ModCon algorithm for discovering security policy conflicts*

Bartosz Brodecki, Jerzy Brzeziński, Piotr Sasak, and Michał Szychowiak

Poznań University of Technology
Piotrowo 2, 60-965 Poznań, POLAND
{BBrodecki,JBrzezinski,PSasak,MSzychowiak}@cs.put.poznan.pl

Abstract. This paper considers the problem of modality conflicts in security policies for distributed environments. An universal and efficient algorithm for discovering modality conflicts (ModCon) is proposed. The algorithm is compared with an ad-hoc approach for solving such conflicts, in order to demonstrate the attained efficiency gain.

1 Introduction

Security policy has become today indispensable for the management of security restrictions and obligations in modern distributed computing systems, and especially in service-oriented environments complying with the SOA (Service-Oriented Architecture, [1]) model. In general, security policy rules explicitly specify the allowed (or prohibited) actions which can (cannot) be performed on particular system objects (e.g. services). Typically, the policies can grow very large (in the number of policy rules) and suffer from problems, like redundancy, modality conflicts [2], separation of duty [3] or ambient authority [4], among others. In this paper we consider modality conflicts which arise when for any single access operation two or more policy rules lead to ambiguous access control decisions.

Usually, modality conflict detection algorithms considered in literature have used an ad-hoc approach of comparing all pairs of policy rules [5]. This approach can be improved by reducing the number of necessary comparisons. Ehab S. Al-Shaer and Hazem H. Hamed have presented [6] a modality conflict discovery algorithm for firewall policies which uses a specific tree structure to represent all rules, significantly reducing the search space. Since those structures are strictly profiled for network addresses, their work applies to policy rules for firewalls or similar devices only.

F. Baboescu and G. Varghese have developed another interesting algorithm for detecting modality conflicts [7]. They also analyse only firewall rules and use a binary tree to model hierarchy of network addresses (separately for source and for destination addresses). Binary vectors are created in nodes of the binary tree

* Acknowledgment: The research presented in this paper was partially supported by the European Union in the scope of the European Regional Development Fund program no. POIG.01.03.01-00-008/08.

to represent relations between source (or destination) addresses of firewall policy rules which use them. Conflict discovery consists in finding common parts of two vectors (one for source address and one for destination address). This algorithm, although very efficient, can only be used for numerical parameters of policy rules (like addresses). Addresses are relatively simple objects. In modern distributed systems, policy objects are often more complex and more complicated relations between them exist.

Jorge Lobo, Emil Lupu et al. [8] have investigated a first–order predicate language to model and analyse general security policies. They use abductive reasoning to estimate if a security policy is free of modality conflicts. The actual conflict detection requires conducting the reasoning separately for each rule, one at a time. Due to the use of a specific policy language, the time complexity of this detection is polynomial with respect to the size of the policy (the number of policy rules). However, this approach does not really support the conflict resolution, since the result of reasoning states only whether the considered rule causes any conflict or not. It does not reveal which other policy rules are in conflict with the considered one.

The above analysis shows, that the existing proposals are inefficient or have limited applicability. Thus, new efficient and universal algorithms for discovering modality conflicts in security policy are strongly required. In this work we propose such a solution for compound policy rules which offers a significantly better efficiency than the ad-hoc approach, similar to those offered by the algorithms described above. Since our proposal is much more universal, it can be considered as advancement in policy conflicts resolution. Compared to the existing solutions, the algorithm must deal with more complex policy objects and thus requires a significantly novel approach.

This paper is organized as follows. The next Section briefly overviews the modality conflict problem in security policies. Section 3 describes first *ad-hoc* approach to discover modality conflicts in a security policy. Then, we propose a new algorithm of significantly lower time complexity. Next, Section 4 discuses the time complexity of the proposal, and in Section 5 we supplement it with an excerpt of experimental results. Finally, Section 6 gives some insights on further research, concluding this paper.

2 The modality conflicts problem

A security policy is a set of rules specifying access control and other security requirements for interactions between system components. Policy rules are usually specified in some form of *subject–action–target* (or SAT) format [9]. The access control authorizations are defined with the use of *restriction* rules. Other types of rules include obligations and capabilities, used to define required protection mechanisms imposed on the authorized interactions. Therefore, we consider policy $\hat{\mathbb{P}}$ to consist of the following elements: restriction policy $\hat{\mathbb{R}}$, obligations policy $\hat{\mathbb{O}}$ and capabilities policy $\hat{\mathbb{C}}$, each composed of the appropriate type of policy rules. Thus, the restriction policy $\hat{\mathbb{R}}$ (considered in this work) is

a set $\{R1, R2, ..., Rk\}$ of all restriction rules in policy $\hat{\mathbb{P}}$. Every restriction rule $Ri(Si, Ti, Ai, Ci) = Di$ is composed of the following *elements*: a *subject* Si, a *target* Ti, *actions* Ai, *conditions* Ci, and a *decision* Di. The *subject* specifies a human user, a user role or an automated client (or group of such elements) to which the policy applies. The *target* specifies the objects (or group of objects) on which action are to be performed. Both subjects and targets are commonly referred to as policy *principals*. The *action* specifies what is permitted for authorization. The *condition* specifies a boolean constraint when the rule is applicable. The *decision* specifies whatever the rule has positive or negative authorization.

In general, modality conflicts are known to arise when two or more policy rules refer to the same subjects and targets but lead to conflicting policy decisions about the allowed actions [2]. As the rule elements can be compound objects, it makes the problem hard to handle. Indeed, typically, some hierarchical relations exist between the principals: for instance, the subjects will belong to user groups or roles and similar targets are often aggregated into groups. Also, policy actions are represented as sets of elementary operations (e.g. read, write, invoke). Finally, each condition is represented by a set of pairs: a condition type and its value (e.g., time={weekday 8:00–15:00}). Therefore, the main difficulty of conflict discovery lies in dealing not with single elements but with compound sets of elements, in a typical case.

Definition 1. ***Modality conflicts*** *are inconsistencies in the security policy specification, which arise when for two rules with opposite decisions there exists nonempty intersection of subjects, targets, actions, and conditions.*

Let us consider a sample restriction policy $\hat{\mathbb{R}}$, which actually includes some modality conflicts. Listing 1.1 presents the policy rules in ORCA language [10], while Listing 1.2 shows the formal representation of the policy with an additional hierarchical relation of policy principals.

As you can see from Listing 1.2, there are two rules $R1$ and $R2$ giving different decisions. Both rules use different subjects ($S1$, $S2$), targets ($T1$, $T2$), actions ($A1$, $A2$) and conditions ($C1$, $C2$). Then, we have a declaration of rule elements (e.g. $S1$ is defined as Role1). For instance, Role1 and Role2 are grouped into Role0 (Role is *parent* for Role1 and Role2). Fig. 1 presents sample hierarchies of subjects and targets in the form of a graph.

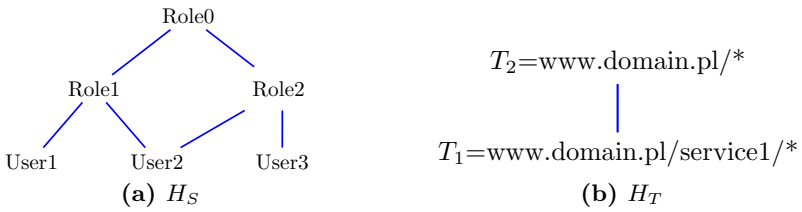
Listing 1.1: Sample restriction policy $\hat{\mathbb{R}}$ in ORCA language

```
Role1 can access www.domain.net/service1/* for {read, write},
    if time=8:00-16:00.
Role2 cannot access www.domain.net/* for {write}, if time
   =12:00-18:00 and holiday.
```

Listing 1.2: Restriction policy $\hat{\mathbb{R}}$ in formal representation

```
1 R1(S1, T1, A1, C1)=Accept.
2 R2(S2, T2, A2, C2)=Deny.
3 S1={Role1}. // declaration of subjects S1 and S2
4 S2={Role2}.
5 T1={"www.domain.net/service1/*"}.   // declaration of Targets
```

```
T2={"www.domain.net/*"}.
A1={read, write}. // declaration of actions allowed
A2={write}.
C1={time=8:00-16:00}. // declaration of conditions
C2={time=12:00-18:00; holiday}.
```



(a) H_S **(b)** H_T

Fig. 1: Sample hierarchy for policy $\hat{\mathbb{R}}$ from Listing 1.2

3 Discovery of modality conflicts

In this Section, we will describe two modality conflict discovery algorithms supporting any policy rules with condition predicates. The first one represents an ad-hoc approach. It will only serve for further reference in complexity analysis. The second algorithm, ModCon, is the actual contribution of this paper.

3.1 Ad-hoc algorithm

The ad-hoc algorithm compares all pairs of rules from policy $\hat{\mathbb{R}}$ with each other to check if they are in a modality conflict. The first step of the algorithm is to check whether rules have different decisions. Then, the algorithm compares the subjects from both rules (the algorithm must be provided with knowledge of the hierarchy of subjects H_S and targets H_T). If the subjects have any common parts (e.g., one role includes the other), then the algorithm checks both targets in the same way as the subjects. Next, the algorithm checks rule actions for any common parts. The last check concerns conditions represented as pairs {type, value}. The algorithm compares only values of the same type. Finally, if all those comparisons discover any common parts, then the two rules are in a modality conflict.

It is easy to see that the ad-hoc algorithm requires $O(n^2)$ rule comparisons, where n is the number of policy rules (for simplicity we consider here each rule comparison as a single step, although, it will actually require to compare compound rule elements, increasing the overall time complexity).

3.2 Improved Modality Conflict discovery (ModCon) algorithm

Now we introduce the improved Modality Conflict discovery algorithm (ModCon) aimed at reducing the number of necessary comparisons of policy rule pairs.

ModCon will use a graph representation of policy principals to initially select rules that might be in a *potential* conflict with any others. Only those rules will be further compared against an actual modality conflict. The expected gain over the ad-hoc algorithm is increased efficiency, as the cost of building the necessary graph will be in general smaller then the cost of full comparison of policy rules (i.e. $O(n^2)$).

ModCon will use additional data structures representing the following relation between policy principals:

Definition 2. *When there exist two principals ($P1$, $P2$) and one of them ($P1$) is a parent of the other ($P2$), then the second principal is* ***directly descendant*** *from the first one (denoted $P1 \rightarrow P2$).*

Note that restrictions of principal $P1$ are inherited by principal $P2$, if there exists principal descendance $P1 \rightarrow P2$.

The ModCon algorithm must be provided with the knowledge about the hierarchy of all principals in order to discover all potentially conflicting rules. **Policy Object Graph** is a directed graph $\overrightarrow{POG}$ representing that hierarchy. The starting node of the $\overrightarrow{POG}$ is wildcard any ("*"). Arcs represent the direct principals descendance, i.e. arc from Px to Py exists in $\overrightarrow{POG}$, if $Px \rightarrow Py$ according to Definition 2. Additionally, by $\overline{POG}$ we will denote a undirected instance of $\overrightarrow{POG}$. The $\overline{POG}$ will be further used only for cycle discovery. We will refer to both representations altogether, simply as POG.

Definition 3. *Path. Path $\overline{P}$ in $\overrightarrow{POG}$ is a sequence of principals, such that each principal is directly descendant from the previous in that sequence, e.g. $\overline{P} = * \rightarrow Role0 \rightarrow Role1 \rightarrow User1$.*

Definition 4. *If there exists a path $\overline{P}$ from a Px into Py, i.e. $Px \rightarrow P_i \rightarrow P_{i+1} \rightarrow ... \rightarrow P_{i+k} \rightarrow Py$, then we say that Py is* ***indirectly descendant*** *of Px.*

If there are two distinct paths having common starting and ending nodes then, there exists a cycle in $\overline{POG}$, which includes all the nodes from both paths. Remark, for the hierarchy in Fig. 2 one cycle exists which includes the following nodes: Role0, Role1, User2, Role2.

For the further discovery of complex dependencies between principals, we distinguish a special case of a POG cycle.

Definition 5. ***Inheritance cycle*** *is a cycle in the POG, which does not include any nodes representing users.*

Remark, in Fig. 2 there is not any inheritance cycle, because principal User1 is a user.

Definition 6. *Hierarchical dependency. A hierarchical dependency of principals (subjects or targets) is one of the following:*

1. Equality (type DT1): Both principals are literally the same.

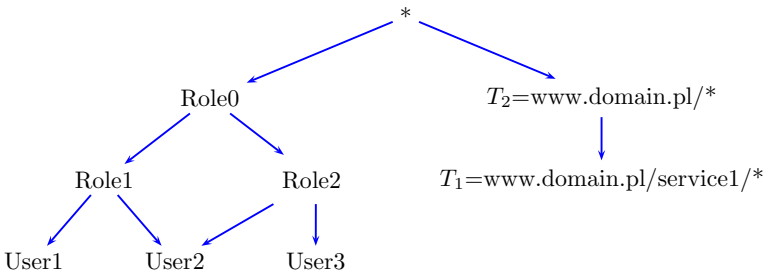


Fig. 2: POG for the hierarchy from Fig. 1

(a) Membership (DT2): One principal includes the second one (which can make a direct or indirect principal descendance).
(b) Inheritance cycle (DT3): Both principals belong to the same inheritance cycle.

Preparation phase The ModCon algorithm requires a preparation phase (Alg. 1) which creates the necessary data structures. The first step of the preparation phase involves creation of POG. Then, all the nodes of POG are extended with binary vectors of the size equal to the number of policy rules (n). The subject of i-th policy rule (Ri), further referred to $Ri.Subject$, is represented in POG as a single node. With that node the ModCon algorithm maintains binary vector BV_S such that $BV_S[i] \leftarrow 1$ (reflecting the fact that the subject is used by the i-th rule). Similar vectors BV_T are maintained for the targets of all rules.

In order to detect hierarchical dependencies between the policy principals, we will propagate the binary vector values through the paths in POG graph. All BV_S will propagate their values down the POG graph to the descendant nodes (Alg. 1 lines 8–14). Moreover, for each inheritance cycle in POG, BV_S from all the nodes belonging to that cycle will logically summate (line 33) to reflect the existence of the inheritance cycle.

In POG, the BV_T will propagate in two directions: up (lines 15–21) and down (lines 22–28) the graph. Therefore, each POG node will maintain three different types of BV_T:

1. For the rule target of a given POG node (denoted as BV_T).
2. For descending propagation of the BV_T down the $\overrightarrow{POG}$ graph ($\overrightarrow{BV}_T$) – to reflect the descendance of targets downward the hierarchy.
3. For ascending propagation of the BV_T up the $\overrightarrow{POG}$ ($\overleftarrow{BV}_T$) – to reflect hierarchical dependency of targets upward (also necessary in some circumstances, as we will show later).

Propagation of vectors through an inheritance cycle in the POG graph will summate $\overrightarrow{BV}_T$ vectors in the same way as propagation of BV_S vectors in the POG graph (lines 29–40).

```
1: function Preparation(RestrictionPolicy {R1, ..., Rn}, Hierarchy {H_S, H_T})
2:     POG ← a graph from H_S, H_T
3:     ℂ ← the set of all inheritance cycles in POG

4:     for i ← 1..n do
5:         POG(Ri.Subject).BV_S[i] ← 1    :: set BV_S[i] in node Ri.Subject
6:         POG(Ri.Target).BV_T[i] ← 1
7:     end for

8:     foreach S in POG do    :: descending propagation
9:         if POG(S).BV_S ≠ ∅ then
10:            foreach direct descendant D of S do    :: i.e. ∀D: S → D
11:                POG(D).BV_S ← POG(D).BV_S ∨ POG(S).BV_S
12:            end for
13:        end if
14:    end for

15:    foreach T in POG do    :: descending propagation
16:        if POG(T).BV_T ≠ ∅ then
17:            foreach direct descendant D of T do
18:                POG(D).BV_T (→) ← POG(D).BV_T ∨ POG(T).BV_T ∨ POG(D).BV_T (→)
19:            end for
20:        end if
21:    end for

22:    foreach T in POG do    :: ascending propagation (from leaves to the root)
23:        if POG(T).BV_T ≠ ∅ then
24:            foreach parent D of T do
25:                POG(D).BV_T (←) ← POG(D).BV_T ∨ POG(T).BV_T ∨ POG(D).BV_T (←)
26:            end for
27:        end if
28:    end for

29:    foreach cycles c ∈ ℂ do    :: all BV in a cycle must be the same
30:        commonBV_S ← ∅
31:        commonBV_T ← ∅
32:        foreach node n ∈ c do
33:            commonBV_S ← commonBV_S ∨ POG(n).BV_S
34:            commonBV_T ← commonBV_T ∨ POG(n).BV_T (→)
35:        end for
36:        foreach node n ∈ c do
37:            POG(n).BV_S ← commonBV_S
38:            POG(n).BV_T (→) ← commonBV_T
39:        end for
40:    end for

41:    return POG
42: end function
```

Alg. 1: Preparation phase of the ModCon algorithm

ModCon algorithm For all policy rules ModCon looks for three BV, one for subject ($POG(Ri.Subject).BV_S$) and two for target ($POG(Ri.Target).\overrightarrow{BV}_T$ and $POG(Ri.Target).\overleftarrow{BV}_T$). These three BV will be used to check which rules potentially conflict with Ri. ModCon algorithm checks BV to find out which rules reflected in $POG(Ri.Subject).BV_S$ are also reflected in $POG(Ri.Target).\overrightarrow{BV}_T$ or in $POG(Ri.Target).\overleftarrow{BV}_T$. These rules create a ***set of potential conflicts*** and will be then processed by rule comparison algorithm (a revised version of ad-hoc).

ModCon makes use of a slight modification of the ad-hoc algorithm described formerly, presented in Alg. 2. It will no longer compare subjects and targets. It will only compare actions and conditions of those rules which have been formerly suspected as a potential conflict. Shall any of these comparisons give a negative result, the compared rules are definitely not conflicting. Otherwise, these rules are discovered as being in an actual modality conflict.

```
1: procedure RuleComparison(PotentialConflictSet {{R1, R2},...,{R2k−1, R2k}})
2:     for i ← 1..k do
3:         if Ri ≠ Ri + 1 then
4:             if Ri.decision ≠ Ri + 1.decision then
5:                 if CompareA(Ri.action, Ri + 1.action) and
   CompareC(Ri.condition, Ri + 1.condition) then
6:                     ConflictSet ← ConflictSet ∪ {(Ri, Ri + 1)}
7:                 end if
8:             end if
9:         end if
10:    end for
11: end procedure
12: function CompareA(A1, A2)    :: A1 and A2 are sets of actions
13:    if A1 ∩ A2 ≠ ⊘ then
14:        return true
15:    end if
16:    return false
17: end function
18: function CompareC(C1, C2)    :: C1 and C2 are sets of conditions
19:    c ← condition_predicate
20:    foreach condition_predicate_type c ∈ C1 do
21:        if c ∈ C2 and C1[c] ∩ C2[c] = ⊘ then    :: intersection of condition
   values
22:            return false
23:        end if
24:    end for
25:    return true
26: end function
```

Alg. 2: Pseudocode of the rule comparison procedure

The main part of the ModCon algorithm is presented in Alg. 3. It takes two arguments. The first one is restriction policy $\hat{\mathbb{R}}$, i.e. a set of n rules (Ri, where i=1..n). The second argument is the hierarchy of all principals (H_S and H_T) needed for creating POG (line 4). The first step consists in checking if a rule (Rj) is in **potential conflict** with others rules (lines 5—7). This is done by finding intersections (at lines 11 and 12) between binary vectors BV_S, $\overrightarrow{BV}_T$ and $\overleftarrow{BV}_T$. The intersection vectors (BV_1 and BV_2) will be set at positions of potentially conflicting rules (lines 14—18). These rule pairs (Rj and every potentially conflicting rule) will be added into **PotentialConflictSet** (line 16). The last step in this algorithm is executing the RuleComparision procedure **PotentialConflictSet** (line 8).

```
1: procedure ModCon(RestritionPolicy {R1, ..., Rn}, Hierarchy {H_S, H_T})
2:     ConflictSet ← ∅
3:     PotentialConflictSet ← ∅
4:     Graph POG ← Preparation({R1, ..., Rn}, {H_S, H_T})
5:     for i ← 1..n do
6:         ConflictDetect(Graph POG,Rule Ri)    :: determine PotentialConflictSet
7:     end for
8:     RuleComparison(PotentialConflictSet)    :: determine ConflictSet
9: end procedure
10: procedure ConflictDetect(Graph POG, Rule R)
11:     BV1 ← POG(R.Subject).BV_S ∧ POG(R.Target).→BV_T
12:     BV2 ← POG(R.Subject).BV_S ∧ POG(R.Target).←BV_T
13:     BV ← BV1 ∨ BV2
14:     if number of bits set in BV ≥ 2 then
15:         foreach Ri :: BV[i] == 1, where i = 1..n do
16:             PotentialConflictSet ← PotentialConflictSet ∪ {R, Ri}
17:         end for
18:     end if
19: end procedure
```

Alg. 3: Main part of the ModCon algorithm

From Definition 1 and Definition 2 a modality conflict between two rules occurs only when the subjects of both rules are *hierarchically dependent* (one is directly or indirectly descendant from the other) and the targets are hierarchically dependent.

Initially, ModCon will try to discover all the pairs (or more precisely – tuples, since there can be more rules in the mutual conflict) of potentially conflicting rules for which the above condition holds. Then, in the set of such potentially conflicting rules, it will process each pair and exclude those pairs which do not fulfill Definition 1. The remaining rules are finally considered to be in a modality conflict. Let us assume that there exists a hierarchical dependency between subjects $S1$, $S2$ of two rules $R1$, $R2$. For the simplicity of presentation, let $R1$

and $R2$ be the only rules in the policy (thus, the size of BV vectors will be 2). We will show, that this dependency will be undoubtly reflected by BV_S vector in the ConflictDiscovery procedure.

First, suppose that $S1$ and $S2$ are literally the same ($S1 = S2$), we will denote it simply as S. Let $POG(S).BV_S$ be the binary vector of node S in $\overrightarrow{POG}$ in the preparation phase of the ModCon algorithm. According to line 5 of Alg. 1 $POG(S).BV_S[1] \leftarrow 1$ and $POG(S).BV_S[2] \leftarrow 1$, as S appears in both $R1$ and $R2$. This appearance will be reflected in the main discovering phase (Alg. 3) when procedure ConflictDiscovery processes rule $R1$. Precisely, it will be reflected by vector $POG(S).BV_S = [11]$ further used in deciding about the content of *PotentialConflictSet* (line 14).

Next, let us suppose that $S2$ is directly descendant from $S1$ ($S1 \rightarrow S2$). Then, in the $\overrightarrow{POG}$ graph an arc exists from $S1$ to $S2$. $POG(S1).BV_S$ is the binary vector of node $S1$ created in $\overrightarrow{POG}$ in the preparation phase of the ModCon algorithm. Similarly, $POG(S2).BV_S$ is the binary vector of node $S2$. According to line 5 of Alg. 1 $POG(S1).BV_S[1] \leftarrow 1$ and $POG(S2).BV_S[2] \leftarrow 1$. Since $S1$ has a direct descendant $S2$ then the propagation of BV makes $POG(S2).BV_S \leftarrow POG(S2).BV_S \vee POG(S1).BV_S$ (lines 8 to 14 of Alg. 1) which results in $POG(S2).BV_S \leftarrow [01] \vee [10] = [11]$. The main part of ModCon (Alg. 3) starts with processing rule $R1$ (line 5), and according to $POG(S1).BV_S = [10]$ then subject $S1$ appears only in rule $R1$ (because in BV_S bit is set only in the first place). Next, ModCon processes rule $R2$, and this time $POG(S2).BV_S = [11]$. As both, the first and second bit of BV_S is set, ModCon realizes the dependency of subjects of the first and the second rule ($S1$ and $S2$).

Suppose that $S2$ is indirectly descendant from $S1$. Then, according to lines 8 – 14 of Alg. 1, every direct descendant Sx of $S1$ will sum its $POG(Sx).BV_S$ with $POG(S1).BV_S$ and propagate the result to its descendants. And every descendant will do the same with its own BV. In consequence, $S2$ will finally sum $POG(S2).BV_S$ with the BV propagated from its ascendants. In this new $POG(S2).BV_S$, the bits corresponding to both rules $R1$ and $R2$ will be set. Then, when the main discovering phase starts with processing rule $R2$, then a dependency between $S1$ and $S2$ will be found.

Finally, we can consider a hierarchical dependency of the form of an inheritance cycle between $S1$ and $S2$. Suppose that $S1$, $S2$, Sx and Sy form the inheritance cycle in the $\overline{POG}$ graph. At the end of the preparation phase all the nodes in the cycle will have the same BV_S, the sum of all BV_S from these nodes (line 33 of Alg. 1), which results in $commonBV_S = [11]$. The main discovering phase (Alg. 3) will set $POG(S1).BV_S = [11]$ for $R1$. Because the first and the second bit of BV_S are set, ModCon sees the dependency between subject $S1$ of rule $R1$ and subject $S2$ of the second rule $R2$.

ModCon also realizes any hierarchical dependency between targets $T1$, $T2$ of two rules $R1$, $R2$. The main difference is that propagation of BV_T in POG creates two vectors $\overrightarrow{BV}_T$ for the descending propagation and $\overleftarrow{BV}_T$ for the ascending propagation. This time, the hierarchical dependency will be undoubtedly reflected by either $POG(R.Target).\overrightarrow{BV}_T$ or $POG(R.Target).\overleftarrow{BV}_T$ vector

in procedure ConflictDiscovery. If $T2$ is descendant from $T1$, the algorithm finds $POG(T1).\overleftarrow{BV}_T = [11]$. If $T1$ is descendant from $T2$, then $POG(T1).\overrightarrow{BV}_T = [11]$.

The ModCon algorithm will include any two rules ($R1$, $R2$) in a *possible conflict set*, when at the same time their subjects are hierarchically dependent and their targets are hierarchically dependent. Remark that a discovering of a possible conflict between two rules will go through two rounds (lines 5–7 of Alg. 3), the first round for $R1$, and the second for $R2$. In many cases any one of those round will discover the conflict. But in few cases, only one of those rounds will do that correctly. For illustration, assume that the dependency between subjects is $S1 \rightarrow S2$ and targets are in any possible dependency. When ModCon starts processing rule $R1$, then it will not discover a potential conflict (since $POG(S1).BV_S = [10]$ and regardless of values of $POG(T1).\overrightarrow{BV}_T$ and $POG(T1).\overleftarrow{BV}_T$, the final vector $BV = [10]$ will not show any potential conflict yet). However, in the second round, ModCon starts processing rule $R2$, and then $POG(S2).BV_S = [11]$. Now, whatever dependency between targets will be, one of vectors $POG(T1).\overrightarrow{BV}_T$, $POG(T1).\overleftarrow{BV}_T$ will have value [11]. Concluding, from the final vector $BV = [11]$, ModCon will now know that both rules are in a potential conflict.

For all rule pairs from *PotentialConflictSet* if actions of both rules have common parts and conditions also have common parts, and decisions are contrary (i.e. a modality conflict occurs), the pair is included in the *ConflictSet* (i.e. finally discovered). As a result, ModCon will include all pairs of rules matching the definition of a modality conflict in the *ConflictSet*.

The formal proof of correctness of a preliminary version of the ModCon algorithm has been presented in [11].

4 Time complexity

We will conduct the analysis of time complexity separately for all three parts of the ModCon algorithm (i.e., preparation phase, rule comparison phase, and main part).

Preparation phase Let us start from the preparation phase (Alg. 1).

Creation time of the POG graph is proportional to the number of principals in hierarchy. Denoting the hierarchy size as $|H|$, we get the time complexity $O(|H|)$.

In lines 4–7 a loop will be executed n times (where n is the number of rules in the policy), to set bits in BV for every subject and every target from policy rules – thus we get $O(n)$ here.

Propagation of BV_S (lines 8–14) takes $|H|$ steps – $O(|H|)$. And propagation of BV_T (descending in lines 15–21, ascending in lines 22–28) takes twice $|H|$ steps – $(O(|H|)$.

At the end of this phase, propagation of BV inside each inheritance cycle (lines 29–40) takes l_c steps, where l_c is the number of nodes inside c-th inheritance cycle. Thus this part takes $\sum_c l_c$ steps. Since, in the worst case, l_c may be equal to $|H|$ (when all the POG nodes form an inheritance cycle), we get time complexity of $O(|H|)$.

Reassuming, the time complexity of this phase linearly depends on the hierarchy size ($O(|H|)$).

Rule comparison phase Rule comparison phase (Alg. 2) will only be executed when at least one rule pair exists in the *PotentialConflictSet*.

In line 2 a loop starts and repeats k–times, where k is the number of policy rule pairs in *PotentialConflictSet* (k in worst case will be equal to n^2, but typically k will be much smaller then n). Two simple function are called in this loop. The first function compare actions, the second compares conditions of policy rules from a given pair (line 5). The time complexity of these two functions is constant ($O(1)$), because the number of comparisons does not depend on the policy size or hierarchy size (the number of possible actions and possible condition predicates are known a priori).

We can see that the time complexity of this phase is $O(k)$, and assuming the typical case of $k \ll n$, it becomes $O(n)$.

Main part of algorithm The main part of the ModCon algorithm (Alg. 3) starts with execution of the preparation phase (line 4). Next, in lines 5–7, another loop will be executed n times – so we get complexity $O(n)$. This loop calls the *ConflictDetect* procedure to perform simple binary operation (lines 11–13) and add a rule pair to the *PotentialConflictSet* when necessary (lines 14–18). The time complexity of this procedure is constant – $O(1)$. At the end of the main part, rule comparison phase starts (analyzed above).

Finally, the overall time complexity of the ModCon algorithm is linear.

5 Experimental results

Our implementation of both ad-hoc and ModCon algorithms has been developed using Java SE 6. We have also used external libraries, like jsocks – to compare IP addresses, and jGrapht – to create POG graph. Cycle detection uses shortest path [12] by Dijkstra and strongly connected components [13][14] by Tarjan.

All tests was conducted on a system/cluster consisting of Dell PowerEdge 860 nodes with one Intel Xeon Quad-core x3230 2.66GHz, 2x4MB L2 cache, 4GB RAM DDR2 800MHz with Linux OpenSuSE 10.3 (64-bit) operating system.

The policies used in the experiment can be characterized by the number of rules (n) and percent of conflicts. Since the theoretical maximum number of conflicting rule pairs in a policy is n^2 (all rule pairs), thus 1% of conflicts means

that the number of conflicting pairs is 1% from n^2 (for a policy of 200 rules this gives 400 conflicting pairs).

The results presented below demonstrate average measurement values from series of 1000 executions of both algorithms for randomly generated policies.

Fig. 3 presents results of the first experiment, with 0,1% of conflicting rules and growing number of rules. Next, Fig. 4 presents results from similar tests with 0,5% of conflicts, this time. The last experiment (Fig. 5) shows the impact of growing percentage of conflicts with a constant number of rules ($n = 500$). All the experiments confirm that the ModCon algorithm performs much better then the ad-hoc one, and offers significantly lower overhead. This allows us to use ModCon for real-time policy verification.

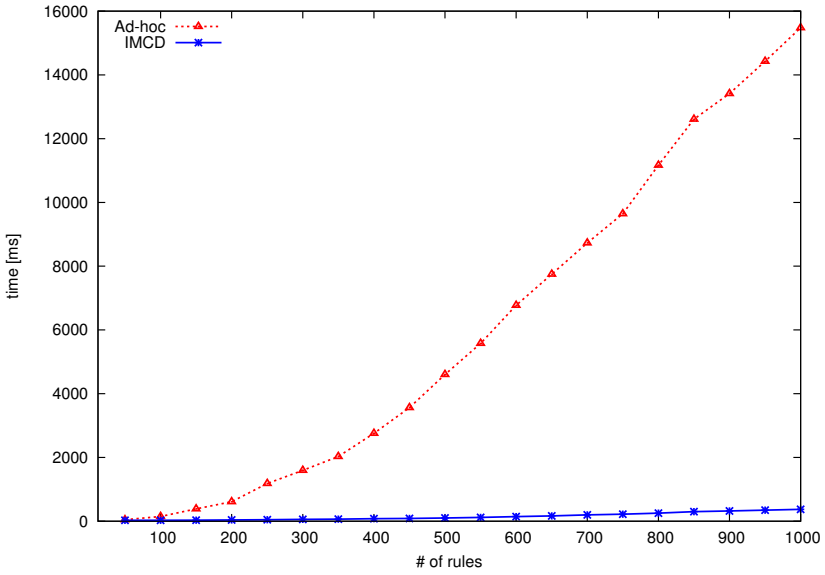


Fig. 3: Results from tests with 0,1% of conflicts

6 Conclusions

In this work, we have discussed the problem of security policy verification, namely the discovery of modality conflicts in policies for distributed systems and presented a novel solution to this problem. Compared to the existing similar approaches, the proposed algorithm is general enough to handle any policy rules and has low time complexity, enabling real-time applications. For distributed systems of more dynamic nature, like service-oriented, obligations policies and capabilities policies play a crucial role in security management. Those types of

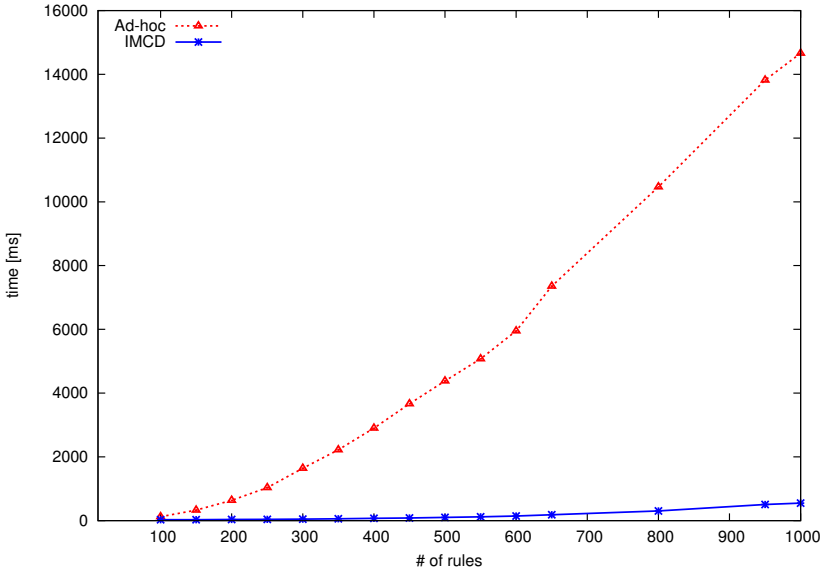


Fig. 4: Results from tests with 0,5% of conflicts

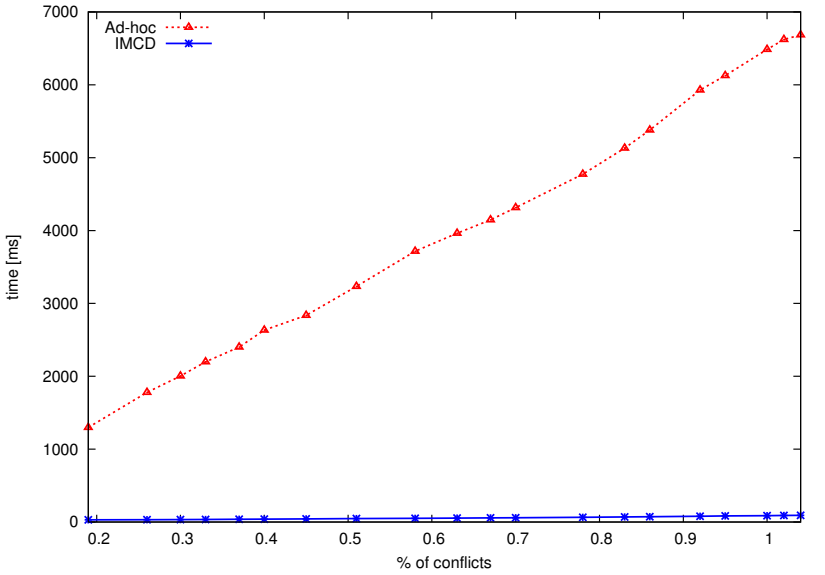


Fig. 5: Results from tests with 500 rules

policies, although being out of the scope of this paper, suffer from very similar modality conflicts. The ModCon algorithm can be easily adapted to handle also those types of conflicts.

The proposed solution has been implemented in the ORCA security policy framework [15] and we are still working on continuous improvement of the ModCon algorithm. One of possible directions is the simplification of the policy principals representation. That could additionally decrease the overhead of the preparation phase of the ModCon algorithm.

References

1. Laskey, K., McCabe, F., Brown, P., MacKenzie, M., Metz, R.: Reference model for Service Oriented Architecture. OASIS Committee Draft 1.0, OASIS Open (2006)
2. Lupu, E., Sloman, M.: Conflicts in policy-based distributed systems management. IEEE Transactions on Software Engineering **25** (1999) 852–869
3. American National Standards Institute, International Committee for Information Technology Standards: Role-Based Access Control. (2004)
4. Li, J., Karp, A.H.: Access control for the services oriented architecture. In: Proceedings of the 2007 ACM workshop on Secure web services, ACM (2007) 9–17
5. Abassi, R., Fatmi, S.G.E.: Dealing with Multi Security Policies in Communication Networks. In: 2009 Fifth International Conference on Networking and Services, IEEE (2009) 282–287
6. Al-Shaer, E., Hamed, H.: Modeling and Management of Firewall Policies. IEEE Transactions on Network and Service Management **1** (2004) 2–10
7. Baboescu, F., Varghese, G.: Fast and scalable conflict detection for packet classifiers. In: 10th IEEE International Conference on Network Protocols, 2002. Proceedings., IEEE Comput. Soc (2002) 270–279
8. Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M., Bandara, A.: A Formal Framework for Policy Analysis. Technical report, Department of Computing, Imperial College London, London (2008)
9. Damianou, N.C., Dulay, N., Lupu, E., Sloman, M.: Ponder: A language for specifying Security and Management Policies for distributed System. Technical report, Imperial College of Science, Technology and Medicine; Department of Computing, London (2000)
10. Brodecki, B., Sasak, P., Szychowiak, M.: Security policy definition framework for SOA-based systems. In G. Vossen, D. D. E. Long, J.X.Y., ed.: 10th Int. Conf. on Web Information Systems Engineering (WISE 2009). Volume 5802 of Lecture Notes in Computer Science., Poznań, Poland, Springer-Verlag (2009) 589–596
11. Brodecki, B., Brzeziński, J., Sasak, P., Szychowiak, M.: Modality conflict discovery for SOA security policies. In Olivier Temam, Pen-Chung Yew, B.Z., ed.: Advanced Parallel Processing Technologies 2011 (APPT 2011). Volume 6965 of Lecture Notes in Computer Science., Shanghai, China, Springer-Verlag (2011) 112–126
12. Dijkstra, E.W.: A Note on Two Problems in Connection with Graphs. Numerical Mathematics **1** (1959) 269–271
13. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1** (1972) 146–160
14. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf. Process. Lett. **8** (1979) 121–123

15. Brodecki, B., Szychowiak, M.: Conflict discovery algorithms used in ORCA. Technical Report TR-ITSOA-OB8-4-PR-11-03, Institute of Computing Science, Poznań University of Technology (2011)