exists * D_j$ then $R \wedge U$ is equivalent to $(U \wedge (\forall * D_j \wedge C_j)) \vee (U \wedge (\exists * D_j \wedge \exists * \neg D_j)) \wedge R$.

In this paper we omit the full proofs, they can be found in the full version of the paper on our repository https://github.com/atlanmod/ACP-FIX. The proof relies on the fact that the universally quantified part triggers only one exclusive rule. We show with Property 3 that any problem found by the enumerative method can be split into disjoint parts called, respectively, *universal* and *existential* parts.

Property 3 (Universal and Existential Parts). Let U a satisfiable problem such that $U => \exists * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j)$ then $U = U \wedge (\forall * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j) \vee U \wedge (\exists * \ (\bigvee_{i \in I_1} \neg D_i \bigvee_{j \in I_0} D_j)$.

This property results from the partition of U related to the universally quantified part and its negation. Exploiting the information given by the enumerative method we expect to optimize the definedness checking for problems found by this method. We analyze now two cases: single or complex problem in order to expect to optimize the naive approach.

Property 4 (Definedness of Single Problem). Let R a rule system and a single problem $U => \exists * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j)$ with a complete binary characteristic, if $U \wedge \ \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1} C_i)$ is satisfiable then U is defined for R.

From R we can build an equivalent exclusive system using the enumerative method and thus we use Property 2. We consider the universal part of the problem, that is $U \wedge \forall * \bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j$. With these conditions only one rule

applies and others do not apply, they lead to the universal part and then the result of $R \wedge U$ comes from a single enumerative rule for R and gives $U \wedge \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1} C_i)$. We now consider the enumerative process but for $R/F/U$ since we need to prove that U is defined for it. Computing the enumerative process for $R/F/U$ gives new rules of the form:

$\forall * \ ((\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j) => ((\bigwedge_{i \in I_1 \wedge \neg F} C_i) \bigwedge_{i \in I_1 \wedge F} (C_i \vee U)))$. We start by analyzing the case of a single problem with a complete binary characteristic.

Property 5 (Removing Criterion for Single Problem). Let U a single problem with positive rules I_1 thus $R/F/U$ is a correct fix if either $I_1 \subset F$ or $I_1 \cap F \neq \emptyset$ and $U \wedge \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1 \cap \neg F} C_i)$ is satisfiable.

In this case there is a unique enumerative rule which applies and we use Property 4 for $R/F/U$. This is only a sufficient condition as we only check the universal part of the problem.

In case of a complex problem U with an incomplete binary characteristic we can obtain a set of complete binary characteristics adding digits not already in the incomplete binary characteristic. A completion $G_1 \cup G_0$ is a subset of $\{1..n\} \backslash (I_1 \cup I_0)$ with positive and negative rules.

Property 6 (Removing Criterion for a Complex Problem). Let U a complex problem, $R/F/U$ is a correct fix if $\neg F \subset I_0$ or $F \cap \neg I_0$ and $U \wedge \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1 \cap \neg F} C_i) \bigwedge_{g \in \neg I_1 \cap \neg I_0 \cap \neg F} ((\forall * D_g \forall * C_g) \vee \forall * \neg D_g)$ is satisfiable.

This criterion generalizes the previous one for single problem.

In the previous cases we defined a sufficient condition to remove a single or a complex problem. From the previous criterion and the decomposition into a universal and an existential part we have: If U is a problem for R then U is defined for $R/F/U$ if and only if the criterion for complex problem is satisfied or if $U \wedge (\exists * \ (\bigvee_{i \in I_1} \neg D_i \bigvee_{j \in I_0} D_j) \wedge R_{\neg F})$ is satisfied.

Property 7 (CNS for complex problem). If U is a complex problem, $R/F/U$ is a correct fix if and only if the universal or the existential part is defined for $R/F/U$.

5.1 Looking for Minimal Size

This subsection discusses how to find a set of rules to modify but with a minimal size. We can define a top-down and a bottom-up process to find a minimal solution. Both ways have a worst complexity which is exponential in the number of satisfiability checks. But the bottom up approach is preferable since it stops once the solution is found. Instead the top down approach, once the solution of size m is found, must prove the minimality of it by checking all the smaller combinations of size $m - 1$. The naive approach consists in modifying a subset of rules and checking the satisfiability of the new system in conjunction with the

problem to remove. Minimal core satisfiability techniques cannot be used here (for instance [14]), since they do not respect the structure of the rule system.

Using our criteria we optimize this search. We know that the set of all rules is a solution but in case of a single or complex problem it is also true if we take F as all the positive rules in the binary and its completion. Indeed, if we are looking for minimal solutions it is sufficient to look inside these positive rules. The reason is that if the criterion is satisfied with $I \cap F$ the part of F not in I does not matter and can be forgotten. Given a problem we defined a `lookup_complex` algorithm which looks for a minimal set of rules. It simply starts with the least possible solution (that is a single rule) and checks the criterion an all the combinations until reaching a minimal solution.

There are two critical points in the time performances of our two solutions: the number of rule combinations to test and the size of the expression to check for satisfiability. Both these aspects have an exponential nature in general. Exploiting the binary information the `lookup_complex` algorithm looks for less combinations than the naive algorithm. Regarding the satisfiability checking we expect to gain but the size of the formula is not a reliable indicator here. The informal reason lies in the form of the universal formula which is closed to CNF (Conjunctive Normal Form) which is at the heart of most of the solvers. To justify it we consider a problem associated to an unsafe rule. We also assume that our rule system contains only free variables and rules with a conjunction of predicates as condition and a disjunction of predicates as conclusion. If $K = 1$ the maximal number of predicates in a condition or a conclusion, the universal part can be seen as a 2-SAT CNF which satisfiability time is polynomial. If $K \geq 2$ we get CNF in the NP complete case but our optimisation relies on the transition phase phenomenon [1]. Analysing the CNF transformation we get a $2 * K$-SAT CNF and we estimate the maximal number of clauses $M \leq 2 * K * n$ while the total number of literals in the clauses is $N \geq (2 * K + n - 1)$. Thus the ratio $\alpha = M/N$ is below the threshold $2^{2*K} * ln(2) - 2 * K$, (as soon as $K \geq 2$), the area where the universal part is probably satisfiable in a small amount of time.

6 Application Examples

The purpose of this section is to show that our removing approach succeeds on middle-size examples. We will focus on removing problems coming from unsafe rules in these examples.

Our first specification is compound of the four previous modules introduced in Sect. 3. The permissions module is rather straightforward, assignment and revocation need to manage discrete time changes. In the assignment of permissions we choose to set the effect at next time. One example is `And(Doctor(T, X), Doctor(T, Y), assign(T, X, Y)) => ReferredDoctor(T+1, Y)`. In this specification the effect of a permission assignment is done at `T+1` which is a simple solution avoiding clashes with the roles module. The revocation module has similar rules to the assignment of permissions. However, we need more complex conditions because before to revoke an assignment it should have been previously done. The corresponding example for revocation of the above rule is `And(Doctor(T, X), revoke(T, X, Y),`

`(P < T), assign(P, X, Y), Not(assign(T, X, Y))) => Not(ReferredDoctor(T+1, Y)))`. An assign and a revocation are not possible at the same time instant because of inconsistency. We already analyzed the roles module and it was easy to process the three new ones in isolation since they have no unsafe rule. One interesting fact is that their composition does not generate new unsafe rules, indeed we get the three explicit unsafe rules coming from the roles module (see Sect. 4).

6.1 A Second Specification

An alternative solution for the specification of the assignment module is to write rules without changing the time instant in the conclusion. In this new specification our example above becomes: `And(Doctor(T, X), Doctor(T, Y), assign(T, X, Y)) => ReferredDoctor(T, Y)`. It generates unexpected conflicts we will solve now. Our analysis shows that we get 91 not unsafe rules and three unsafe rules in nearly 8 s. Thus using our `lookup_complex` procedure we find that these problems are all removed by modifying the rule: `[5]`. The enumerative computation of the new system shows that it has no more unsafe rule. Now these modifications could produce new interactions with the other modules. In fact only the roles module has new unsafe rules with the assignment module, indeed there are 3 new unsafe rules. These unsafe rules are coming from the negation of the 11^{th} rule in assignment and the `lookup_complex` shows that the 4^{th} rule is the minimal fix for all these problems. Fixing these three problems we compute the enumerative solution for the 4 modules together and we do not get new unexpected unsafe rules. The result was computed in nearly 5200 s and generates 20817 not unsafe rules and the three explicit unsafe rules from the roles module. This example shows that we can select some problems and remove them from the specification while minimizing the impact on the rule system.

Table 1. Measures for two policies

Usecase	Naive algorithm		Lookup algorithm		Additional measures		
	NS	NT	LS	LT	PR	DS	TF
Healthcare policy	1	1.01 s	1	0.2 s	10.1	0	509%
ContinueA policy	1.53	115 s	1.53	31 s	3.9	0	794%

We compare the `naive` and our `lookup_complex` algorithms and compute several measures which are summarized in the Table 1. We consider 123 unsafe problems occurring before the final fix in the composition of the four modules. Note that in this setting our example is not simply variable-free because we fix two rules adding some complex existential expressions. We compute[3] the following measures in Table 1: for the naive approach the mean of minimal size (NS), mean

[3] These results were computed with 10 runs when it was sensible in time, that is all cases except three (amongst 530) for the ContinueA policy.

of time (NT), the same for the lookup method with LS, LT and in addition the mean of positive rules in each problem (PR), the maximum of differences between size of the selection (DS) and the mean of the ratio: naive time divided by lookup time (factor time TF). But this example is specific on one point: the problems are not so numerous and related to some specific rules in the assignment module. Thus most of the problems (but the first three) are related to the 4^{th} rule of the assignment module.

6.2 The ContinueA Example

To consolidate our results we consider the ContinueA policy[4] we already analyzed in [2] and which was the study of several previous work [4,6]. This policy has 47 rules, which are pure first-order with at most two parameters. The original example is in XACML which forces the conflict resolution using combining algorithms. To stress our algorithms we do not consider ordering or meta-rules but a pure logical version of the rules. The result is that we have a great amount of problems amongst them 530 unsafe rules while the number of not unsafe rules is 302 (computed in 97 s). We process all the unsafe problems that is 530, see Table 1. The following observations confirm what was observed on the healthcare example, except that now we have many more problems to analyze. First we observed that the minimal set of fixing rules is generally low (between 1 and 5 rules) and this shows that finding it is relevant to minimize the modifications in the rule system. Another point is that due to the combinatorial explosion it is really costly to go up to more than 4 rules (see Table 2). The second point is that the lookup algorithm does not deviate from the naive one regarding the size of the minimal set. We do not get exactly the same selection set in 33% of the cases, due to the different ordering in the search for minimal, but the minimal sizes are always the same. Regarding the time to proceed, the lookup outperforms the naive one by a factor between 60% and 5000% with a median of nearly 800%. For this example we also compute the distribution per selection size, and the mean time for each algorithms.

Table 2. Selection distribution

Selection size	Frequency	Naive mean time	Lookup mean time	Time factor
1	63%	0.33 s	0.04 s	800%
2	27%	6.6 s	1.15 s	573%
3	8%	124 s	24 s	517%
4	3%	1573 s	281 s	560%
5	0.5%	88890	3433	256%

[4] http://cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/.

6.3 Discussion

Regarding the healthcare example, we defined two versions which have finally only three explicit unsafe rules and we remove only few problems. For the ContinueA example removing all the unsafe problems can be done modifying all the 47 rules with an increase in size of $24910 * US$, where US is the median size of the problems. Using the minimal selection of rules the increase in size is $796 * US$. The naive algorithm needs nearly 17 h to compute the minimal selections while the lookup takes 4.6 h. Our experiments also confirm that our time improvement is twofold: the restricted space to search for a minimal selection and checking first the universal part. We do not detail this here but the picture appears in our repository (https://github.com/atlanmod/ACP-FIX) as well as the source of the prototype, the examples and some measures.

It is not relevant to expect to remove all the problems. Furthermore we should also cope with the set of real requests which will decrease the amount of such undefined requests. Nevertheless an assistance should be provided to identify what are the critical problems. This is a tricky issue. The presented technique is also correct for 1-undefined problems arising in not unsafe rules. We also process some of these problems: for the 320 problems in ContinueA we get a mean for $TF = 120\%$ while with 3000 problems (nearly 10% of the problems) of the healthcare we get a mean of 800%. Thus our technique is generally useful for any kinds of problems and furthermore the reader may note that our examples after fixing are not longer simply variable-free. This is the case when we fixed the role and assign modules of the healthcare example since we add existentially quantified expressions. However, the performances were similar and we need more experiments and analysis to precisely understand the applicability of the proposed method.

Fixing a problem means to associate to U a single reply rather than inconsistent replies. This is similar to removing the problem but in addition we need to choose a reply which in general cannot be automatic. In this case the fixing principle is to change the conclusion of a rule $(D => C)$ in $(D => (C \vee (U \wedge OK))$ where OK stands for the correct reply to U. We did not yet investigate it but our current work is a good basis to solve this more complex problem.

7 Conclusion

Automatically removing problems in a policy is important to sanitize it. Sometimes there are too many problems and they are difficult to understand at least for non experts of the system. Getting simplified problems can help in solving them, however it is a complex and costly issue. Our work demonstrates that, under the conditions of the satisfiability decision and the time to proceed, we can automatically remove a selection of problems. Furthermore, we are able to minimize the size of the modifications as well as improving the time to proceed. We demonstrate it on two policies of middle size: an ARBAC and an ABAC.

This work leaves open many questions, first is about checking the existential part while getting a minimal size close to the exact minimal size. However, it

seems tricky because most of our attempts to relax the existential part lead to an unsatisfiable expression. Second, we can benefit by more case studies from related work (e.g. [6,12,14]) to statistically justify that we can get a low minimum in real cases, or we can synergize with related works. The main problem is how to faithfully encode the complete case studies (e.g. encoding explicit/implicit UML semantics plus invariants as rules in our system). The third track is to explore the benefit of checking the universal part only.

References

1. Achlioptas, D., Naor, A., Peres, Y.: Rigorous location of phase transitions in hard optimization problems. Nature **435**, 759–764 (2005)
2. Cheng, Z., Royer, J.-C., Tisi, M.: Efficiently characterizing the undefined requests of a rule-based system. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 69–88. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_5
3. Cuppens, F., Cuppens-Boulahia, N., Garcia-Alfaro, J., Moataz, T., Rimasson, X.: Handling stateful firewall anomalies. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IAICT, vol. 376, pp. 174–186. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30436-1_15
4. 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)
5. Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Martinez, S., Cabot, J.: Management of stateful firewall misconfiguration. Comput. Secur. **39**, 64–85 (2013)
6. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. IEEE Trans. Dependable Secure Comput. **10**(6), 341–354 (2013). https://doi.org/10.1109/TDSC.2013.18
7. Jha, S., Li, N., Tripunitara, M., Wang, Q., Winsborough, W.H.: Towards formal verification of role-based access control policies. IEEE Trans. Dependable Secure Comput. **5**(4), 242–255 (2008)
8. Monperrus, M.: Automatic software repair: a bibliography. ACM Comput. Surv. **51**(1), 17:1–17:24 (2018). https://doi.org/10.1145/3105906
9. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamantae Informatica **89**(4), 511–538 (2008)
10. Neri, M.A., Guarnieri, 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). https://doi.org/10.1109/ESTEL.2012.6400092
11. Ni, Q., et al.: Privacy-aware role-based access control. ACM Trans. Inf. Syst. Secur. **13**(3), 24:1–24:31 (2010). https://doi.org/10.1145/1805974.1805980
12. Son, S., McKinley, K.S., Shmatikov, V.: Fix Me Up: repairing access-control bugs in web applications. In: 20th Annual Network and Distributed System Security Symposium. Usenix, San Diego (2013)
13. Stoller, S.D., Yang, P., Ramakrishnan, C.R., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, 28–31 October 2007, pp. 445–455 (2007)
14. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. 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