Removing Problems in Policy

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

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

Abstract. Analysing conflicting situations is a major problem in many systems and a well-known topic in security. In this paper, reusing a method to make explicit all the conflicting requests, we show how to automatically remove these problems from the policy. Our proposal is proved on rule-based systems in first-order logic and we apply it to an administrative role-based access control policy and an attribute-based access control policy.

1 Introduction

Analyzing conflicting situations in security policies is an important area of research and many proposals exist. Many approaches are focused on detecting various kinds of problems [1,2,3,4,5], while some others are interested in fixing these problems [6,7,8,9,10]. 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 explicit the conflicts, we reuse a general and logical method defined in [5]. This method, in addition to reveal the problems, provides some information which are beneficial in order to cure the problems. In our current work we focus on removing some problems in the specification. Removing the problem means that we modify the policy and then the problem will disappear and no new problem will be added. Of course we need to preserve, as far as possible, the original behaviour of the policy while minimizing the time and the size to remove a problem. We provide a first naive approach to remove a problem but as soon as we consider minimizing the size of the modifications it becomes exponential. Exploiting the enumerative method of [5] we are able to provide criteria to optimize this process. Our first contribution is to formalize the criterion to make defined a problem while minimizing the rule modifications in a first-order rule-based logical context. The second contribution is an experiment on a administrative RBAC policy showing how to cure some problems. Regarding performances of the removing process we confirm our results on another policy, inspired from 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 Section 6 we evaluate our method on our initial use case and another ABAC use case. Lastly, In Section 7 we conclude and sketch future work.

2 Related Work

Our work is under the umbrella of automatic software / system repair, which aims for automatically finding a solution to software / system bugs without human intervention. There exists many work under this broad topic, and we refer to [6] for a review. In this section, we discuss some of related work on automatic repairing bugs for rule-based systems. There are various kinds of bugs in rule-based systems that researches try to find automatic fixes, e.g. remove redundancies [9], fix misconfigurations [7]. The bugs we address in this work is conflict bugs in rule-based systems. These conflicts lead to the runtime failure such that by sending a request to the system, it would return several incompatible replies. This distinguish ourselves to efforts that addressing other kinds of bugs.

Among approaches for automatic repairing conflicts in rule-based systems, meta-rules is one of the most common way. The general idea is that when conflicts occur, pre-defined meta-rules (e.g. first-applicable, prioritization [11]) 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 [9].

Son et al. [10] repairs access-control policies in web applications. They first reverse engineering 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 [8]. The author presents a reduction from problem domain to MaxSMT, and then proposes a way to fix bugs by solving the set cover problem.

3 Background

In this section, we introduce concepts/notations that will be consistently used in the rest of the paper. We refer to our previous paper [5] for more details on these concepts.

To facilitate our introduction, we illustrate on a variation of administrative RBAC policies given by [12][3]. The original example contains 61 rules. In this

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

work, we rewrite them in FOL, and modularize them into 5 modules, and parameterize appropriate rules with one integer for discrete time. Here, 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)                       %3rd 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 undefined request is that when it is evaluated, R would give a list of contradictory/unsatisfiable replies, therefore make behaviors of the system unpredictable.

We previously propose in [5] an optimized method to enumerate and classify all undefined requests in a rule system. The method translates the original system into an equivalent system of exclusive rules. Each *exclusive rule* abstracts that what kind of replies will be generated, provided certain set of rules in the original rule system are 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.

We formulate generated exclusive rules in the Z3 SMT solver[4]. Based on the result from the solver, not only we can detect all undefined requests, we can also separate them into 2 more fine-grained categories for the policy developer to review for potential problems in the given rule system:

- 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).

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

The presentation of (not)/unsafe exclusive rules uses a special representation called binary characteristics to link the output analysis result to the original rule system. Each binary characteristic at position i of output represents a rule at the corresponding position in the original system. It is an enumeration type with 3 value, i.e. 0/1/-1 indicates 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 equals to the total number of rules in the original system. Otherwise the binary characteristic is *incomplete* and contains -1 values.

Listing 1.2. The roles module analysis

```
1   ----------- UNSAFE --------------
2  [0, 0, 0, 1, 1, 1, -1, -1, -1, -1, -1]
3    And(Not(Nurse(T, X)), Doctor(T, X),
4        Not(PrimaryDoctor(T, X)),
5        Not(Receptionist(T, X)), Patient(T, X)) => False
6  [1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
7    And(Patient(T, X), PrimaryDoctor(T, X)) => False
8
9  ... another 2 unsafe exclusive rules
10  ----------- NOT UNSAFE --------------
11 [0, 0, 0, 1, 0, 0, 1, -1, -1, -1, -1]
12   And(Nurse(T, X), Not(Doctor(T, X)), Patient(T, X), Not(PrimaryDoctor(T, X)))
13      => And(PatientWithTPC(T, X), Employee(T, X))
14
15 ... another 9 not unsafe exclusive rules
```

Applying the method given in [5] on the example shown in Listing 1.1, a total of 10 not unsafe and 4 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, helping to identify problems in a rule system:

– Implicit unsafe rules, which are not contained by the original system. They usually implies some overlapping condition that would cause conflicts, and primary sources of problems. For example, from lines 2 to 5, the unsafe rule points out a problem. It represents indeed several undefined requests, for instance `Doctor(T, X)` will trigger (last rule 11) `PrimaryDoctor(T, X)` which clashes with `Patient(T, X)` and rule 1. Interestingly, this problem is not detected under the radar of [12].
– Explicit unsafe rules (e.g. lines 6 and 7), is a rule contained in the original system and which 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 identified by our optimized enumeration method clearly induce problems in the original rule system. However, fixing all identified undefined requests is rather a complicated but not rewarding task. On the one hand, it is difficult to guarantee that the iterative fixing process terminates, since hard to tell that there will be no new undefined requests occur after a fix. On the other hand, fixing all undefined requests will result a tautological and rather useless rule system, since the system cannot infer new facts from the logical context.

Therefore, we propose an alternative solution, which is similar to "quick fix" feature that appears in most of integrated develop environments. First, by analysing the result of our enumeration method, the rule developers select a set of critical undefined requests to fix. Next, our solution proposes a "quick fix" to remove selected undefined requests. In the process, we strive to pinpoint a minimal set of rules in the original system for removing selected undefined requests. By doing so, we minimize modifications that need to make, 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 selected undefined requests while not introducing new ones.

For example, let us consider to remove 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 produce Listing 1.3 as output. The output states one rule (i.e. 10th rule) in the original system that need to be changed, and what it should change 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: [10],
2 Suggested fix: Doctor(T, X) =>
3 Or(PrimaryDoctor(T, X), Exists([T, X], And(Not(Nurse(T, X)), Doctor(T, X),
4          Not(Receptionist(T, X)), Patient(T, X), Not(PrimaryDoctor(T, X)))))
```

Once the change has been made, applying our optimized enumeration method again will result 10 not unsafe rules as before, but the selected implicit unsafe rule has been removed and only 3 explicit unsafe remains. At this stage, rule developers can chose 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 as it seems an error in the roles specification.

In what follows, we present an overview of our removing process (Section 4.1), and the effectiveness of our removing process, and which semantics behaviours of the original system can be preserved after removing (Section 4.2).

4.1 Overview of Removing Process

Let U be the undefined requests (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, the result system contains more and more rules when fixes iterated, which compromises understandability and maintainability. Another way is to globally restrict the set of input requests by adding a condition, for instance 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 to automatically modify a selected set of rules in the original system. 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 undefined requests to fix, and represented by a FOL 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 note $R/F/U$ for this modification.

4.2 Discussion

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 it 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.

Semantic preservation. A natural question to ask is that: What behaviours of the original rule system could rule developer 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 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 pinpoint a minimal set of rules in the original system for removing the selected undefined requests (Section 5).

5 Finding Minimal Selection

Let U 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 is 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 [5] we now 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 $(U \wedge (\forall * D_j \wedge C_j)) \vee (U \wedge (\exists * D_j \wedge \exists * \neg D_j)) \wedge R$.*

The proof of this property appears in Appendix A, it relies on the fact that the universally quantified part triggers only one exclusive rule. We show with the 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 it 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 $U => \exists * \bigwedge_{i\in I_1} D_i \bigwedge_{j\in I_0} \neg D_j$ 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 built an equivalent exclusive system using the enumerative method and thus we use the Lemma 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 leads 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 noted U 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. The full proof is given in AppendixA.

In case of a complex problem U with an incomplete binary 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, this set of completion is noted $\mathcal{G}$.

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 and its proof is detailed in AppendixA.

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

Our second problem is to find a set of rules to modify but with a minimal size. We can define a top-down and a bottom-up processes to find a minimal solution. Both ways have a worst complexity which is exponential in the number of satisfiability checking. But the look up way is better since once the solution is found it stops. While the look down should, once the solution of size m is found, 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 [8]), 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 strictly 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 is with 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 [13]. 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 is effective 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 Section 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 than 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 Section 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)`. But it generates unexpected conflicts we will solve now. Our analysis shows that we get 91 not unsafe rules and three unsafe rules in nearly 8s. Thus using our `lookup_complex` procedure we find that these problems are all removed by modifying the rules: `[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 5200s 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 rules system.

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

Table 1. Measures for two Policies

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[5] the following measures in Table 1: for the naive approach the mean of minimal size (`NS`), mean 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 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 three first) 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[6] we already analyzed in [5] and which was the study of several previous work [14,9]. 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 seconds). We process all the unsafe problems that is 530, see Table 1. The following observations confirm what was observed on the

[5] 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.

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

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 rules 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 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.

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

Table 2. Selection Distribution

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 hours to compute the minimal selections while the lookup takes 4.6 hours. Our experiments also confirm that our time improvement is twofolds: the restricted space to search for a minimal selection and checking first the universal part. We do not detail this here but the picture in appendix A shows it.

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 anymore 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 expert of the system. Getting simplified problems can help in solving them, however it is a complex and costly issue. Our work demonstrate 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. Our future work will explore how to optimize our criteria, especially the existential case.

This work leaves open many questions, one 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. A statistical analysis could justify the fact that in real cases we get a low minimum, and another track is to explore the benefit of checking the universal part only.

References

1. Jha, S., Li, N., Tripunitara, M., Wang, Q., Winsborough, W.H.: Towards formal verification of role-based access control policies. IEEE Transactions on Dependable and Secure Computing **5**(4) (2008) 242–255
2. Montangero, C., Reiff-Marganiec, S., Semini, L.: Logic-based conflict detection for distributed policies. Fundamantae Informatica **89**(4) (2008) 511–538
3. Ni, Q., Bertino, E., Lobo, J., Brodie, C., Karat, C.M., Karat, J., Trombeta, A.: Privacy-aware role-based access control. ACM Trans. Inf. Syst. Secur. **13**(3) (July 2010) 24:1–24:31
4. Neri, M.A., Guarnieri, M., Magri, E., Mutti, S., Paraboschi, S.: Conflict detection in security policies using semantic web technology. In: Satellite Telecommunications (ESTEL), IEEE (2012) 1–6

5. Cheng, Z., Royer, J.C., Tisi, M.: Efficiently characterizing the undefined requests of a rule-based system. In: IFM 2018. (2018)
6. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys **51**(1) (2018) 17:1–17:24
7. Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N., Martinez, S., Cabot, J.: Management of stateful firewall misconfiguration. Computers and Security **39** (2013) 64–85
8. Wu, H.: Finding achievable features and constraint conflicts for inconsistent metamodels. In: 13th European Conference on Modelling Foundations and Applications, Marburg, Germany, Springer (2017) 179–196
9. Hu, H., Ahn, G.J., Kulkarni, K.: Discovery and resolution of anomalies in web access control policies. IEEE Transactions on Dependable and Secure Computing **10**(6) (2013) 341–354
10. 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, San Diego, California, USA, Usenix (2013)
11. Cuppens, F., Cuppens-Boulahia, N., García-Alfaro, J., Moataz, T., Rimasson, X.: Handling stateful firewall anomalies. In Gritzalis, D., Furnell, S., Theoharidou, M., eds.: Information Security and Privacy Research. Volume 376 of IFIP Advances in Information and Communication Technology., Springer (2012) 174–186
12. 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, October 28-31, 2007. (2007) 445–455
13. Achlioptas, D., Naor, A., Peres, Y.: Rigorous location of phase transitions in hard optimization problems. Nature **435** (2005) 759–764
14. 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)

A Proof of the Universal Criteria

Below is a proof of the Property 2. Note that if R is exclusive it is equivalent to $\forall * \bigoplus_{1 \leq i \leq n}(D_i \wedge C_i) \vee \bigwedge_{1 \leq i \leq n}(\neg D_i))$ which can be proved by recurrence on n. We split U in two parts related to $\forall * D_j$ and its negation. Since we do not have free variables in these parts we can distribute them under the scope of the outside universal quantifier. For the first part, let $R \wedge U \wedge (\forall * D_j)$, the conjunction with each $(D_i \wedge C_i)$ if $i \neq j$ it is unsatisfiable as well as the conjunction with the last term. For $i = j$ corresponding to the application of the j rule, its application is $U \wedge \forall * ((\forall * D_j) \wedge D_i \wedge C_j)$, and equivalent to $U \wedge \forall * (D_j \wedge C_j)$. $\square$

Proof of the Property 5. We analyze the case of a single problem noted U_s with a complete binary characteristic. There are three exclusive cases for the rule to apply. If $I_1 \subset F$ the result is U_s satisfiable, since the rule conclusion is $(\bigwedge_{i \in I_1} C_i) \vee U$. If $I_1 \subset \neg F$ the result is unsatisfiable, since the rule conclusion is $\bigwedge_{i \in I_1} C_i$ and U is a problem. In the remaining case, U_s is an undefined request thus $U_s = U_s \wedge \exists * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j) \wedge \neg(\bigwedge_{i \in I_1} C_i)$, where I_1 is the set of positive rules. Thus we can apply the Property 4 for $R/F/U$, the result is $U_s \wedge \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \wedge \forall * ((\bigwedge_{i \in I_1 \wedge \neg F} C_i)((\bigwedge_{i \in I_1 \wedge F} C_i) \vee U))$ equivalent to $U_s \wedge \forall * (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j) \wedge \forall * (\bigwedge_{i \in I_1 \wedge \neg F} \ C_i)$ which should be satisfiable. $\square$

The proof of the Criterion 6 is based on the binary completion and the use of the criterion for the single problems. The first step is to compute $R/F/U$ using the enumerative method which leads to rules
$\forall * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j) =>$
$((\bigwedge_{i \in (I_1 \cup G_1) \cap F} (C_i \vee U)) \bigwedge_{i \in (I_1 \cup G_1) \cap \neg F} C_i)$, where G_1, G_0 represent the completion of I_1, I_0. For the conclusion there are three cases which lead respectively to true, false, and to check a complex expression. The first case is if $\neg F \subset I_0$ then $F \subset I_1 \cup G_1$ and the conclusion of each rule contains U thus the conjunction contains U and the conjunction with it is satisfiable. In the second case, if $F \subset I_0$, there is no positive rules fixed and U is unsatisfiable with $\forall * \bigwedge_{i \in I_1 \cup G_1} C_i$ then the conjunction with $U \wedge R/F/U$ is unsatisfiable. Thus we now assume that F and $\neg F$ intersect the complement of I_0 which is $I_1 \cup G_1 \cup G_0$ and any enumerative rule is like this: $\forall * \ (\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j) \ =>$ $((\bigwedge_{i \in I_1 \cup G_1} C_i) \vee (U \wedge \bigwedge_{i \in (I_1 \cup G_1) \cap \neg F} C_i))$. Thus U is equivalent to $U \wedge \exists *$ $(\bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j)(\bigvee_{G \in \mathcal{G}} \exists * \ (\bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j)$. Then we distribute each part on $R/F/U$ and since we have complete characteristics only the rules matching it applies as in the Property 4. Thus we get a union
$\bigvee_{G \in \mathcal{G}} U \wedge \forall * (\bigwedge_{i \in I_1 \cup G_1} D_i \bigwedge_{j \in I_0 \cup G_0} \neg D_j) \forall * ((\bigwedge_{i \in I_1 \cup G_1} C_i) \vee (U \wedge \bigwedge_{i \in (I_1 \cup G_1) \cap \neg F} C_i))$ which implies $U \wedge R/F/U$. But U is a problem and thus in conjunction with $\forall *$ $\bigwedge_{i \in I_1 \cup G_1} C_i$ it is unsatisfiable. We also factorize $U \wedge \forall * \bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1 \cap \neg F} C_i$. This leads to $U \wedge \forall * \ \bigwedge_{i \in I_1} D_i \bigwedge_{j \in I_0} \neg D_j \bigwedge_{i \in I_1 \cap \neg F} C_i \wedge$
$(\bigvee_{G \in \mathcal{G}} \forall * \ (\bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j \ \bigwedge_{i \in G_1 \cap \neg F} C_i))$. Let $G_1 \cap \neg F$ we can factorize every expression containing it getting $\bigwedge_{i \in G_1 \cap \neg F} C_i \bigwedge_{i \in G_1 \cap \neg F} D_i \bigwedge_{j \in G_0} \neg D_j \wedge$ $(\bigvee_{G \in \mathcal{GF}} D_G)$ where $\mathcal{GF}$ is the completion for rules in $\neg I_1 \cap \neg I_0 \cap F$. It simplifies in $\bigwedge_{i \in G_1 \cap \neg F} C_i \bigwedge_{i \in G_1 \cap \neg F} D_i \bigwedge_{j \in G_0} \neg D_j$ and the union becomes $(\bigvee_{G \in \mathcal{G}/\mathcal{F}} \forall *$ $(\bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j \ \bigwedge_{i \in G_1 \cap \neg F} C_i))$, where $\mathcal{G}/\mathcal{F}$ is the subsets of rules in $\neg I_1 \cap$ $\neg I_0 \cap \neg F$. We remark that $\bigwedge_{h \in G \cap \neg F} (D_h \wedge C_h) \vee \neg D_h =$
$\bigvee_{G \in \mathcal{G}/\mathcal{F}} (\bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j \ \bigwedge_{i \in G_1 \cap \neg F} C_i))$. Then a final factorization produces $U \wedge \forall * \ (\bigwedge_{i \in G_1} D_i \bigwedge_{j \in G_0} \neg D_j \ \bigwedge_{i \in I_1 \cap \neg F} C_i) \wedge (\bigwedge_{g \in G \cap \neg F} ((\forall * \ D_g \wedge \forall *$ $C_g) \vee (\forall * \neg D_g))$. $\square$

B Additional Results

The fact that the minimal selection has a small size can surely be founded by a statistic or probabilitic analysis. A consequence of it is that in the naive approach there is no significative difference between checking $U \wedge R$ and $U \wedge R_{\neg F}$. To confirm that our time improvement is twofolds we did some complementary experiments, see Figure 1 for results in the case of the ContinueA policy and the unsafe problems. We also observed that evaluating only the universal part gives results close to the real minimum while the time performance was not significantly different from evaluating also the existential part. We test three different algorithms:

- A: The naive approach

– B: The naive approach with a selection based on the binary characteristic
– C: The lookup with the selection and checking the universal then the existential formula.

In this figure we report the time factors of the comparaison between A-B, B-C and A-C. In average both features have an impact on the time factor but the benefits are reversed: the benefit of testing the universal part decreases with the binary characteristic size while it is the opposite for the selection.

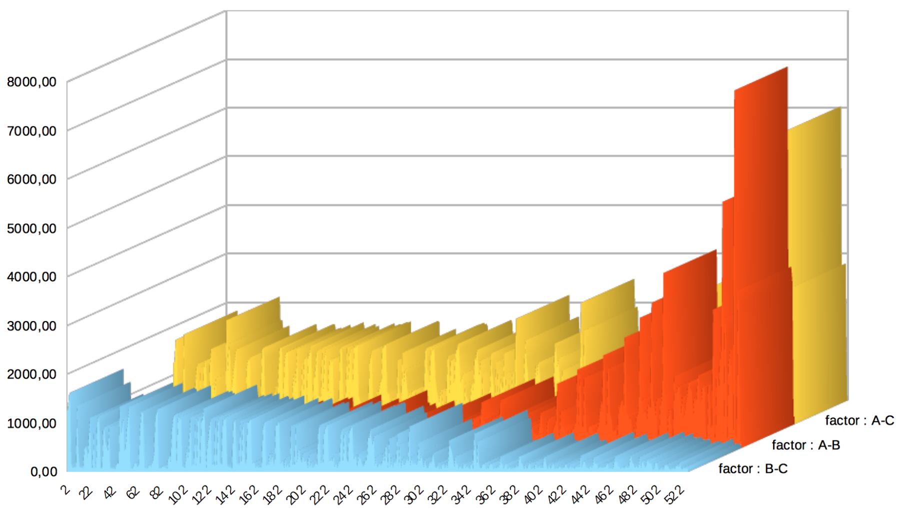

Fig. 1. Time factors