2013-01-25

A Novel Approach to White-Box Policy Analysis

Balasubramaniam, Jayalakshmi

UNIVERSITY OF CALGARY

A Novel Approach to White-Box Policy Analysis

by

Jayalakshmi Balasubramaniam

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

January, 2013

Abstract

The access control systems in dynamic environments contain composite access control policies, that combine decisions from multiple component policies using policy combining algorithms. In such dynamic environments, analysis of policies is a challenge.

In this thesis, I propose a white-box policy analysis Decision in Context (DIC), that would analyse component policies situated inside a composite policy. For generality, the DIC query is defined in an XACML-style policy composition framework. The DIC query is implemented via a reduction to either propositional satisfiability or pseudo boolean satisfiability instances, after which standard solvers can be invoked to complete the evaluation. Empirical analyses have been conducted to compare the relative efficiency of the SAT and PBS encodings. The latter is found to be the more effective encoding, in reducing DIC queries containing majority voting policy combining algorithms.

Table of Contents

List of Tables

List of Figures and Illustrations

Chapter 1

Introduction

With the advent of the digital age, access control systems have seen widespread applications. Today, major sectors such as Healthcare, Banking, etc. need to protect sensitive information such as electronic health records (EHRs) of patients, financial data of the customer, etc., against unauthorized accesses and usages. These sectors are divided into a number of work units (a hospital has different departments— Cardiology, Neurology, Anaesthetics, Critical care, etc.; a bank may have different units— Legal, Financial Management, Marketing, etc.). Hence, each of these distributed units may have different policies governing the disclosure of data. This creates a need for a policy composition framework [38, 15, 39, 32, 13, 18] that would specify (1) allowable policy compositions, and (2) ways to combine individual authorization decisions from a set of policies. Moreover, these kind of dynamic environments have their information disclosure policies regularly updated, thus making policy analysis in such a composition framework a challenge.

1.1 Motivation

The motivation behind this thesis work is elaborated in this section.

1.1.1 Need for a White-Box Policy Analysis

An access control policy is said to be a *composite policy* if it is composed of more than one sub-policy. When such a composite policy exists for a resource, each sub-policy within this composite policy returns an authorization decision. The composite policy's final authorization decision is a combination of the individual decisions returned by its sub-policies. Combining algorithms are used to perform these combinations of individual decisions re-

turned by sub-policies within a composite policy in order to reach a final decision. The composition framework is a definition of (1) the way in which a composite policy is composed of sub-policies, and (2) the way in which the sub-policy decisions are combined in this composite policy.

It is imperative that analysis be conducted on these composite policies in such composition frameworks, to prevent the following erratic behaviour: (a) granting illegitimate accesses, and (b) denying legitimate accesses. Policy analysis in general means observing the authorization decision returned by an access control policy on receiving a specific type of access request. For example in [15, 34, 23] an access control policy is analysed to verify that the policy returns specific authorization decisions for specific type of access requests. This kind of analysis helps in verifying expected policy behaviour. Other existing policy analysis and testing techniques have been elaborated in Section 2.3.

By black-box policy analysis we mean an analysis technique in which a composite policy's global behaviour is inspected. In contrast, by white-box policy analysis we mean an analysis technique that inspects the behaviour of sub-policies in a composite policy. If a composite policy exhibits problematic behaviour, error detection by means of traditional policy analysis techniques [15, 34, 23, 25] that are mostly black-box techniques is not sufficient. They view the composite policy as a black-box during policy analysis. This kind of analysis is useful in identifying problematic behaviour, but cannot be used to identify the source of such a behaviour.

In order to identify the source of a composite policy's erratic behaviour, we need to consider the following:

(a) the working of the sub-policies within a composite policy,

(b) the structural composition of a composite policy, and

(c) the procedure followed to combine individual decisions from sub-policies in a composite policy.

In other words, the composite policy is to be viewed as a white-box rather than a black-box during policy analysis. This dictates the main difference between white-box and black-box policy analysis.

Thus, there exists a need to perform such kind of white-box policy analysis, that would consider (a), (b), and (c) listed above during analysis of a composite policy, and hence help in identifying the source of a composite policy's erratic behaviour.

1.1.2 Need for a General Policy Analysis Technique

There are different access control models available, that dictate how an access is granted/ denied. Composite policies may be present in any kind of access control model. If a policy analysis technique caters to policies in a specific kind of access model, the scope of applicability of the analysis technique is greatly reduced. Adopting such analysis techniques to access control systems implementing other kinds of access control models is problematic.

To avoid such problems and increase its scope of applicability, a policy analysis technique should be general. It should not be catered to a specific kind of access control model. This is a very important aspect to be considered when developing a new policy analysis technique.

1.1.3 Need for an Efficient Technique to Analyse Majority Voting Algorithms

An interesting class of combining algorithms are the combining algorithms based on voting schemes. What makes these algorithms interesting is the fact that they consider the number of sub-policies returning an authorization decision, when deciding the final decision of a composite policy. There are three important majority voting schemes, namely, simple majority voting, absolute majority voting, and super permit majority voting. The difference among these schemes lies in the maximum number of sub-policies that should return an authorization decision in a composite policy, for that decision to be returned as a final decision. The following problems are present in the existing work related to majority voting algorithms.

- These majority voting combining algorithms do not have the same represen-

tation as other combining algorithms in certain existing policy composition frameworks [39, 32]. These different representations makes analysing composite policies in such frameworks difficult.

- Even if the majority voting schemes have an uniform representation as the rest of the combining algorithms in the framework, they have not been handled efficiently [15].

For the above reasons, there exists a need to have a clear, uniform, and efficient representation of majority voting combining algorithms in a policy composition framework, developed to aid policy analysis.

1.2 Contributions

The following are the contributions made in this work.

(a) A new kind of policy analysis along the lines of white-box testing, has been proposed. I present the Decision in Context (*DIC*) analysis. This *DIC* analysis takes three inputs—a composite policy, a reference to a sub-policy in this composite policy, and a set of one or more authorization decisions. This query determines two things: (a) whether the sub-policy referenced is evaluated when the composite policy is evaluated, and (b) whether the decision returned by the sub-policy is among the authorization decisions specified in the input.

(b) I propose a generic policy composition framework in the style of the XACML policy language [38]. I give the abstract syntax and intuitive semantics of policies. The *DIC* analysis is formulated in terms of this policy composition framework. This policy composition language is general, as a range of access control models can be captured in our policy language. Hence, our analysis can be applied to them. Our analysis technique is not tied to a specific kind of access control model.

(c) I give a compilation algorithm that transforms an analysis query to a *propositional formula satisfiability* (SAT) instance. Since this is not an efficient transformation of a query involving composite policies that use majority voting combining algorithms, I further propose a unique approach to transform the DIC query to a *Pseudo Boolean Satisfiability* (PBS) problem instance.

(d) I implement both compilation algorithms. I compare the efficiency of these two implementations, and demonstrate that PBS is a better encoding than SAT, especially when majority voting algorithms are involved.

1.3 Overview of Thesis

This thesis is organized as follows.

In Chapter 2, I give the theoretical background behind understanding the concepts used in this thesis.

In Chapter 3, I describe the generic policy composition framework, based on which the analysis query is formulated.

In Chapter 4, I explain the new analysis technique, namely, the DIC query, and its applications.

In Chapter 5, I elucidate the compilation algorithm $DIC2SAT$, used to compile a DIC query to a SAT instance.

In Chapter 6, I describe the compilation algorithm $DIC2PBS$, used to compile a DIC query to a PBS instance.

In Chapter 7, I report the empirical study conducted to compare the efficiency of the SAT and PBS encodings of the DIC query, as well as the results of this empirical study.

Finally, in Chapter 8, I conclude with a summary of contributions made in this thesis, work related to the research conducted, and some future directions this research can take.

Chapter 2

Background

This chapter introduces the concepts related to the work done in this thesis. My work is to be applied in an access control system. In Section 2.1, I give a basic description of an access control system. My work analyses policies in an XACML-style policy composition framework. In Section 2.2, I introduce XACML, its policy hierarchy, and its policy composition framework. My work gives a technique to perform analysis on access control policies. In Section 2.3, I describe some existing techniques to perform analysis and testing on access control policies. In this work, analysis of policies is done by transforming the analysis query's satisfiability into either a *SAT* or a *PBS* instance. I describe propositional satisfiability and pseudo boolean satisfiability in Sections 2.4 and 2.5, respectively.

2.1 Access Control

In today's world, more information is stored digitally than in paper format. An *information system* is a system that stores such digital information. With the use of digital information comes the need to protect this information from unauthorized accesses. This information is protected using an *access control system*, where there are certain policies regulating who can access what in the information system.

The data that needs to be protected in an information system is collectively referred to as the *resources* (*res*) of the system. The entities that demand to access these resources are *accessors*. The type of operation that an accessor needs to perform on a resource is referred to as an *action* (read, write, etc.) on a resource. An *access request* (r) is a demand made by an accessor to access a resource in a system to perform a particular action on it. This access request can be *allowed* or *denied* by the access control system. This decision of either

allowing or denying a request is referred to as an *authorization decision*. This authorization decision is returned on receiving an access request, by consulting the policies specified for a resource.

The information that will be needed by the access control system to reach an authorization decision on an access request is referred to as the *protection state* (ps) in the access control system. An *access control policy* defines the conditions under which an authorization decision is to be reached on an access request. The satisfiability of the conditions specified is determined against the pair (ps, r), i.e. the protection state of the system and the access request. There are various access control models based on the type of protection state of the access control system. Some of the widely implemented access control models are — Access Control Matrices [31], Role Based Access Control (RBAC) [42], Attribute Based Access Control (ABAC) [50].

2.2 XACML

XACML is an access control policy language. It is used to model policies in an attribute based access control system. An Attribute Based Access Control System returns authorization decisions based on the attributes of the accessor, resource, action, or the environment from which the request is made. A protection state in such an access control system consists of the set of all attributes of all users, resources, allowable actions on resources, and the environment of the information system.

The XACML policy language is made up of three elements: *Rule*, *Policy*, *PolicySet*. The XACML elements are specified in XML. The XACML specification is given in [38]. The authorization decision returned by an element in the XACML policy language, the policy language model used in XACML, and a description of the rule and policy combining algorithms used in XACML are described in the succeeding sections.

2.2.1 Authorization Decisions - XACML

An element in the XACML policy language can return one of the four decisions: *permit*, *deny*, *not-applicable*, *indeterminate*.

1. *permit*: The return of this decision by an XACML policy language element results in the access request being granted.

2. *deny*: The return of this decision by an XACML policy language element results in the access request being denied.

3. *not-applicable*: This decision is returned by an XACML policy language element when that element is not an applicable element for the access request. The determination of an element's applicability is explained in Section 2.2.2.

4. *indeterminate*: This decision is returned by an XACML policy language element if an error occurs during an element evaluation, or when a policy combining algorithm fails to reconcile the conflicting decisions returned by component policies.

2.2.2 Policy Language Model

XACML policy language model consists of three elements: *Rule*, *Policy* and *PolicySet*. These elements form a policy hierarchy with the *Rule* at the bottom of the hierarchy and the *PolicySet* at the top of the hierarchy. A brief description of each of these elements follows.

Rule

Rule is the atomic element in the XACML policy model. The components in an XACML *Rule* are : *Target*, *Condition* and *Effect*.

(a) *Target*: This element determines the applicability of a *Rule* to an access request. The *Target* specifies a boolean expression. This boolean expression is defined on the attributes of the accessor, resource, action, or environment. The boolean expression is evaluated against $(ps,\ r)$. If the boolean expression returns *True*, then the *Rule* is applicable to the access request. Otherwise, the *Rule* is not applicable to the access request.

(b) *Condition*: This component of a *Rule* gives a boolean expression, defined on the attributes of the accessor, resource, action, or environment. The boolean expression is evaluated against $(ps,\ r)$ only if the target boolean expression returns *True*. The evaluation of this boolean expression if returning *True* would result in the *Effect* of this *Rule* being returned. If this component is not present, then the *Effect* value will be returned by the *Rule* if the boolean expression specified in its *Target* evaluates to *True*.

(c) *Effect*: Two values are allowed for this *Rule* component— *permit*, *deny*. This is the value that will be returned if both the *Target* and *Condition* of the *Rule* return *True*.

If either the *Target* or *Condition* returns *False* the *Rule* returns a *not-applicable* decision. If an error occurs during a *Rule*'s evaluation the *Rule* returns *indeterminate*.

Policy

The next element in the policy hierarchy is *Policy*. The *Policy* composes of (i) a *Target*, (ii) one or more *Rule* elements, (iii) a *Rule Combining Algorithm*, and (iv) zero or more *Obligations*. A description of each of these *Policy* components follows.

(a) *Target*: This component is similar to the *Rule* target. It is a boolean expression on the attributes of the accessor, resource, action, or environment, which if satisfied makes the *Policy* applicable to the request.

(b) *Rule Set*: This component contains at least one *Rule* element in a set. The decision

returned by the *Policy* is a combination of *Rule* effects returned by the rules in this rule set.

(c) *Rule Combining Algorithm*: This defines the procedure to be followed to combine the effects of the *Rule* elements in the rule set. The available rule combining algorithms in XACML is given in Section 2.2.3.

(d) *Obligations*: Zero or more obligations can be specified in the *Policy*. An obligation is an action that is returned along with the final decision of the *Policy*. The accessor of the resource is obliged to perform the obligation(s) after access has been granted by the *Policy*.

If the *Target* returns *False*, the *Policy* returns *not-applicable*. If an error occurs during *Policy* evaluation, the *Policy* returns *indeterminate*. The *Policy* returns an *indeterminate* decision if a combining algorithm fails to harmonize the decisions returned by the individual rules in the rule set of the *Policy*.

PolicySet

This is the topmost element in the policy hierarchy of XACML policy model. It composes of the following components: (i) *Target*, (ii) a set consisting of one or more *Policy* elements and zero or more *PolicySet* elements, (iii) a *Policy Combining Algorithm*, and (iv) zero or more *Obligations*. Each of these is explained below.

(a) *Target*: This component is similar to the target of *Rule* and *Policy*. The applicability of a *PolicySet* to a request is determined by the satisfiability of the boolean expression, that is defined on the attributes of the accessor, resource, action, or the environment.

(b) *Set*: This component is a set consisting of one or more *Policy* elements, and zero or more *PolicySet* elements. The final decision returned by an applicable *PolicySet* is a combination of decisions returned by each element in this set.

(c) *Policy Combining Agorithm*: This algorithm gives the procedure to reach the final decision, from the individual decisions returned by each element in the *Set*. An explanation of the available policy combining algorithms is given in the next section.

(d) *Obligations*: Zero or more obligations can be attached to a *PolicySet*. The accessor of the resource is required to fulfill certain obligations if access is granted by this *PolicySet*.

If the *Target* returns *False*, the *PolicySet* returns *not-applicable*. If an error occurs during the *PolicySet*'s evaluation, *indeterminate* is returned. The *PolicySet* returns an *indeterminate* decision if a combining algorithm fails to harmonize the decisions returned by the individual policies/policy sets in the *PolicySet*.

2.2.3 Policy and Rule Combining Algorithms

XACML provides three rule combining algorithms and four policy combining algorithms. The three rule combining algorithms are: *permit-overrides*, *deny-overrides*, *first-applicable*. The four policy combining algorithms are: *permit-overrides*, *deny-overrides*, *first-applicable*, *only-applicable*. The policy combining algorithms are described in Table 2.1. We adopt the XACML policy combining algorithms in our composition framework. The policy combining algorithm combines decisions from the elements in the set of the *PolicySet*. This set may consist of one or more policies or zero or more policy sets. Each element in the set returns a decision by evaluation against (ps, r), where ps is the protection state and r is the access request.

2.3 Access Control Policy Analysis and Testing

Most access control systems are dynamic, and consequently, subject to frequent change. The change to an access control system may be in the form of (a) addition of a new access control policy, (b) removal of an existing access control policy, or (c) modification of an existing access control policy. These changes may inadvertently impact the behaviour of the

Table 2.1: XACML Policy Combining Algorithms

Dec	*permit-overrides*	*deny-overrides*	*first-applicable*	*only-applicable*
permit	At least one element in the set returns *permit*.	None of the elements in the set evaluate to *deny* or *indeterminate*, and at least one of the elements returns *permit*.	The first element in the set whose *Target* evaluates to *True*, returns *permit*.	The only element in the set whose *Target* evaluates to *True*, returns *permit*.
deny	None of the elements in the set evaluate to *permit*, and, at least one of the elements evaluates to *deny*.	At least one element in the set returns *deny*.	The first element in the set whose *Target* evaluates to *True*, returns *deny*.	The only element in the set whose *Target* evaluates to *True*, returns *deny*.
indeterminate	None of the elements in the set evaluate to *permit* or *deny*, and at least one of the elements returns *indeterminate*.	None of the elements in the set evaluate to *deny*, and at least one of the elements evaluates to *indeterminate*.	If *indeterminate* is returned by an element, then, this will be the decision returned.	The only element in the set whose *Target* evaluates to *True*, returns *deny*, or, More than one element's *Target* evaluate to *True*.
not-applicable	The *Target* of every element in the set returns *False*.	The *Target* of every element in the set returns *False*.	The *Target* of every element in the set returns *False*.	The *Target* of every element in the set returns *False*.

access control system. So, the detection of this accidental impact on the access control system upon performing changes to some of its access control policies is indispensable. Moreover, a policy administrator may want to test a policy before introducing it into the access control system, to prevent any inadvertent behaviour of the access control system. In this section, I discuss some of the existing work related to the analysis and testing of access control policies for the purpose of detecting unintended effects of a change to the access control system.

(a) *Policy Similarity Analysis:* In [15] and [34], two access control policies are compared to determine if they are similar. The policy similarity analysis can be done in two ways. The first technique to detect similarity between two policies is to determine the number of access requests for which both of these policies return an identical authorization decision [15, 34]. The second technique as given in [34] is to compare the specification of either policies, and retrieve the common conditions under which these policies return an authorization decision for an access request. This kind of analysis helps in grouping together similar policies.

(b) *Property Verification Analysis:* In [15], [34], [25] and [23], access control policies are analysed to determine if they satisfy a condition/property. An access control policy's *property* can be defined in terms of the authorization decision that the policy is supposed to return to a specific kind of access request (any access request satisfying certain conditions defined on the request attributes). This kind of property analysis can be performed on a single policy or a group of access control policies. This kind of property analysis helps in detecting erroneous policy behaviour.

(c) *Change Impact Analysis:* In [23], a policy before and after change is compared. The aim is to study the change in authorization decisions that are returned by the policy before and after change. This kind of analysis helps ensure that the change in behaviour of a modified policy is as expected.

(d) *Policy Testing:* In [36], the authors have proposed a tool that would generate test cases (access request - decision pairs) to perform thorough testing on an access control policy. These test cases generated are such that they cover the testing of every *Rule* in an XACML *Policy*. They give test cases that will help perform white-box testing on the *Rule* element of XACML *Policy*. This kind of testing helps in identifying the source of a problematic policy behaviour.

2.4 Propositional Formula Satisfiability

A propositional variable is a variable that can be assigned a truth value of either *True* or *False*. A propositional formula consists of a set of these propositional variables combined using the *and*, *or*, or *not* boolean operations.

Propositional Satisfiability (SAT) is the problem of determining whether there exists a truth assignment for all propositional variables in a propositional formula such that the propositional formula evaluates to *True*. SAT is a decision problem, since we are deciding

whether the propositional formula is satisfiable or not. This problem is NP-Complete [16, 29].

A conjunction of a set of boolean formulae is two or more boolean formulae combined using the *and* ($\wedge$) boolean operation. A disjunction of a set of boolean formulae is two or more boolean formulae combined using the *or* ($\vee$) boolean operation. A literal is a propositional variable, or the negation of this propositional variable. Let x be a propositional variable, then $\neg x$ is a negative literal and x is a positive literal. A disjunction of a set of boolean literals is defined as a *clause*. A propositional formula is in Conjunctive Normal Form (*CNF*) if the formula is a conjunction of clauses. All propositional formulae can be converted to an equivalent *CNF*. An efficient transformation from a propositional formula to *CNF* is given in [17, page 999].

An example for a satisfiable *SAT* instance is given in Example 2.4.1, and an example of an unsatisfiable *SAT* instance is given in Example 2.4.2.

Example 2.4.1. Let us consider a *SAT* instance as follows. Here, x_1, x_2 and x_3 are propositional variables.

$\phi = x_1 \vee (x_2 \wedge \neg x_3)$

An equivalent CNF of the above formula is as follows.

$\phi' = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3)$

Now, the satisfiability of this formula depends on assigning a *True* value to literal x_1. Since, assigning $x_1 = True$, makes both clauses in the *CNF* satisfiable, this propositional formula is satisfiable.

Example 2.4.2. Let us consider a *SAT* instance as follows. Here, x_1, x_2, and x_3 are propositional variables.

$\phi = x_1 \wedge (x_2 \wedge x_3 \wedge \neg x_1)$

An equivalent CNF of the above formula is as follows.

$\phi' = x_1 \wedge x_2 \wedge x_3 \wedge \neg x_1$

The above *CNF* is unsatisfiable, since we need both the positive and negative literals— x_1

and $\neg x_1$ to be assigned a truth value *True*. Since such an assignment is impossible, the propositional formula is unsatisfiable.

SAT was the first known NP-Complete problem [16, 29]. The research on developing techniques to efficiently solve *SAT* is very active, and has been ongoing for the past 50 years. There are two types of methods to determine the satisfiability of a propositional formula. They can be complete methods or incomplete methods. The complete methods are systematic, and are guaranteed to return whether or not an instance is satisfiable. The incomplete techniques are not guaranteed to solve the *SAT* problem instance. The Davis–Putnam–Logemann–Loveland (DPLL) algorithm [20] is a complete method to determine the satisfiability of a *SAT* instance. The incomplete methods involve local search techniques to solve *SAT* [44, 24].

SAT solvers are available in plenty today. These SAT solvers return whether a *SAT* instance is satisfiable. Some of the widely used SAT solvers are MiniSat [4, 21], SAT4J Core [7], zChaff [2], GRASP [3], and Siege [9].

2.5 Pseudo Boolean Constraint Satisfaction

A *pseudo boolean variable* is a variable that can be assigned either one of the integer values 0 or 1. The pseudo boolean value 0 is comparable to boolean *False*, and the pseudo boolean value 1 is comparable to boolean *True*. Hence, the type of the variable that can only take either of these integer values is "pseudo boolean".

A *pseudo boolean constraint* has one of the following general forms.

$$a_1y_1 + a_2y_2 + a_3y_3 + \cdots + a_ny_n \geq b \tag{2.1}$$

$$a_1y_1 + a_2y_2 + a_3y_3 + \cdots + a_ny_n \leq b \tag{2.2}$$

$$a_1y_1 + a_2y_2 + a_3y_3 + \cdots + a_ny_n = b. \tag{2.3}$$

In (2.1), (2.2) and (2.3), every a_i and b are integer constants, and every y_i is a pseudo boolean

variable, where $1 \leq i \leq n$. A pseudo boolean constraint can either be an equation such as (2.3), or an inequality, such as (2.1) and (2.2). The above pseudo boolean constraints are linear constraints. A *pseudo boolean constraint set* consists of one or more pseudo boolean constraints (linear and/or non-linear constraints) on a set of pseudo boolean variables. A *solution* to such a pseudo boolean constraint set is an assignment of values 0 or 1 to the pseudo boolean variables in this set such that all constraints in this set are satisfied.

The *Pseudo Boolean Satisfiability* (*PBS*) problem is the problem of determining whether a solution to the given pseudo boolean constraint set exists, such that all constraints in the set are satisfied. This problem is also a decision problem. The *PBS* is an NP-Complete problem [29]. Examples of satisfiable and unsatisfiable *PBS* instances are given in Example 2.5.1 and Example 2.5.2 respectively.

Example 2.5.1. Let us consider a set of 2 pseudo boolean constraints. Here, y_1, y_2, y_3 and y_4 are pseudo boolean variables.

$$\begin{aligned} y_1 + y_2 + 2y_3 &\geq 4 \\ y_1 + y_4 &\leq 1 \end{aligned} \tag{2.4}$$

These two constraints are satisfied by the 0–1 assignment $y_1 = 1, y_2 = 1, y_3 = 1, y_4 = 0$. Hence, this *PBS* instance is satisfiable.

Example 2.5.2. Let us consider a set of 4 pseudo boolean constraints. Here, y_1, y_2, y_3 and y_4 are pseudo boolean variables.

$$\begin{aligned} y_1 + 2y_2 &\leq 2 \\ y_1 + y_3 &\geq 2 \\ y_2 + y_4 &\geq 1 \\ y_3 + y_4 &= 1 \end{aligned} \tag{2.5}$$

These 4 constraints cannot all be satisfied by any 0–1 assignment to the pseudo boolean variables y_1, y_2, y_3 and y_4. Hence, this *PBS* instance is unsatisfiable.

There are some techniques that can be applied to a PBS instance to determine its satisfiability. One such technique is to make use of a backtrack algorithm similar to the DPLL algorithm [20], to examine if a satisfiable 0–1 assignment exists for all pseudo boolean variables in the *PBS*. Another technique is to convert the *PBS* into *SAT* and then determine its satisfiability. Some local search techniques, similar to ones used in *SAT* can also be used on *PBS* [48]. Integer Linear Programming(ILP) solving techniques have also been efficiently used to solve *PBS* instances, though these techniques are more general and not specific to the 0,1 domain of pseudo boolean constrains. A *PBS* solver thus solves a *PBS* instance by using the technique best optimized to its application. The exploration of *PBS* is not as mature as that of *SAT*.

Some of the known PBS solvers are Pueblo [45], MiniSat+ [5, 22], PBS v2.1 [6], SAT4J Pseudo [8], Borg [1], and Bsolo [35]. These solvers offer efficient means to solve *PBS* instances.

Chapter 3

An XACML-Style Policy Composition Framework

In this chapter, I define an XACML-style policy composition framework. This framework describes the abstract syntax and semantics of my policy language. The reason for creating such a framework is to provide a concrete basis for articulating the details of the *DIC* query.

3.1 Comparison with the XACML Framework

XACML is a widely used access control policy language. It provides means for specifying individual access control policies, and a number of policy combining algorithms to compose access decisions in composite access control policies. I base the design of my policy composition framework on XACML in order to demonstrate the applicability of *DIC* analysis to realistic policy languages. Yet, my framework is more abstract and uniform when compared to XACML. This is to simplify the design of *DIC* analysis. The following highlights the features of XACML that my framework attempts to capture.

Firstly, the core specification document of XACML v2.0 lists a set of four rule/policy combining algorithms [38, page 12]: (1) deny-overrides (do), (2) permit-overrides (po), (3) first-applicable (fa), and (4) only-applicable (oa). All the above are used to compose policies in our policy language. In addition, we also deal with a special class of combining algorithms known as the majority voting algorithms. Such algorithms have not been sufficiently dealt with in previous work related to access control policy analysis [15, 34, 39, 32]. These algorithms will be explained in details later in this chapter.

Secondly, the *Rule* structure of XACML consists of three main parts - *target*, *condition* and *effect*. A *target* is a single or a set of conditions on the attributes of the entities of the access control system and determines the applicability of a *rule* to an access request. A

condition is a single or a set of conditions on the attributes of the entities of the access control system. If the *condition* is satisfied the *Rule* would return the *effect* as the access decision. An *effect* can be either a *permit* or *deny* decision. A *policy* in XACML consists of a *target* and a collection of these *rules* whose decisions are combined using a *rule combining algorithm*. A *policy set* in XACML is a combination of *policies* whose access decisions are combined using a *policy combining algorithm*. I adopt a simplified hierarchy in our framework, and do not differentiate between *rules* and *policies*.

Thirdly, an XACML policy can give one of the four decisions — permit, deny, not applicable or indeterminate. The "permit" decision means the access is granted, and the "deny" decision means that the access is denied. The "not applicable" decision is returned by an XACML policy on a request if its *target* condition is not satisfied by the request. An "indeterminate" decision will be returned in two cases:

(a) an error occurs during the evaluation of a policy;

(b) a policy combining algorithm fails to harmonize the contradicting decisions of the component policies.

Case (b) will be further discussed later in the chapter. In our framework, the access control policies can return one of the four access decisions in the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$ which are similar to "permit", "deny", "not applicable" and "indeterminate" decisions of XACML, respectively. We assume in our framework that an error during a policy evaluation cannot occur. That is, case (a) does not apply, and only case (b) is possible.

Fourthly, XACML supports dynamic retrieval of policies at the time of evaluation of their evaluation (i.e., it is possible to refer to some policies, policy sets, or rules at a different host, at the time of specification of a policy or policy set in XACML). In our work, we assume that all policies are statically available at the time of their evaluation.

Finally, XACML supports associating a set of obligations (a set of actions to be performed by the accessor if the access request is granted) with a policy or a policy set. We do not deal

with obligations in this thesis.

3.2 Access Control Model

In our policy composition framework we assume an underlying generic access control model. I explain about a generic access control model in this section. Any information system has a set of resources, a set of users accessing these resources, and a set of access types allowed on the resources. An access control model is defined to protect these set of resources.

A request r is made by a user to access a resource in a particular way. A request can in general be viewed as a tuple (u, res, a), where u is the user who wants to perform action a on resource res. A request predicate rp is a condition on one of the following— the user wanting to access a resource, the resource being accessed, the type of access that the user needs on the resource being accessed. An access control model has a protection state— ps, at any given time. The protection state holds information regarding the resources, users, and allowable access types. A request predicate rp is evaluated against the current protection state ps and a request r, and a boolean value of either *True* or *False* is returned as the result of evaluation. Every state-request pair (ps, r) induces a truth assignment to the request predicates, such that, a request predicate is assigned the truth value 1 if it is satisfied in that (ps, r) and 0 otherwise.

An access control system of the above general form can be **uniform** or **non-uniform**. By an *uniform access control system*, we mean that every possible truth assignment of request predicates in the access control system has at least one protection state and request pair (ps, r), in which the request predicates with their corresponding truth assignment can be realized. By a *non-uniform access control system*, we mean that there exists one or more truth assignments for request predicates in the access control system that does not have an associated protection state and request pair (ps, r) in which the request predicates with that truth assignment is realized.

The generic model described above can accommodate different concrete access control systems, as can be seen in the following examples.

Example 3.2.1. *Attribute Based Access Control System*

In this type of access control, access is granted based on the attributes of the resources and users in the information system. Here *ps* constitutes the following— attributes of users, attributes of resources, attributes of the environment from which access is made, and the different access types allowed.

Let us consider an example policy and its evaluation in an Attribute Based Access Control system. We consider a university precious resource usage policy.

pol: Students in their second year of graduate study or higher are granted access to precious resources of the university.

Let rp_1, rp_2, rp_3 be three request predicates.

rp_1: res.type = precious

rp_2: user.role = graduate student

rp_3: user.year-of-study ≥ 2

Now *pol* can be defined as a XACML *Rule* as follows.

$Target = rp_1$

$Condition = rp_2 \wedge rp_3$

$Effect = permit$

Let r_1 and r_2 be two access requests received from two graduate students A and B, where, A is in the first year of study and B is in the third year of graduate study.

r_1: (u=A, res=printer, a=print)

r_2: (u=B, res=printer, a=print)

The request predicates rp_1, rp_2, and rp_3 will be evaluated against a (ps, r) pair, where

ps is the protection state and r is the access request. Since A is a graduate student in his first year of graduate study evaluating rp_3 against (ps, r_1) will return *False*, while the other two request predicates return *True*. So, when the XACML *Rule* representation of *pol* is evaluated against (ps, r_1), a *not applicable* decision will be returned. This is because the *Condition* of the *Rule* does not return *True*.

Similarly, the request predicates rp_1, rp_2, rp_3 will be evaluated against (ps, r_2) to determine their satisfiability. All of these request predicates evaluate to *True*. So, when the XACML *Rule* representation of *pol* is evaluated against (ps, r_2), a *permit* decision will be returned. This is because both *Condition* and *Target* of the *Rule* return *True*.

Example 3.2.2. *Role Based Access Control System*

In this type of access control system, access decisions are made based on the *role* of the user requesting the resource in the system. The protection state ps in this model constitutes the following— user-role assignment, role-permission assignment, role hierarchy.

Let us consider an example file access policy in an organization.

pol: Managers have write access on file X.

Let rp_1, rp_2 and rp_3 be three request predicates.

rp_1: user.role = Manager

rp_2: action = write

rp_3: res = file X

Now *pol* can be defined as a XACML *Rule* as follows.

$Target = rp_2 \wedge rp_3$

$Condition = rp_1$

Effect = permit

Let r_1 and r_2 be two requests received from two users of the system A and B, where A

is in a managerial position and B is in a clerical position.

r_1: (u=A, res= file X, a=write)

r_2: (u=B, res= file X, a=write)

The request predicates rp_1, rp_2, and rp_3 will be evaluated against (ps, r_1) pair, where ps is the protection state. All of these request predicates evaluate to *True*. So, when the XACML *Rule* representation of *pol* is evaluated against (ps, r_1), a *permit* decision will be returned. This is because both *Condition* and *Target* of the *Rule* return *True*.

The request predicates rp_1, rp_2, and rp_3 will be evaluated against (ps, r_2) pair, where ps is the protection state. Since B is in a clerical position the evaluation of rp_1 will return *False*, while the evaluation of the other two request predicates return *True*. So, when the XACML *Rule* representation of *pol* is evaluated against (ps, r_2), a *not applicable* decision will be returned. This is because the *Condition* of the *Rule* does not return *True*.

Example 3.2.3. *Facebook-style Access Control System*

In this type of access control system access is granted based on the existence of a relationship type between the accessor and resource owner. The protection state ps is the current social network that providing information on the users, resources, the owner relationship between every resource and its owner (user), and the relationship between users.

Let us consider an example Facebook policy governing the download of photos by one user from another user.

pol: A user A is allowed to download a photo from a user B, if A is $B's$ *friend*.

Let rp_1 be a request predicate.

rp_1: The requester and the owner are friends.

Let r_1 and r_2 be requests received from users D and E to download a photo owned by F, where D is a friend of F and E is not a friend of F in the current social graph.

r_1: (u=D, res=photo, a=download)

r_2: (u=E, res=photo, a=download)

Figure 3.1: Abstract Syntax of the XACML-style Policy Composition Framework

$\mathcal{POLICY}$	::=	`p`	(Permit Policy)
	|	`d`	(Deny Policy)
	|	$\mathcal{COND} \longrightarrow \mathcal{POLICY}$	(Conditional Policy)
	|	$\mathcal{PCA}(\mathcal{POLICY}^{+})$	(PCA Policy)
$\mathcal{PCA}$	::=	`po`	(Permit-Overrides)
	|	`do`	(Deny-Overrides)
	|	`fa`	(First-Applicable)
	|	`oa`	(Only-Applicable)
	|	`smv`	(Simple Majority Voting)
	|	`amv`	(Absolute Majority Voting)
	|	`spmv`	(Super Permit Majority Voting)
$\mathcal{COND}$	::=	$\top$	(*True* condition)
	|	$\bot$	(*False* condition)
	|	$\mathcal{RP}$	(Request Predicate)

Let ps be the current protection state of the system. The request predicate rp_1 returns *True* and *False* when evaluated against (ps, r_1) and (ps, r_2) respectively.

3.3 Policy Composition Framework

In this section, I describe the abstract syntax and semantics of my policy composition framework.

3.3.1 Syntax

The abstract syntax of the policy composition framework is given by Figure 3.1 in BNF format.

A $\mathcal{POLICY}$ in our framework can be one of the four types. We refer to `p` and `d` as *atomic* policies and the other two as *composite* policies since they contain other policies as their components. The policy that is evaluated when a condition is satisfied for a Conditional Policy is referred to as its *component* policy. The policies in the policy set of a PCA Policy

Table 3.1: Semantics of $\mathcal{POLICY}$

$\mathcal{POLICY}$	**Name**	**Description**
p	Permit Policy	This policy always returns $\boldsymbol{p}$ (i.e., "permit").
d	Deny Policy	This policy always returns $\boldsymbol{d}$ (i.e., "deny").
$\mathcal{COND} \longrightarrow \mathcal{POLICY}$	Conditional Policy	A Conditional Policy *con*→*pol* returns the decision returned by the *pol* (an instance of type $\mathcal{POLICY}$) if *con* (an instance of type $\mathcal{COND}$) specified is evaluated to *True* in (*ps, r*), otherwise $\boldsymbol{n}$ is returned.
$\mathcal{PCA}(\mathcal{POLICY}^{+})$	PCA Policy	A PCA Policy of the form $pca(pol_1, pol_2, \ldots, pol_k)$ combines the decisions returned by $pol_1, pol_2, \ldots, pol_k$ using the policy combining algorithm *pca*.

are also referred to as its *component* policies. Any policy that appears in an composite policy, including the composite policy itself is referred to as a *sub-policy* of the composite policy. Policies that are structured according to the abstract syntax provided in our framework can be effectively analysed using the *DIC* query.

3.3.2 Semantics

A policy *pol* (i.e., an instance of type $\mathcal{POLICY}$) is evaluated against a pair (*ps, r*), where *ps* is the protection state of the access control system, and r is the request for which an authorization decision is to be returned. The result of evaluation is one of the four decisions in the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Similarly, a condition *con* (i.e., an instance of type $\mathcal{COND}$) is also evaluated against a pair (*ps, r*). The result of evaluation is a boolean value (i.e., either *True* or *False*).

Also, a composite PCA Policy is evaluated in a recursive fashion, with each policy in the policy set evaluated in the order that they appear, depending on the policy combining algorithm used in the PCA Policy.

The semantics of policies and conditions are summarized in Table 3.1 and 3.2 respectively.

From the Table 3.1, we refer to the four different policy types as Permit Policy, Deny Policy, PCA Policy and a Conditional Policy. This would be the form of reference for these

Table 3.2: Semantics of $\mathcal{COND}$

$\mathcal{COND}$	**Description**
$\top$	This is the true condition. This condition is satisfied in any (*ps, r*) pair, where *ps* is the protection state and r is the request predicate.
$\bot$	This is the false condition. This condition is not satisfied in any (*ps, r*) pair, where *ps* is the protection state and r is the request predicate.
$\mathcal{RP}$	This is a request predicate (rp) that is defined on any of the entities in the current protection state of the access control system. It is evaluated against a (*ps, r*) pair, where *ps* is the protection state and r is the request predicate. It returns either the boolean value $True$ or $False$ depending on whether it is satisfied or not satisfied, respectively.

policy types hereafter in this thesis.

Conditional Policy

The Conditional Policy's condition has the semantics given in Table 3.2. This is the equivalent of XACML's *target* and *condition* fields of an XACML rule which is the basic component of the XACML policy hierarchy. The condition in the conditional policy is evaluated against (*ps, r*). This abstract condition is what makes our policy language general, and, applicable to the different types of access control models.

The applicability of a policy to a (*ps, r*) pair (where *ps* is the protection state and r is the request predicate) is determined by the satisfiability of the condition in a Conditional Policy. If the condition is not satisfiable, then the Conditional Policy is not applicable to the (*ps, r*) pair, and it returns an $\boldsymbol{n}$ decision.

PCA Policy

The next type of composite policy in our framework is the PCA Policy that composes the access decisions from each of its component policies using the *pca* (an instance of type $\mathcal{PCA}$) specified. The *pca* defines a procedure for combining the decisions from the policy set in a PCA Policy. These policy combining algorithms can be subdivided into two types: standard XACML combining algorithms (`po, do, oa, and fa`), and Majority Voting Schemes

Table 3.3: Semantics of $\mathcal{PCA}$- Standard XACML Combining Algorithms

Dec	`po`	`do`	`fa`	`oa`
p	At least one policy in the policy set returns ***p***.	None of the policies in the policy set returns either ***d*** or ***i*** and at least one policy returns ***p***.	The first policy in the policy set that does not return an ***n*** returns a ***p***.	Only one policy in the policy set returns ***p*** while all other policies return ***n***.
d	None of the policies returns ***p*** and at least one policy returns ***d***.	At least one policy in the policy set returns ***d***.	The first policy in the policy set that does not return an ***n*** returns a ***d***.	Only one policy in the policy set returns ***d*** while all other policies return ***n***.
i	None of the policies returns either ***p*** or ***d*** and at least one policy returns ***i***.	None of the policies in the policy set returns ***d*** and at least one policy returns ***i***.	The first policy in the policy set that does not return an ***n*** returns a ***i***.	Only one policy in the policy set returns *i* and all other policies return ***n***, **OR**, More than one policy in the policy set returns a decision other than ***n***.
n	None of the policies in the policy set returns ***p***, ***d*** or ***i***.	None of the policies in the policy set returns ***p***, ***d*** or ***i***.	None of the policies in the policy set returns ***p***, ***d*** or ***i***.	None of the policies in the policy set returns ***p***, ***d*** or ***i***.

`smv, amv, and spmv`.

The semantics of the standard XACML combining algorithms is in Table 3.3. The table describes the conditions under which a PCA Policy returns a decision on receiving an access request when its component policies are combined using each of the standard XACML policy combining algorithms.

The majority voting schemes are described below.

Simple Majority Voting: In this voting algorithm, ***p*** or ***d*** will be returned based on which of these two decisions are being returned by a strict majority of policies in the policy set. An ***i*** decision will be returned in the following cases:

(a) if ***p*** and ***d*** are not returned by any of the policies in the policy set, and if at least one of the policies in the policy set return ***i***;

(b) if the number of policies returning ***p*** is equal to the number of policies returning ***d***.

If none of the policies in the policy set return a ***p***, ***d***, or ***i***, then ***n*** will be returned.

Absolute Majority Voting: In this voting algorithm, ***p*** or ***d*** is returned if the number of

policies returning either of these decisions is strictly greater than one-half of the number of policies in the policy set. An ***i*** decision will be returned in the following cases:

(a) if the number of policies returning ***p*** or ***d*** is not strictly greater than one-half of the total number of policies in the policy set;

(b) if none of the policies in the policy set returns ***p*** or ***d*** and at least one of them return ***i***.

If none of the policies in the policy set return a ***p***, ***d*** or ***i***, then ***n*** will be returned.

Super Permit Majority Voting: In this voting scheme, ***p*** is returned only if the number of policies returning ***p*** is strictly greater than two-thirds of the total number of policies in the policy set. A ***d*** is returned if the number of policies returning ***p*** is not strictly greater than two-thirds of the total number of policies in the policy set and at least one of the policies in the policy set return ***d***. An ***i*** decision will be returned in the following cases:

(a) if the number of policies returning ***p*** is not strictly greater than two-thirds of the total number of policies in the policy set, and there is no policy in the policy set returning ***d***;

(b) if ***p*** and ***d*** are not returned by any of the policies in the policy set, and at least one of these policies returns ***i***.

If none of the policies in the policy set return a ***p***, ***d*** or ***i***, then, ***n*** will be returned.

The semantics of these majority voting schemes is specified formally in Table 3.4, which depends on the following definition. Let $\{pol_1, pol_2, \ldots, pol_k\}$, where $k \geq 0$, be the policy set to be combined. Also, each policy pol_j in the policy set is associated with four variables p_j, d_j, n_j, i_j, such that (a) each of these variables can take the value 1 or 0, and (b) $p_j + d_j + i_j + n_j = 1$. If pol_j returns a permit decision then $p_j = 1$, and $d_j = i_j = n_j = 0$.

Operationally, the PCAs will evaluate the sub-policies in the policy set one by one. The algorithm `po` will stop evaluating sub-policies as soon as one of them is evaluated to a ***p*** decision. Similarly, `do` stops when a ***d*** is encountered, and `fa` stops when a decision other

Table 3.4: Semantics of $\mathcal{PCA}$- Majority Voting Schemes

Dec	*smv*	*amv*	*spmv*
p	$\Sigma_{j=1}^{k} p_j > \Sigma_{j=1}^{k} d_j$	$\Sigma_{j=1}^{k} p_j \geq \lfloor k/2 \rfloor + 1$	$\Sigma_{j=1}^{k} p_j \geq \lfloor 2k/3 \rfloor + 1$
d	$\Sigma_{j=1}^{k} d_j > \Sigma_{j=1}^{k} p_j$	$\Sigma_{j=1}^{k} d_j \geq \lfloor k/2 \rfloor + 1$	$\Sigma_{j=1}^{k} p_j \ngeq \lfloor 2k/3 \rfloor + 1$ $\Sigma_{j=1}^{k} d_j \geq 1$
n	$\Sigma_{j=1}^{k} n_j = k$	$\Sigma_{j=1}^{k} n_j = k$	$\Sigma_{j=1}^{k} n_j = k$
i	$\Sigma_{j=1}^{k} p_j = \Sigma_{j=1}^{k} d_j$ $\Sigma_{j=1}^{k} (p_j + d_j + i_j) \geq 1$	$\Sigma_{j=1}^{k} p_j \ngeq \lfloor k/2 \rfloor + 1$ $\Sigma_{j=1}^{k} d_j \ngeq \lfloor k/2 \rfloor + 1$ $\Sigma_{j=1}^{k} (p_j + d_j + i_j) \geq 1$	$\Sigma_{j=1}^{k} p_j \ngeq \lfloor 2k/3 \rfloor + 1$ $\Sigma_{j=1}^{k} d_j = 0$ $\Sigma_{j=1}^{k} (p_j + i_j) \geq 1$

than ***n*** is encountered. Other PCAs (`oa, smv, amv, spmv`) will always evaluate every sub-policy in the policy set.

3.3.3 Derived Forms of Authorization Decisions

The following are the derived forms of authorization decisions as policies in our composition framework. These policies return the specific decisions on every possible (*ps, r*) evaluation, where *ps* is a protection state and *r* is an access request. A direct application of these derived forms is in complex policy pruning, given in Section 4.4.4.

(a) ***p*** : `p`

(b) ***d*** : `d`

(c) ***i*** : `oa (p, d)`

(d) ***n*** : $\perp \rightarrow$ `p`

3.3.4 Examples

Consider two syntactically valid policies in our framework, as given below. These two example policies are evaluated against the same (*ps, r*), where *ps* is the current protection state and *r* is an access request. The result of evaluating request predicates rp_1 and rp_2 against (*ps, r*) is given in Table 3.5.

Table 3.5: Evaluation Results of Request Predicates rp_1 and rp_2 against (ps, r)

Request Predicate	*Evaluation Result*
rp_1	*True*
rp_2	*False*

Table 3.6: Decision returned by each Composite Sub-policy in Example 3.3.1

Policy ID	**Sub-policy**	**Dec**
1	`spmv` (`fa` (`p`, `d`) , $rp_1 \rightarrow$ `d` , `d`)	***d***
2	`fa` (`p`, `d`)	***p***
3	$rp_1 \rightarrow$ `d`	***d***

Example 3.3.1. A PCA Policy,

`spmv` (`fa` (`p`, `d`) , $rp_1 \rightarrow$ `d` , `d`).

The decision returned by every composite sub-policy in this composite policy is given in Table 3.6 in a top-down fashion. I will refer to each of these composite policies in the description below using the number given in the **Policy ID** column for the respective composite policy.

This PCA Policy is composed of three component policies. The individual decisions of these policies are combined using the policy combining algorithm `spmv`. In this combining algorithm, all policies in the policy set will be evaluated to determine the number of policies retuning each decision. In the following paragraphs I explain the semantics of each of the component sub-policies in the order of their evaluation within this policy set.

Sub-policy **2** is a PCA Policy combined using `fa`. This combining algorithm returns the decision from the first policy that does not return an ***n***. In the policy set of **2** this policy is `p`, that always returns a ***p*** decision. Hence, the decision returned by **2** is ***p***.

The composite policy **3** is a Conditional Policy. Since the condition rp_1 is satisfied in by (ps, r), **3**'s component policy will be evaluated and its decision returned. This component policy is `d` that always returns a ***d***. So, the decision returned by **3** is ***d***.

The next component policy of 3.3.1 is an atomic policy `d`, hence, the decision returned

Table 3.7: Decision returned by each Composite Sub-policy in Example 3.3.2

Policy ID	Sub-Policy	Dec
1	$rp_1 \rightarrow \texttt{po}\big(\texttt{d}, rp_2 \rightarrow \texttt{p}, \texttt{oa}(rp_2 \rightarrow \texttt{d}, \texttt{d})\big)$	***d***
2	$\texttt{po}\big(\texttt{d}, rp_2 \rightarrow \texttt{p}, \texttt{oa}(rp_2 \rightarrow \texttt{d}, \texttt{d})\big)$	***d***
3	$rp_2 \rightarrow \texttt{p}$	***n***
4	$\texttt{oa}(rp_2 \rightarrow \texttt{d}, \texttt{d})$	***d***
5	$rp_2 \rightarrow \texttt{d}$	***n***

by this component policy will be ***d***.

The `spmv` returns a ***d*** if the number of policies returning a ***p*** is not strictly greater than two-thirds of the total policies in the policy set and at least one policy in the set return a ***d***. Since the policies in the policy set of **1** satisfy this condition, the combined decision returned by **1** is ***d***.

Example 3.3.2. A Conditional Policy,

$rp_1 \rightarrow \texttt{po}\big(\texttt{d}, rp_2 \rightarrow \texttt{p}, \texttt{oa}(rp_2 \rightarrow \texttt{p}, \texttt{d})\big)$

This policy is a composite Conditional Policy. The decision returned by every composite sub-policy of 3.3.2 is given in Table 3.7 in a top-down fashion. I will refer to each of these composite sub-policies in the description below using the number given in the **Policy ID** column for the respective composite policy.

The composite policy **1** returns the decision returned by its component policy **2**, since the condition rp_1 is satisfied in $(ps,\ r)$.

The policy **2** is a PCA Policy composed of four component policies whose decisions are combined using the `pca`, `po`. In this algorithm, the policies in the policy set will be evaluated in order, till a ***p*** is returned; else, all the component policies will be evaluated.

The first component policy of **2** is an atomic policy `d`. This component returns a ***d*** decision. The next policy in the policy set will be evaluated since this is not a ***p***.

The second component policy is **3**. This is a Conditional Policy, in which the condition rp_2 is not satisfied in $(ps,\ r)$. Hence, **3** returns an ***n***. The next policy in the policy set will

be evaluated since this is not a ***p***.

The third component policy of **2** is a PCA Policy composed of two component policies, whose decisions are combined using `oa`. This algorithm returns the decision of the only policy in the policy set to return either a ***p***, ***d*** or ***i***. If more than one policy returns a ***p***, ***d*** or ***i*** decision, ***i*** is returned. So, all policies in the policy set will be evaluated to check for this condition when this algorithm is used.

The first component policy of **4** is **5**. This is a Conditional Policy, whose condition rp_2, is not satisfied in (*ps, r*). Hence **5** returns a ***n***. The second component policy of **4** is a Deny Policy `d`. This component returns a ***d***. Of the two policies in the policy set only one policy returns a ***d***, whereas the other returns a ***n***. Hence, the decision returned by **4** is ***d***.

Since the policies in the policy set of **2** have not returned a ***p***, the algorithm `po` checks the decisions returned by each policy in the policy set to see if at least one policy returns a ***d***. Since this is true, the ***d*** is returned as the final decision of **2** and hence as the final decision of **1**.

Chapter 4

The *DIC* Query

In this chapter I introduce the *DIC* query, which is short for *Decision in Context* query. I also illustrate the need for defining the *DIC* query to perform policy analysis. The *DIC* query has many applications, some of which are explained in this chapter. Some derived forms of the *DIC* query are also described in this chapter.

4.1 Need for the *DIC* Query

The policy analysis techniques given in previous work in this area aim to perform analysis on an access control policy by examining its output for different input cases. The following are important concepts to consider during policy analysis.

- The access control policy being analysed can be a sub-policy of another composite policy. In this case, the analysed policy may behave differently when analysed as part of a composite policy than when being analysed on its own.
- The policy composition framework used to combine decisions from individual policies may influence the behaviour of a policy being analysed. Hence, the policy composition framework also needs to be taken into account when analysing a policy in a composition framework.
- An analysis technique that can analyse both composite policies and any sub-policies that it may contain is needed to perform rigorous policy analysis.

The *DIC* query aims to incorporate all of the above factors. It provides techniques to analyse a sub-policy contained within a composite policy in the context of evaluation of the composite policy. It takes into account the policy composition framework being used

in a composite policy, to determine whether the sub-policy being analysed is evaluated at all. It provides techniques for analysing composite policies as a whole and sub-policies contained within a composite policy, in the context of evaluation of the composite policy. The description of a *DIC* query is given in the following section. This would illustrate how each of these concepts have been integrated in our query.

4.2 Requirements for Usage of the *DIC* Query

The *DIC* query provides a means for analysing sub-policies within a composite policy. In order to facilitate this analysis, we need a technique to distinctly identify each sub-policy within a composite policy (including itself). So, we assume that in our framework, each sub-policy within a composite policy is uniquely identified by a *label*. One of the inputs of the *DIC* query is such a label.

Example 4.2.1. Consider a syntactically valid policy in our policy composition framework.
$rp_1 \rightarrow \texttt{fa}\Big(rp_2 \rightarrow \texttt{p}, rp_3 \rightarrow \texttt{d}, rp_4 \rightarrow \texttt{po}\big(rp_3 \rightarrow \texttt{p}, rp_5 \rightarrow \texttt{d}\big)\Big)$
The Abstract Syntax Tree of this policy along with its labels is given in Figure 4.1. Each node in the tree represents a sub-policy, and each of these sub-policies has a numeric label, given at the bottom right hand corner of the node.

Let LABEL(*pol*) be a function, such that it takes an instance of $\mathcal{POLICY}$ and returns the label of that policy in the abstract syntax tree.

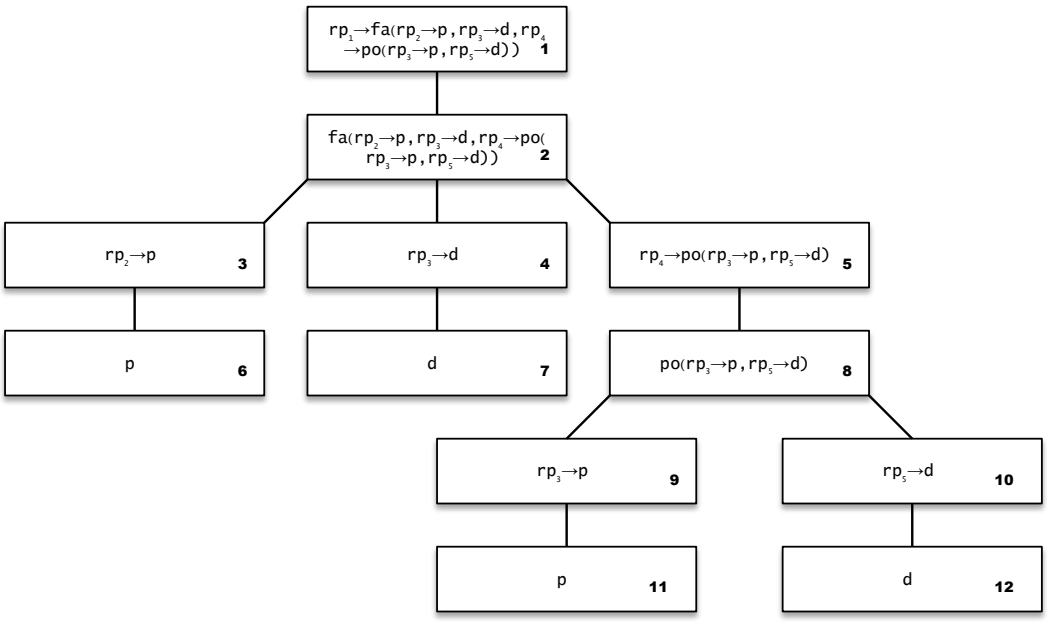
For example, if *pol* is the policy given in Example 4.2.1, then LABEL(*pol*) returns **1**, which is the label of *pol* as given in Figure 4.1.

4.3 Decision In Context Query

The *DIC* query has the following general form,

$$DIC(pol,\ com,\ ds)$$

Figure 4.1: An Abstract Syntax Tree for Example 4.2.1



The *DIC* query takes three inputs and returns a single output. The query's inputs and output are described below.

Input:

1. *pol* An access control policy which is an instance of $\mathcal{POLICY}$.
2. *com* The label of a sub-policy of *pol*.
3. *ds* A non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Output: The *DIC* query returns one of the following outputs.

- *Yes* — This output value is returned if there exists a protection state *ps* and a request r such that, when *pol* is evaluated against (*ps,* r), the sub-policy identified by label *com* is recursively evaluated, and this evaluation of the sub-policy returns a member of *ds*.
- *No* — This output value is returned if, for every protection state *ps* and request r, when *pol* is evaluated against (*ps,* r), either the sub-policy identified by label

Figure 4.2: Query Language with Boolean combinations

$$
\begin{array}{lcll}
\psi, \psi' & ::= & q & q \text{ is an atomic } DIC \text{ query} \\
 & | & \psi \wedge \psi' & \text{Conjunction} \\
 & | & \psi \vee \psi' & \text{Disjunction} \\
 & | & \neg\psi & \text{Negation}
\end{array}
$$

com is not evaluated, or the evaluation of this sub-policy does not return a member of *ds*.

A *DIC* query is said to be **satisfied** by a variable assignment (a truth assignment to request predicates appearing in the query), iff the evaluation of *pol* against this variable assignment results in (a) the recursive evaluation of the sub-policy of *pol* identified by *com*, and (b) this recursive evaluation returns a member of *ds*. A *DIC* query is **satisfied** by a state-request pair (*ps, r*) iff it is satisfied by the variable assignment realized in that (*ps, r*). A *DIC* query is **satisfiable** if it is satisfied by some state-request pair (*ps, r*). A *DIC* query is **valid** if it is satisfied by every state-request pair.

4.3.1 Well-Formed *DIC* query

The *DIC* query is called a *well-formed DIC* query if the following condition on the inputs *com* and *pol* holds.

$$com \sqsubseteq \textsc{label}(pol)$$

A *DIC* is a well-formed *DIC* query if *com* is a label of a sub-policy of *pol*. This condition ensures that the *DIC* query operates on valid inputs.

4.3.2 Boolean Combinations on *DIC* query

Figure 4.2 gives the *DIC* Query Language that allows boolean combinations on the *DIC* queries.

A conjunction (disjunction) of two *DIC* queries q_1 and q_2 is said to be **satisfied** by a variable assignment iff the assignment satisfies both (either) q_1 and (or) q_2. The negation of a *DIC* query q is **satisfied** by a variable assignment iff the assignment doesn't satisfy q.

A conjunction (disjunction) of two *DIC* queries q_1 and q_2 is said to be **satisfied** by a state-request pair (*ps, r*) iff the variable assignment realized in that (*ps, r*) is satisfied by both (either) q_1 and (or) q_2. The negation of a *DIC* query q is **satisfied** by a state-request pair (*ps, r*) iff the variable assignment realized in that (*ps, r*) doesn't satisfy q.

A conjunction (disjunction) of two *DIC* queries q_1 and q_2 is **satisfiable** if some state-request pair (*ps, r*) satisfies both (either) q_1 and (or) q_2. The negation of a *DIC* query q is **satisfiable** if some state-request pair (*ps, r*) doesn't satisfy q.

A conjunction (disjunction) of two *DIC* queries q_1 and q_2 is **valid** if every state-request pair satisfies both (either) q_1 and (or) q_2. The negation of a *DIC* query q is **valid** if every state-request pair (*ps, r*) doesn't satisfy q.

4.3.3 Examples

In this sub-section I give some examples to demonstrate the behaviour of the *DIC* query on various inputs. Table 4.1 provides a column each for each input: *pol*, *com*, *ds*, and also a column for output from the *DIC* query. Square brackets ([ ]) surround every sub-policy of *pol* in each example specified in the Table. The label of each sub-policy in *pol* is given in the bottom right hand corner outside the square brackets.

From the set of examples given in Table 4.1, one can deduce that the *DIC* query returns a *Yes* or *No* output taking into consideration the following: (a) the decision returned by *pol* in the context of the composite policy of which it is part of, and (b) the policy composition framework that is used to combine decisions if *pol* is placed in a composite PCA Policy. Also evident from the above examples is that the *DIC* query provides techniques to analyse both a composite policy and any sub-policies it may contain.

Table 4.1: Behaviour of *DIC* Query on Sample Inputs

No.	*pol*	*com*	*ds*	O/P
1	$[rp_1 \rightarrow [\texttt{oa}([\texttt{d}]_3, [rp_2 \rightarrow [\texttt{p}]_5]_4)]_2]_1$	1	$\{\boldsymbol{i}\}$	*Yes*
2	$[rp_1 \rightarrow [\texttt{oa}([\texttt{d}]_3, [rp_2 \rightarrow [\texttt{p}]_5]_4)]_2]_1$	1	$\{\boldsymbol{d}\}$	*Yes*
3	$[rp_1 \rightarrow [\texttt{oa}([\texttt{d}]_3, [rp_2 \rightarrow [\texttt{p}]_5]_4)]_2]_1$	1	$\{\boldsymbol{n}\}$	*Yes*
4	$[rp_1 \rightarrow [\texttt{oa}([\texttt{d}]_3, [rp_2 \rightarrow [\texttt{p}]_5]_4)]_2]_1$	1	$\{\boldsymbol{p}\}$	*No*
5	$[\texttt{po}([\texttt{d}]_2, [rp_2 \rightarrow [\texttt{p}]_4]_3)]_1$	1	$\{\boldsymbol{i}\}$	*No*
6	$[\texttt{po}([\texttt{d}]_2, [\texttt{p}]_3)]_1$	2	$\{\boldsymbol{d}\}$	*Yes*
7	$[\texttt{po}([\texttt{p}]_2, [\texttt{d}]_3)]_1$	3	$\{\boldsymbol{d}\}$	*No*
8	$[\texttt{smv}([rp_1 \rightarrow [\texttt{d}]_6]_2, [rp_2 \rightarrow [\texttt{p}]_7]_3, [rp_3 \rightarrow [\texttt{d}]_8]_4, [rp_1 \rightarrow [\texttt{p}]_9]_5)]_1$	1	$\{\boldsymbol{p}\}$	*Yes*
9	$[\texttt{smv}([rp_1 \rightarrow [\texttt{d}]_6]_2, [rp_2 \rightarrow [\texttt{p}]_7]_3, [rp_3 \rightarrow [\texttt{d}]_8]_4, [rp_1 \rightarrow [\texttt{p}]_9]_5)]_1$	1	$\{\boldsymbol{i}\}$	*Yes*
10	$[\texttt{smv}([rp_1 \rightarrow [\texttt{d}]_6]_2, [rp_2 \rightarrow [\texttt{p}]_7]_3, [rp_3 \rightarrow [\texttt{d}]_8]_4, [rp_1 \rightarrow [\texttt{p}]_9]_5)]_1$	3	$\{\boldsymbol{n}\}$	*Yes*

4.4 Applications

The policy analysis that we have described above using the *DIC* query has a wide range of applications. In this section I explain the usage of *DIC* query in change impact analysis, break-glass reduction analysis, dead policy identification and complex policy pruning.

4.4.1 Change Impact Analysis

Access control systems are dynamic, and hence prone to frequent change. When policy administrators add new policies to the system, or remove or modify existing policies in the system, there are bound to be side effects (unanticipated changes) on the other access control policies in the system. *DIC* queries can be formulated to facilitate the effective identification of these side effects.

For ease of explanation of this application I refer to the access control policy to be analysed for change impact as pol_{old}, before a change has been made to it. After the change, the policy is referred to as pol_{new}. The label of this policy or a sub-policy that it contains, before and after change is different. In the following description *ps* is the protection state and *r* is the access request. *pol* is evaluated against a (*ps, r*) pair.

A policy analyst wishing to perform a change impact analysis on a modified composite policy should proceed in the following manner.

(a) For a given protection state-request pair ($ps,\ r$), a sub-policy l of pol is said to be *applicable* to l if the evaluation of pol against this ($ps,\ r$) pair results in the recursive evaluation of l. The policy analyst begins by determining if the policy revision results in a change of applicability of a sub-policy l of pol.

If there has been an increase (or decrease) in the number of protection state-request pairs in which sub-policy l is applicable, then it is called an *expansion* (or *contraction*) *of applicability*. By *expansion* (or *contraction*), we mean an increase (or decrease) in the number of variable assignments (truth assignment to the request predicates in the *DIC* query) in which the sub-policy l is applicable in pol_{new}. To be more precise, by expansion we mean that the set of state-request pairs for which l is applicable in pol_{old} is a proper subset of the set of state-request pairs for which l is applicable in pol_{new}. The *contraction* of applicability can be described in a similar way.

A *DIC* query can be formulated to confirm an expectation of a policy analyst, regarding a sub-policy's expansion or contraction of applicability after change to pol.

Let q^{app}_{old} and q^{app}_{new} be two *DIC* queries analysing pol_{old} and pol_{new} respectively. Let $pol^{'}$ be the sub-policy of pol we are interested in analysing. pol' is labeled com_{old} and com_{new} in pol_{old} and pol_{new} respectively.

$$q^{app}_{old} = DIC(pol_{old}, com_{old}, \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{n},\boldsymbol{i}\})$$
$$q^{app}_{new} = DIC(pol_{new}, com_{new}, \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{n},\boldsymbol{i}\})$$

There is no contraction of applicability iff (4.1) is valid. Expansion of applicability can then be confirmed by checking that the query (4.2) is satisfiable. Similarly, there is no expansion of applicability iff (4.3) is valid. Contraction of applicability can then be confirmed by checking that the query (4.4) is satisfiable.

$$\neg q_{old}^{app} \vee q_{new}^{app} \tag{4.1}$$

$$q_{new}^{app} \wedge \neg q_{old}^{app} \tag{4.2}$$

$$\neg q_{new}^{app} \vee q_{old}^{app} \tag{4.3}$$

$$q_{old}^{app} \wedge \neg q_{new}^{app} \tag{4.4}$$

(b) Once a policy analyst determines whether the applicability of a sub-policy has changed, he may wish to determine whether the decision returned by that sub-policy has changed between the old and new versions of the composite policy *pol*.

Let q_1 and q_2 be two *DIC* queries analysing pol_{old} and pol_{new} respectively. Let pol' be the sub-policy we are interested in analysing. pol' is labeled com_{old} and com_{new} in pol_{old} and pol_{new} respectively.

$q_1 = DIC(pol_{old}, com_{old}, \{dec\})$ where $dec \sqsubset \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$

$q_2 = DIC(pol_{new}, com_{new}, \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\} \setminus \{dec\})$

The query (4.5) will return *Yes* iff the decision retuned by the sub-policy changes for a (*ps, r*) pair between the modified and unmodified versions of the composite policy.

$$q_1 \wedge q_2 \tag{4.5}$$

Thus by issuing the query (4.5), it is possible to determine a change in the behaviour of sub-policy pol'.

4.4.2 Break-Glass Reduction Analysis

In [10], Ardagna et al. have introduced the concept of using policy spaces to reduce the number of emergency accesses (referred to as break-glass accesses), made to healthcare information systems to access patient health records.

A policy space can be described as a strict exclusive subset of policies in an access control system, that protects the Electronic Health Records (EHRs) of patients. The authors describe four such policy spaces in an access control system— positive authorization policy space ρ^+, negative authorization policy space ρ^-, planned exception policy space ε^P, and unplanned exception policy space ε^U.

- ρ^+ is the policy space that consists of the subset of policies granting access under normal circumstances.
- ρ^- consists of those policies denying access under normal circumstances.
- ε^P contains policies that grant access under certain exceptional circumstances, which under normal circumstances will not be granted.
- ε^U is a policy space containing a single policy that grants access to all access requests, that do not satisfy the conditions stipulated in the other three policy spaces. This policy grants access with certain obligations such as logging and manual auditing.

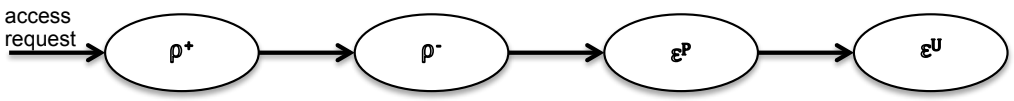
Individual decisions from policies in each policy space are combined using the `fa` combining algorithm.

An example of these policy spaces in our policy composition framework, and the global policy *pol* in such an access control system is given below.

$$
\begin{aligned}
pol &= \texttt{fa}(\rho^+, \rho^-, \varepsilon^P, \varepsilon^U) \\
\rho^+ &= \texttt{fa}(rp_1 \rightarrow \texttt{p},\ rp_2 \rightarrow \texttt{p},\ rp_3 \rightarrow \texttt{p}) \\
\rho^- &= \texttt{fa}(rp_4 \rightarrow \texttt{d},\ rp_5 \rightarrow \texttt{d},\ rp_6 \rightarrow \texttt{d}) \\
\varepsilon^P &= \texttt{fa}(rp_7 \rightarrow \texttt{p},\ rp_8 \rightarrow \texttt{p},\ rp_9 \rightarrow \texttt{p}) \\
\varepsilon^U &= \texttt{p}
\end{aligned}
$$

Here request predicates rp_1, rp_2, and rp_3 are conditions under which an access is granted. The request predicates rp_4, rp_5, and rp_6 are conditions under which an access is denied.

Figure 4.3: Policy Evaluation Flow adapted from [10]



Request predicates rp_7, rp_8, and rp_9 are exceptional conditions under which an access is granted. If an access request does not satisfy any of these request predicates, access will be granted by the Permit Policy in the policy space ε^U.

The evaluation of access requests occurs in the following way. When an access request is received, each policy (in the order they are specified within the policy space) in each policy space (in the order in which they are specified above) will be evaluated. The purpose is to find the first policy that does not return an $\boldsymbol{n}$ decision against this access request in the current protection state ps of the system. Once such a policy is identified, the decision returned by the evaluation of this policy will be returned as the authorization decision of the request.

The access control system in this case consists of a single composite policy, made up of the four policy spaces as its component policies. The decisions from these policy spaces are combined using the `fa` combining algorithm.

These policy spaces are prioritized in the order of their specification. Any incoming access request is evaluated against each policy space, till an applicable policy for that request is found as given in the Figure 4.3. The authors propose to reduce the occurrence of exceptions (requests to which access is granted by the latter two policy spaces) in the following ways: (1) by moving frequent applicable policies from ε^P to the policy spaces ρ^+ and ρ^-, and (2) adding new policies in ρ^+ and ρ^- that will capture the frequent conditions under which requests are granted from ε^U (by reviewing access logs).

Let us consider the problem of identifying whether there has been a change in the policy space that grants/denies access to an access request, after techniques to reduce break-glass occurrences have been introduced. More specifically, a policy analyst may be interested in

Figure 4.4: Matrix Specification of the Movement problem

	ρ^+_{new}	ρ^-_{new}	ε^P_{new}	ε^U_{new}
ρ^+_{old}			×	×
ρ^-_{old}			×	×
ε^P_{old}	✓	✓		
ε^U_{old}	✓	✓		

analysing whether there exists an access request (a) that was granted/denied access by policy space A before change, and (b) it is granted/denied access by policy space B after change. This is referred to as the *movement* problem.

A policy analyst can be interested in identifying a set of expected movements and a set of unexpected movements. Consider the matrix given in Figure 4.4. In this matrix " × " indicates the type of movement that is not expected, and "✓" indicates the type of movement that is expected. In this matrix, policy spaces with subscript *old* represent the original policy spaces before a change has been made. Policy spaces with subscript *new* represent policy spaces after a change has been made.

For example, there is a " × " mark on the row ρ^-_{old} and column ε^P_{new}. This unexpected movement is from ρ^-_{old} to ε^P_{new}. The movement problem here is in determining whether there exists a (*ps, r*) pair (*ps* is the protection state and r is the access request) that was denied access by ρ^-_{old} before change, and is granted access by ε^P_{new} after change.

Given such a matrix it is possible to construct a set of *DIC* queries, to identify whether a set of expected movements has occurred, and to ensure that a set of unexpected movements has not occurred.

Let *pol* be a composite PCA Policy as follows.

$pol =$ `fa` (ρ^+, ρ^-, ε^P , ε^U)

Let com^+, com^-, com^U and com^P point to the policy spaces ρ^+, ρ^-, ε^P and ε^U respectively,

both before and after change to *pol*.

Let pol_{old} be the composite policy before a change to reduce break-glass attempts has been made. Let pol_{new} be the composite policy after such a change has been made.

For the matrix in Figure 4.4 the corresponding set of *DIC* queries are given below.

$$DIC(pol_{old}, com^{+}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{P}, \{\boldsymbol{p}\}) \tag{4.6a}$$

$$DIC(pol_{old}, com^{+}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{U}, \{\boldsymbol{p}\}) \tag{4.6b}$$

$$DIC(pol_{old}, com^{-}, \{\boldsymbol{d}\}) \wedge DIC(pol_{new}, com^{P}, \{\boldsymbol{p}\}) \tag{4.6c}$$

$$DIC(pol_{old}, com^{-}, \{\boldsymbol{d}\}) \wedge DIC(pol_{new}, com^{U}, \{\boldsymbol{p}\}) \tag{4.6d}$$

$$DIC(pol_{old}, com^{P}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{+}, \{\boldsymbol{p}\}) \tag{4.7a}$$

$$DIC(pol_{old}, com^{P}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{-}, \{\boldsymbol{d}\}) \tag{4.7b}$$

$$DIC(pol_{old}, com^{U}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{+}, \{\boldsymbol{p}\}) \tag{4.7c}$$

$$DIC(pol_{old}, com^{U}, \{\boldsymbol{p}\}) \wedge DIC(pol_{new}, com^{-}, \{\boldsymbol{d}\}) \tag{4.7d}$$

The first set of *DIC* queries (4.6) indicate whether the unexpected movements specified have occurred. If all of these queries (4.6) return *No*, then none of the unexpected movements has taken place.

Similarly, if all of the second set of *DIC* queries (4.7) return *Yes* then all of the expected movements specified in the matrix have taken place. Thus, given a movement specification matrix with expected and unexpected movements specified, it is possible to determine if the specification is satisfied or not using two sets of *DIC* queries (4.6) (4.7) as specified.

For larger access control systems a tool that would take as input a movement specification matrix and return *Yes* if the set of expected movements has occurred and the set of unexpected movements has not occurred, can be built. It determines the output by encoding the matrix as sets of *DIC* queries.

These movement problems are important in identifying the changes in the composite policies handling an access request, in an access control system. They help in identifying the correct and incorrect changes in the policies handling any request. This is an important kind of policy analysis.

We have shown how to effectively use our *DIC* query to detect the reduction in break-glass occurrences in such an access control system.

4.4.3 Dead Policy Identification

Another application of the *DIC* query is in the identification of dead policies. By dead policies (much the same like dead code) we refer to those sub-policies in a composite policy that are never evaluated. If a sub-policy identified by *com* is a dead policy, then the following query returns *No*.

$$DIC\ (pol,\ com,\ \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\})$$

Once a dead policy has been identified, the reason behind its existence can be analysed by the policy administrator of the access control system. It is possible that a dead policy exists simply as a side effect of modification of a composite policy.

For example, the dead policy may be one of the policies in the policy set of a PCA Policy, where the policy decisions are combined using the `po` algorithm. The presence of a `p` (Permit Policy) before a sub-policy causes this sub-policy to not be evaluated at all. This is because evaluation of policies in the policy set will be stopped once a $\boldsymbol{p}$ decision is returned by one of the policies in the policy set. Hence, this sub-policy is a dead policy, and its presence is not necessary in the policy set of this PCA Policy.

For a big policy, the search for dead policies can be performed in a top-down manner. More specifically, if we are to represent policies in our policy composition framework as an Abstract Syntax Tree as given in Example 4.2.1, we can perform the pre-order traversal of the nodes in the AST, and perform dead policy analysis in the order the nodes are visited.

4.4.4 Complex Policy Pruning

Another useful application of the *DIC* query is in determining parts of the composite policy that can be pruned, and replace them by simpler policies. Composite sub-policies that return the same decision for all (*ps, r*) pairs (*ps* is the protection state and *r* is the access request) can be pruned by replacing the composite policy by the respective derived form of the decision, as given in Section 3.3.3 in Chapter 3.

The *DIC* query *q*'s satisfiability confirms the existence of a (*ps, r*) pair (*ps* is the protection state and *r* is the access request) where the *DIC* query returns *Yes*.

Given a *DIC* query

$$q = DIC(pol,\ com,\ \{dec\}),\ \text{where}\ dec \in \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\},$$

the complement query q^c is defined to be the *DIC* query

$$q^c = DIC(pol,\ com,\ \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\} \setminus \{dec\}).$$

Thus intuitively the complement of a *DIC* query *q* checks if there exists a (*ps, r*) in which the evaluation of *pol* results in a recursive evaluation of the sub-policy identified by *com*, and this evaluation returns a decision which is not *dec*. If the complement query q^c returns *No*, then in all possible (*ps, r*) pairs either (a) the sub-policy labeled *com* is not evaluated, or (b) it evaluates to *dec*. Please note that finding the complement of a *DIC* query *q* whose decision set is $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$ gives an invalid *DIC* query whose decision set is an empty set.

Example 4.4.1. Consider the query *q* as given below. Let *pol* be a composite access control policy that is being analysed. Let *com* identify a sub-policy of *pol* that has a complex structure.

$$q = DIC(pol,\ com,\ \{\boldsymbol{p}\ \})$$

The complement of this *DIC* query q^c as given below.

$$q^c = DIC(pol,\ com,\ \{\boldsymbol{d},\ \boldsymbol{i},\ \boldsymbol{n}\ \})$$

If q^c returns *No*, then the sub-policy identified by *com* in *pol* can be replaced by a `p` (Permit Policy).

The check for pruning complex policies can also be performed in a top-down manner, specifically in the order the nodes are visited in the pre-order traversal of the AST of a composite policy. This application of our *DIC* query is comparable to the policy tree pruning described in [18].

4.5 Complexity of *DIC* Query

The *DIC* query satisfiability can be defined as a decision problem, **DIC-SAT** problem.

DIC-SAT Problem: Given a well-formed, atomic *DIC* query *DIC* (*pol*, *com*, *ds*) of a uniform access control system, is the *DIC* query satisfiable?

This **DIC-SAT** problem is NP-Complete. For a decision problem to be NP-Complete, it needs to belong to the complexity class NP, and it needs to be NP-Hard.

The **DIC-SAT** algorithm is in NP, as it is possible to guess a satisfying truth assignment, and then evaluate *pol* to check that the *DIC* query is satisfied by the assignment.

The NP-Hardness of the **DIC-SAT** problem can be demonstrated by a reduction from an NP-Hard decision problem. A reduction from the NP-Hard problem monotone one-in-three 3SAT [43] is given below.

Every clause in a monotone one-in-three 3SAT is a disjunction of three positive literals. The monotone one-in-three 3SAT decision problem is the problem of deciding whether there exists a truth assignment that satisfies exactly one variable in each of its clauses. This problem is a known NP-Complete problem [43].

Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be an instance of monotone one-in-three 3SAT, where each C_i is of the form $x_1^i \vee x_2^i \vee x_3^i$. Let q_ϕ be a *DIC* query of the form *DIC* (*pol*, LABEL(*pol*), $\{\boldsymbol{p}\}$), where *pol* is defined as follows:

$$\texttt{amv}(pol_1, pol_2, pol_3, \cdots, pol_k, pol_{k+1}, pol_{k+2}, \cdots, pol_{2k-1}),$$

where every $pol_j = \texttt{d}$ for $k+1 \leq j \leq 2k-1$.

Each pol_i for $1 \leq i \leq k$ can be defined as follows (where i denotes the i^{th} clause of ϕ),

$$\texttt{oa(}x_1^i \rightarrow \texttt{p}, x_2^i \rightarrow \texttt{p}, x_3^i \rightarrow \texttt{p)}\texttt{.}$$

From the above reduction we see that q_ϕ is satisfiable (as a *DIC* query) iff there is a truth assignment that satisfies exactly one variable in each clause of ϕ.

4.6 $\mathcal{DEC}$ Query

The *DIC* query has a simpler form form that can be used to customize what we would like to learn about a policy being analysed. This query is the $\mathcal{DEC}$ query, which is the decision query. This query can be used to perform simple analysis on the policies in an access control system.

The general form of this query is given below.

$$\mathcal{DEC}\ (pol,\ ds)$$

This query takes as input two values,

- *pol* which is an instance of syntactic category $\mathcal{POLICY}$, and
- *ds* which is a non-empty subset of the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

It outputs one of the following.

- *Yes* is returned if there exists a protection state *ps* and a request r such that, when *pol* is evaluated against (*ps,* r), the evaluation returns a member of *ds*.
- *No*, if there exists no such (*ps,* r) pair.

This query is useful in performing black-box policy analysis, where the policy is analysed to determine if it can return an expected decision. The placement of the policy in a composite

policy is ignored in this analysis. This is similar to the type of black-box analysis being done in [15, 34].

The $\mathcal{DEC}$ query can be reduced to a *DIC* query as follows.

$$\mathcal{DEC}(pol,\ ds) \triangleq DIC\ (pol,\ \textsc{label}(pol),\ ds) \tag{4.8}$$

The reduction from the $\mathcal{DEC}$ query to the *DIC* query involves setting *com* to be the label of *pol* in the *DIC* query. Thus, the *DIC* query in (4.8) will return a *Yes* output iff the reduced $\mathcal{DEC}$ query returns *Yes*.

Chapter 5

Compilation of *DIC* Query: *SAT* Encoding

The satisfiability of a *DIC* query can be reduced to the satisfiability of a propositional formula. In other words, a *DIC* query can be encoded as an instance of the *SAT* problem. I explain such an encoding in this chapter. The encoding converts the conditions under which the *DIC* query returns a *Yes* output into a boolean formula, such that the satisfiability of this boolean formula coincides with the satisfiability of the *DIC* query.

5.1 The *DIC*2*SAT* Compilation Algorithm

The compilation of the *DIC* query using the *SAT* encoding is given by the *DIC*2*SAT* algorithm. The pseudocode of this *DIC*2*SAT* algorithm is given in Appendix A. A description of the *DIC*2*SAT* algorithm's precondition, inputs and output follows.

Input *DIC*2*SAT* takes three inputs.

- *pol*: An access control policy which is an instance of the syntactic category $\mathcal{POLICY}$.
- *com*: The label of a sub-policy of *pol*.
- *ds*: A decision set which is a non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Precondition Inputs *com* and *pol* should be such that the following condition holds.

$$com \sqsubseteq \textsc{label}(pol)$$

The condition given is satisfied if and only if *com* is the label of a sub-policy of *pol*.

Output *DIC*2*SAT* returns a propositional formula ϕ such that the query *DIC*(*pol*, *com*, *ds*) is satisfiable iff ϕ is satisfiable.

5.1.1 Assumptions

We make the following assumptions to support the transformation of the *DIC* query to a *SAT* instance using the *DIC*2*SAT* algorithm.

(a) We assume that all PCA Policies have a policy set of size at least 2. This is a reasonable assumption since the case where the PCA Policy consists of a single policy in its policy set is uninteresting. This is because the evaluation of such a PCA Policy always returns the decision returned by this single policy in the policy set, irrespective of the combining algorithm used.

(b) We also assume that the underlying access control system is **uniform** as explained in Section 3.2.

5.2 Overall Structure of *DIC*2*SAT*

The *DIC*2*SAT* algorithm is a *syntax directed recursive* algorithm. The pattern of recursion closely follows the inductive pattern of the grammar rules. The abstract syntax of our policy language is given in Figure 3.1. The algorithm performs case analysis to handle each syntactic variant in the grammar. The skeleton of the pseudocode of the *DIC*2*SAT* algorithm illustrating this behaviour is given in Figure 5.1.

The algorithm handles two base cases and eight inductive cases. All of these are explained below.

- Base Cases: The syntactic variants `p` and `d` of type $\mathcal{POLICY}$ are the base cases. The algorithm returns a simple propositional formula in either case.

- Inductive Cases: The syntactic variants representing Conditional Policy and the seven different PCA Policies constitute the eight inductive cases of the algorithm. In these cases, the algorithm is recursively applied to the sub-policies. The propositional formula returned by the recursive calls are combined, to

produce the propositional formula returned by the algorithm for an inductive case.

Each syntactic case has sub-cases. These sub-cases are generated by the following two criteria.

1. The first criterion is whether *com* is the label of *pol*, or the label of a proper sub-policy of *pol*.

2. The second criterion is whether each of the decision values $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$ belongs to *ds*.

The two criteria can be combined to generate a maximum of $2 \times 4 = 8$ sub-cases in total. Furthermore, based on the input sometimes multiple sub-cases may apply. One example of such an instance is when *ds* contains multiple decision values.

In each main case a disjunction of propositional formulae will be built and returned by the *DIC2SAT* algorithm. Here, each applicable sub-case may contribute a single disjunct to this disjunction. In addition, the pseudocode in the appendix is organized such that each main case is handled by a specialized subroutine.

5.3 A Description of the Behaviour of *DIC2SAT* on Certain Input Cases

For a better understanding of the *DIC2SAT* algorithm I outline here the behaviour of *DIC2SAT* for some input cases.

Case 5.3.1. $pol = \mathtt{p}$, $com = \textsc{label}(pol)$, *ds*

In this case, the encoding returned by *DIC2SAT* depends only on *ds*. The label *com* can only refer to p itself since the latter is an atomic policy. The encoding returned by *DIC2SAT* in this case is given below.

Figure 5.1: Skeleton of the PseudoCode of $DIC2SAT(pol, com, ds)$

(a) Skeleton for all *pol* types

```
1.  l := LABEL(pol)
2.  switch (pol) {
3.      case p:
4.          if (p ∈ ds) then
5.              ...
6.          else
7.              ...
8.      case d:
9.          if (d ∈ ds) then
10.             ...
11.         else
12.             ...
13.     case con → pol′:
14.         if l =com then
15.             ...
16.         else /* com ⊏ l */
17.             ...
18.         if (n ∈ ds) and (l = com) then
19.             ...
20.     case po (pol1, pol2, ..., polk):
21.         ...
22.     case do (pol1, pol2, ..., polk):
23.         ...
24.     case fa (pol1, pol2, ..., polk):
25.         ...
26.     case oa (pol1, pol2, ..., polk):
27.         ...
28.     case smv (pol1, pol2, ..., polk):
29.         ...
30.     case amv (pol1, pol2, ..., polk):
31.         ...
32.     case spmv (pol1, pol2, ..., polk):
33.         ...
    }
```

(b) Skeleton of *pol* type PcaPolicy

```
1.  case pca (pol1, pol2, ..., polk):
2.      if l = com then
3.          if (p ∈ ds) then
4.              ...
5.          if (d ∈ ds) then
6.              ...
7.          if (i ∈ ds) then
8.              ...
9.          if (n ∈ ds) then
10.             ...
11.     else /* com ⊏ l */
12.         ...
```

$$\phi = \begin{cases} \top & \text{if } \boldsymbol{p} \in ds \\ \\ \bot & \text{if } \boldsymbol{p} \notin ds \end{cases}$$

The satisfiability of propositional formula ϕ returned by $DIC2SAT$ reflects the satisfiability of DIC on the same inputs. This is because the evaluation of pol can return a member of ds only if $\boldsymbol{p}$ is a member of ds, which is what the $DIC2SAT$ encodes in this case.

Case 5.3.2. $pol = rp \rightarrow pol^{'}, com, ds$

Let rp be the boolean variable that encodes the truth value of the satisfiability of rp. The propositional formula returned by $DIC2SAT$ in this case for every possible input condition is illustrated below.

$$\phi = \begin{cases} rp \wedge DIC2SAT(pol^{'}, \textsc{label}(pol^{'}), ds) & \text{if } com = \textsc{label}(pol) \text{ and } \boldsymbol{n} \notin ds \\ \\ rp \wedge DIC2SAT(pol^{'}, com, ds) & \text{if } com \sqsubset \textsc{label}(pol) \\ \\ \Big(rp \wedge DIC2SAT(pol^{'}, com, ds)\Big) \vee \neg rp & \text{if } com = \textsc{label}(pol) \text{ and } \boldsymbol{n} \in ds \end{cases}$$

The rationale behind each of the cases for ϕ is described below.

1. In case 1 since com is the label of pol and $\boldsymbol{n} \notin ds$, the encoding generated encodes the following conditions — the condition rp is satisfied, sub-policy pol' is evaluated, and this evaluation results in a member of ds being returned.

2. In case 2 since com is the label of one of the descendent sub-policies of pol, the encoding generated will encode the following conditions — the condition rp is satisfied, sub-policy pol' is evaluated and this evaluation results in the sub-policy labeled com being evaluated, and the latter evaluation returns a member of ds.

3. In case 3 com is the label of pol and $\boldsymbol{n} \in ds$. The Conditional Policy pol can return a $\boldsymbol{n}$ decision under two circumstances— the condition rp is not satisfied

or, *rp* is satisfied and the sub-policy *pol'* returns $\boldsymbol{n}$ on its evaluation. Thus, these two conditions are encoded in this case.

Case 5.3.3. $pol = \texttt{fa}\ (pol_1, pol_2, \ldots, pol_k),\ com,\ ds$

I further divide this case into the following two sub-cases, based on *com* value. This is because the encoding returned by $DIC2SAT$ differs based on *com*.

Sub-case 5.3.3.1. $com = \textsc{label}(pol)$

This is the sub-case where *com* is the label of the given *pol*. The propositional formula returned by $DIC2SAT$ in this sub-case for every possible input condition is illustrated below.

$$\phi = \begin{cases} \bigvee_{j=1}^{k} F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{p}\}) & \text{if } \boldsymbol{p} \in ds \\ \bigvee_{j=1}^{k} F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{d}\}) & \text{if } \boldsymbol{d} \in ds \\ \bigvee_{j=1}^{k} F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{i}\}) & \text{if } \boldsymbol{i} \in ds \\ \neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\}) & \text{if } \boldsymbol{n} \in ds \end{cases}$$

where,

$$F\text{-}app(pol_1, pol_2, \ldots, pol_j) = \left(\bigwedge_{i=1}^{j-1} DEC2SAT(pol_i, \{\boldsymbol{n}\})\right) \wedge DEC2SAT(pol_j, ds)$$

$$OneOf((pol_1, pol_2, \ldots, pol_k), ds) =$$

$$DEC2SAT(pol_1, ds) \vee DEC2SAT(pol_2, ds) \vee \cdots \vee DEC2SAT(pol_k, ds)$$

Here, the $DEC2SAT$ encoding can be arrived at by using the $DIC2SAT$ using the following reduction from $DEC2SAT$ to $DIC2SAT$.

$$DEC2SAT(pol,\ ds) \triangleq DIC2SAT(pol,\ \textsc{label}(pol),\ ds)$$

The propositional formula returned above under each input condition can be described as follows.

(a) $ds \cap \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\} = \phi$

In the `fa` combining algorithm, the evaluation of policies in the policy set continues till a policy in this set returns a decision other than $\boldsymbol{n}$. Let *dec* be this decision. For *dec* to be returned by *pol* in this sub-case, the first-applicable policy in the policy set should return *dec*. It is necessary to consider the possibility that any policy in the policy set of *pol* can be a first-applicable policy for a (*ps, r*) pair (*ps* is the protection state, and *r* is the access request). So the encoding generated by *DIC2SAT* reflects the conditions (1) any policy in the policy set of *pol* can be the first-applicable policy, and (2) the first-applicable policy returns *dec*.

(b) $\boldsymbol{n} \in ds$

An $\boldsymbol{n}$ decision will be returned by the `fa` combining algorithm if and only if every policy in the policy set returns an $\boldsymbol{n}$. The *DIC2SAT* algorithm encodes this condition under this input case.

Thus every possible propositional formula that can be returned in this sub-case has been described.

Sub-case 5.3.3.2. $com \sqsubset \text{LABEL}(pol)$

This is the sub-case where *com* points to a proper sub-policy of *pol*. The propositional formula returned in this sub-case is given below.

$$\phi = \Big(\bigwedge_{i=1}^{j-1} DEC2SAT(pol_i, \{\boldsymbol{n}\})\Big) \wedge DIC2SAT(pol_j, com, ds)$$

Here, pol_j is the sub-policy among $(pol_1, pol_2, \ldots, pol_k)$ for which $com \sqsubseteq pol_j$.

The propositional formula returned by *DIC2SAT* is such that the propositional formula is satisfiable if and only if the following conditions are satisfied.

- All policies in the policy set of *pol* that are evaluated before pol_j should return $\boldsymbol{n}$.

- The evaluation of pol_j should result in the recursive evaluation of sub-policy labeled *com*.
- The evaluation of this sub-policy should return a member of *ds*.

Thus the behaviour of $DIC2SAT$ for this case has been described.

5.3.1 Applicability to Non-Uniform Access Control System

The $DIC2SAT$ algorithm can be directly applied to uniform access control systems (Section 3.2). However, it can also be easily adapted to handle non-uniform systems. For a given *DIC* query, there may be dependencies among the truth values of the request predicates that appear in that *DIC* query, as not every variable assignment (a truth assignment to every request predicate in the *DIC* query) is realized by some state-request pair (*ps, r*). In such a scenario, the dependencies among these request predicates can be encoded as a propositional formula ψ. If a propositional formula returned by $DIC2SAT$ is ϕ, then we can test the satisfiability $\phi \wedge \psi$ to find if the corresponding *DIC* query is satisfiable in a non-uniform access control system.

The generalization of the above technique helps in determining the satisfiability of a *DIC* query for which certain conditions (that can be expressed as boolean combinations of request predicates) hold.

5.4 Illustrative Example

In this section I give a concrete example and discuss the execution of the $DIC2SAT$ algorithm to better explain its working.

Example 5.4.1. Consider the following inputs

$pol = rp_1 \rightarrow \texttt{fa}\Big(rp_2 \rightarrow \texttt{p}, rp_3 \rightarrow \texttt{d}, rp_4 \rightarrow \texttt{fa}\big(rp_3 \rightarrow \texttt{p}, rp_5 \rightarrow \texttt{d}\big)\Big)$,

com = the label of the AST node $(rp_5 \rightarrow \texttt{d})$,

$ds = \{ \boldsymbol{d} \}$

In this example, the *DIC2SAT* algorithm, will generate an encoding that will encode the following conditions.

1. Condition rp_1 should be satisfiable. This will ensure that the sub-policy in *pol* will be evaluated.
2. Let pol_j denote the sub-policy $rp_4 \rightarrow$ `fa` $\left(rp_3 \rightarrow \texttt{p},\ rp_5 \rightarrow \texttt{d}\ \right)$ in the policy set of *pol*. The combining algorithm `fa` stops evaluating the sub-policies in a policy set, once a sub-policy in the policy set returns a decision other than $\boldsymbol{n}$. In this example, for sub-policy pol_j to be evaluated all sub-policies that are evaluated before pol_j in the policy set should return $\boldsymbol{n}$. So the *DIC2SAT* algorithm should generate an encoding that would reflect the condition that all of the policies in the policy set evaluated before pol_j return $\boldsymbol{n}$.
3. The sub-policy pol_j is again a PCA Policy, consisting of two policies in its policy set (pol_{j1}, pol_{j2}). One of these policies $pol_{j2} = rp_5 \rightarrow$ `d` is the sub-policy labeled *com*. For pol_{j2} to be evaluated, the policy pol_{j1} that will evaluated before this sub-policy should return $\boldsymbol{n}$ (since the combining algorithm is `fa`). The algorithm encodes this condition.
4. Condition rp_5 in pol_{j2} should be satisfiable. This will ensure that the sub-policy in pol_{j2} will be evaluated.
5. The algorithm will then encode the condition that the sub-policy `d` should return $\boldsymbol{d}$ (since $\boldsymbol{d}$ is the only decision in *ds*).

The *DIC2SAT* algorithm being syntax directed is recursively executed for smaller policy constructs. The various recursive calls, the inputs given to each recursive call and the formula returned at each recursive call is given by Table 5.1.

Table 5.1: Recursion Table of $DIC2SAT$ for Example 5.4.1

No	Input	Output
0	$pol = rp_1 \to \texttt{fa}\Big(rp_2 \to \texttt{p}, rp_3 \to \texttt{d}, rp_4 \to \texttt{fa}\big(rp_3 \to \texttt{p}, rp_5 \to \texttt{d}\big)\Big)$ com = the label of AST node $rp_5 \to \texttt{d}$ $ds = \{\boldsymbol{d}\}$	$\phi = rp_1 \wedge \phi_1$
1	$pol = \texttt{fa}\ \Big(rp_2 \to \texttt{p},\ rp_3 \to \texttt{d},\ rp_4 \to \texttt{fa}\ \big(rp_3 \to \texttt{p},\ rp_5 \to \texttt{d}\ \big)\Big)$ com = the label of AST node $rp_5 \to \texttt{d}$ $ds = \{\boldsymbol{d}\}$	$\phi_1 = \phi_2 \wedge \phi_3 \wedge \phi_4$
2	$pol = rp_2 \to \texttt{p}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_2 = (rp_2 \wedge \phi_5) \vee \neg rp_2$
3	$pol = \texttt{p}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_5 = \bot$
4	$pol = rp_3 \to \texttt{d}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_3 = (rp_3 \wedge \phi_6) \vee \neg rp_3$
5	$pol = \texttt{d}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_6 = \bot$
6	$pol = rp_4 \to \texttt{fa}\ \big(rp_3 \to \texttt{p},\ rp_5 \to \texttt{d}\ \big)$ com = the label of AST node $rp_5 \to \texttt{d}$ $ds = \{\boldsymbol{d}\}$	$\phi_4 = rp_4 \wedge \phi_7$
7	$pol = \texttt{fa}(rp_3 \to \texttt{p}, rp_5 \to \texttt{d})$ com = the label of AST node $rp_5 \to \texttt{d}$ $ds = \{\boldsymbol{d}\}$	$\phi_7 = \phi_8 \wedge \phi_9$
8	$pol = rp_3 \to \texttt{p}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_8 = (rp_3 \wedge \phi_{10}) \vee \neg rp_2$
9	$pol = \texttt{p}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{n}\}$	$\phi_{10} = \bot$
10	$pol = rp_5 \to \texttt{d}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{d}\}$	$\phi_9 = rp_5 \wedge \phi_{11}$
11	$pol = \texttt{d}$ $com = \text{LABEL}(pol)$ $ds = \{\boldsymbol{d}\}$	$\phi_{11} = \top$

Since, there exists a satisfying truth assignment for the boolean variables in the final propositional formula generated by the $DIC2SAT$ algorithm ($rp_1 = rp_4 = rp_5 = True, rp_2 = rp_3 = False$), we can conclude that the query $DIC(pol,\ com,\ ds)$ is satisfiable too. This is clear from examining the inputs, since, there can exist a protection state ps and request r such that, evaluating request predicates rp_1, rp_4, rp_5 against $(ps,\ r)$ returns $True$, and, evaluating request predicates rp_2, rp_3 against $(ps,\ r)$ returns $False$.

5.5 Implementation

The implementation details of the $DIC2SAT$ algorithm and a description of the SAT solver used is given in Section 7.2.2.

Chapter 6

Compilation of the *DIC* Query: *PBS* Encoding

The satisfiability of the *DIC* query can be reduced to the satisfiability of a set of pseudo boolean constraints, collectively referred to as the pseudo boolean constraint satisfaction problem. This compilation of the *DIC* query can be done using the *PBS* encoding. The encoding converts the conditions under which the *DIC* query can return a *Yes* output into a set of pseudo boolean constraints. Hence, the satisfiability of these pseudo boolean constraints coincides with the satisfiability of the *DIC* query. In this chapter, I explain the need for an encoding other than the *SAT* encoding, and then I explain the alternative *PBS* encoding of the *DIC* query.

6.1 Need for the *PBS* encoding of *DIC* query

The need for this type of encoding of the *DIC* query arises from using the *DIC* query to analyse PCA Policies that use one of the majority voting algorithms to combine decisions from its components. The *SAT* encoding of a *DIC* query containing such a PCA Policy is a brute force encoding. The size of the encoding thus generated has an exponential growth as the size of the PCA Policy increases. *PBS*, however, provides a more efficient encoding.

There are three majority voting algorithms: Simple Majority Voting (`smv`), Absolute Majority Voting (`amv`), and Super Permit Majority Voting (`spmv`). A description of these voting algorithms was given in Section 3.3.2. A decision is returned by one of these algorithms based on the number of policies returning a decision in the policy set of the PCA Policy.

In Section 6.1.1, I describe the *SAT* encoding for these three algorithms and illustrate the problems with such an encoding.

6.1.1 *SAT* Encoding of a *DIC* query involving majority voting schemes

In [15], Bruns and Huth give a brute force *SAT* encoding of policies, involving majority voting algorithms. These majority voting schemes consider the number of policies returning a decision when arriving at a final decision for a PCA Policy.

Let $(pol_1, pol_2, \ldots, pol_k)$ be the policy set of a PCA Policy, where k is the number of policies in the policy set. Let a be a positive integer such that $a \geq 1$. For the `amv` voting algorithm, $a = \lfloor \frac{k}{2} \rfloor + 1$ policies in the policy set have to return a $\boldsymbol{p}$ (or $\boldsymbol{d}$) decision for $\boldsymbol{p}$ (or $\boldsymbol{d}$) to be returned as the final decision. Similarly for the `spmv` voting algorithm, $a = \lfloor \frac{2 \times k}{3} \rfloor + 1$ policies in the policy set should return the $\boldsymbol{p}$ decision, for the latter to be returned as the final decision. The brute force encoding essentially enumerates all possible $\binom{k}{a}$ combinations of policies in the given policy set. The resulting propositional formula is a disjunction of all possible a size subsets of policies in the policy set, returning the respective decision (either $\boldsymbol{p}$ or $\boldsymbol{d}$). The size of the resulting propositional formula is even worse for the `smv` voting algorithm, since at least $\sum_{a=1}^{\lfloor \frac{k}{2} \rfloor + 1} \binom{k}{a}$ combinations of policies will be generated in the brute force encoding.

The integer function denoting the number of combinations that are generated on using the `amv` algorithm is in $\Omega(2^{\frac{n}{2}})$, meaning that at least $2^{\frac{n}{2}}$ combinations are generated for the `amv` algorithm. Here n is the number of component policies in the policy set of the PCA Policy. The integer function denoting the number of combinations that are generated on using the `spmv` algorithm is in $\Omega(\frac{3}{2}^{\frac{2 \times n}{3}})$, meaning that at least $\frac{3}{2}^{\frac{2 \times n}{3}}$ combinations are generated for the `spmv` algorithm, when the input policy set has n component policies. For the `smv` voting algorithm, the integer function denoting the number of combinations generated is in $\Omega(2^n)$, since at least 2^n combinations will be generated for the `smv` voting algorithm.

An algorithm for enumerating all $\binom{k}{a}$ combinations in lexicographic order for specific k and a values is given in [41, page 181]. The following example illustrates how the brute force encoding is generated for the different majority voting algorithms.

Example 6.1.1. Consider the query $DIC(pol,\ com,\ ds)$, where,

$pol = \mathsf{pcamv}(pol_1, pol_2, pol_3)$,

$com = \text{LABEL}(pol)$, and,

$ds = \{\boldsymbol{p}\}$.

Here pcamv is a policy combining algorithm, such that pcamv $\in$ { smv, amv, spmv}. The policies pol_1, pol_2 and pol_3 are instances of the syntactic category $\mathcal{POLICY}$.

The following gives the SAT encoding adapted from [2] for the DIC query in 6.1.1 based on different values of pcamv.

(a) pcamv = smv

$$
\begin{aligned}
\phi \;=\; & \Big(DEC2SAT(pol_1,\{\boldsymbol{p}\}) \wedge DEC2SAT(pol_2,\{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3,\{\boldsymbol{p}\})\Big) \\
& \bigvee\Big(\big(DEC2SAT(pol_1,\{\boldsymbol{p}\}) \wedge DEC2SAT(pol_2,\{\boldsymbol{p}\})\big) \\
& \vee\big(DEC2SAT(pol_1,\{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3,\{\boldsymbol{p}\})\big) \\
& \vee\big(DEC2SAT(pol_2,\{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3,\{\boldsymbol{p}\})\big)\Big) \\
& \bigvee\Big(\Big(DEC2SAT(pol_1,\{\boldsymbol{p}\}) \\
& \wedge\big(\neg DEC2SAT(pol_2,\{\boldsymbol{d}\}) \vee \neg DEC2SAT(pol_3,\{\boldsymbol{d}\})\big)\Big) \\
& \wedge\big(\neg DEC2SAT(pol_2,\{\boldsymbol{d}\})\big) \wedge \big(\neg DEC2SAT(pol_3,\{\boldsymbol{d}\})\big) \\
& \vee\Big(DEC2SAT(pol_2,\{\boldsymbol{p}\}) \\
& \wedge\big(\neg DEC2SAT(pol_1,\{\boldsymbol{d}\} \vee \neg DEC2SAT(pol_3,\{\boldsymbol{d}\})\big)\Big) \\
& \wedge\big(\neg DEC2SAT(pol_1,\{\boldsymbol{d}\})\big) \wedge \big(\neg DEC2SAT(pol_3,\{\boldsymbol{d}\})\big) \\
& \vee\Big(DEC2SAT(pol_3,\{\boldsymbol{p}\}) \\
& \wedge\big(\neg DEC2SAT(pol_1,\{\boldsymbol{d}\} \vee \neg DEC2SAT(pol_2,\{\boldsymbol{d}\})\big)\Big)\Big) \\
& \wedge\big(\neg DEC2SAT(pol_1,\{\boldsymbol{d}\})\big) \wedge \big(\neg DEC2SAT(pol_2,\{\boldsymbol{d}\})\big)
\end{aligned}
$$

(b) pcamv = amv

$$\begin{aligned}\phi \quad = \quad & \big(DEC2SAT(pol_1, \{\boldsymbol{p}\}) \wedge DEC2SAT(pol_2, \{\boldsymbol{p}\})\big) \\ & \bigvee \big(DEC2SAT(pol_1, \{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3, \{\boldsymbol{p}\})\big) \\ & \bigvee \big(DEC2SAT(pol_2, \{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3, \{\boldsymbol{p}\})\big)\end{aligned}$$

(c) pcamv = spmv

$$\phi \quad = \quad DEC2SAT(pol_1, \{\boldsymbol{p}\}) \wedge DEC2SAT(pol_2, \{\boldsymbol{p}\}) \wedge DEC2SAT(pol_3, \{\boldsymbol{p}\})$$

In the above *SAT* encoding the *DEC2SAT* encoding can be arrived at by using the *DIC2SAT* using the following reduction from *DEC2SAT* to *DIC2SAT*.

$$DEC2SAT(pol,\ ds) \triangleq DIC2SAT(pol,\ \textsc{label}(pol),\ ds)$$

In case (a) for smv, the brute force *SAT* encoding is to enumerate all possible subsets (of sizes 1 to $\lfloor \frac{k}{2} \rfloor + 1$) of the policy set, such that the number of policies in the policy set returning a $\boldsymbol{p}$ is greater than the number of policies returning $\boldsymbol{d}$ for each such generated combination of policies returning $\boldsymbol{p}$.

In case (b) for amv, the brute force *SAT* encoding is to enumerate all possible $\binom{3}{2}$ ($k = 3$, $a = 2$, because one plus one-half of 3 is 2) combinations of the policies in the policy set that can return a $\boldsymbol{p}$ decision.

In case (c) for spmv, the brute force *SAT* encoding is to enumerate all possible $\binom{3}{3}$ ($k = 3$, $a = 3$, because one plus two-thirds of 3 is 2) combinations of the policies in the policy set that can return a $\boldsymbol{p}$ decision.

6.1.2 Problems

The following are some of the problems that are encountered when using the *SAT* encoding, for compiling the *DIC* query, as shown in the above example. The number of combinations n generated for such an encoding for each majority voting algorithm is given in Table 6.1. As could be inferred from the table, n has an exponential growth as k increases.

The resulting propositional formula (a disjunction) that encodes these combinations is directly proportional to n, since a disjunct will be generated for each generated combination.

Table 6.1: Rough Estimate of the Number of Combinations Generated for each Voting Algorithm in *DIC2SAT*

`smv`	`amv`	`spmv`
$n \geq \sum_{a=1}^{\lfloor k/2 \rfloor + 1} \binom{k}{a}$	$n = \binom{k}{a}$	$n = \binom{k}{a}$
$n = f(k) = \Omega(2^k)$	$n = f(k) = \Omega(2^{\frac{k}{2}})$	$n = f(k) = \Omega(\frac{3}{2}^{\frac{2 \times k}{3}})$
k is the number of policies of the policy set.	k is the number of policies in the policy set, and	k is the number of policies in the policy set, and
	$a = \lfloor k/2 \rfloor + 1$.	$a = \lfloor 2 \times k/3 \rfloor + 1$.

Thus the size of the propositional formula generated grows exponentially. The representation of this propositional formula in *CNF* is sizeable, with a large number of clauses in the resulting *CNF*. Consequently, substantial memory and time is required by *DIC2SAT* to encode a *DIC* query containing a majority voting algorithm. Hence, the *SAT* encoding of such a *DIC* query is not efficient.

Thus there is a need for an efficient encoding of the *DIC* query that involves the majority voting schemes. We propose the *PBS* encoding of the *DIC* query to overcome the problems above.

6.2 *DIC2PBS* Compilation Algorithm

The compilation of the *DIC* query using the *PBS* encoding is given by the *DIC2PBS* algorithm. The pseudocode of this *DIC2PBS* algorithm is given in Appendix B. A description of *DIC2PBS* algorithm's inputs, precondition, output, and postconditions follows.

Input *DIC2PBS* takes three inputs.

- *pol*: An access control policy that is an instance of the syntactic category $\mathcal{POLICY}$.
- *com*: The label of a sub-policy of *pol*.
- *ds*: A decision set which is a non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Precondition The following condition on inputs *com* and *pol* should hold.

$$com \sqsubseteq \text{LABEL}(pol)$$

The condition given is satisfied if and only if *com* identifies a sub-policy of *pol*.

Output *DIC2PBS* returns a $\langle CS, x\rangle$ pair. Here CS is a set of pseudo boolean constraints, and x is a pseudo boolean variable. This output satisfies the following postconditions.

Postcondition The following are the postconditions.

(a) The output $\langle CS, x\rangle$ is such that

- let RV be the set of variables in CS that correspond to request predicates; let AV be the set of other variables in CS;
- $x \in AV$;
- CS is always satisfiable, and for every variable assignment to the variables in RV, there is always an extension of that variable assignment to cover also the variables in AV, such that CS is satisfied;
- suppose σ is a variable assignment for the request predicates in *pol*; if σ satisfies the input *DIC* query, then every extension of σ to a satisfying assignment of CS will assign 1 to x; if σ does not satisfy the input *DIC* query, then every extension of σ to a satisfying assignment of CS will assign 0 to x; consequently, the input *DIC* query is satisfiable iff there is a variable assignment for CS for which x is assigned 1.

(b) Let CS' be the set of pseudo boolean constraints such that

$$CS' = CS \bigcup \{x = 1\}, \text{ where } \langle CS, x\rangle \text{ is the output of } DIC2PBS.$$

The constraint set CS' is satisfiable iff the *DIC* query on the same set of input values is satisfiable.

As described, the *DIC2PBS* algorithm generates a pseudo boolean constraint set and a pseudo boolean variable as output. We refer to such a pair given as output by the *DIC2PBS* as an *Instrumented Constraint Set* (*ICS*). Henceforth, all references to the *DIC2PBS* algorithm's output is considered to be an *ICS*.

6.3 Overall Structure of *DIC2PBS*

The *DIC2PBS* algorithm is also a *syntax directed recursive* algorithm. The pattern of recursion closely follows the inductive pattern of the grammar rules. The algorithm performs case analysis to handle each syntactic variant in the grammar. The skeleton of the pseudocode of the *DIC2PBS* algorithm illustrating this behaviour is given in Figure 6.1.

The algorithm handles two base cases and eight inductive cases. Either of these are explained below.

- Base Cases: The syntactic variants `p` and `d` of type $\mathcal{POLICY}$ are the base cases. The algorithm returns a simple *ICS* in either case.
- Inductive Cases: The syntactic variants representing Conditional Policy and the seven different PCA Policies constitute the eight inductive cases of the algorithm. In these cases, the algorithm is recursively applied to the sub-policies. The *ICS*s returned by the recursive calls of the algorithm are combined, to generate the final *ICS* returned by the algorithm for each inductive case.

Each syntactic case has sub-cases. These sub-cases are generated by the following two criteria.

1. The first criterion is whether *com* is the label of *pol*, or the label of a proper sub-policy of *pol*.
2. The second criterion is whether each of the decision values $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$ belongs to *ds*.

The two criteria can be combined to generate a maximum of $2 \times 4 = 8$ sub-cases in total. Furthermore, based on the input sometimes multiple sub-cases may apply. One example of such an instance is when *ds* contains multiple decision values.

In each main case an *ICS* which in turn is a combination of *ICS*s, will be built and returned by the *DIC*2*PBS* algorithm. Here, each applicable sub-case may contribute an *ICS* to this final *ICS* generated. Furthermore, the pseudocode in the appendix is organized such that each main case is handled by a specialized subroutine.

Please note that in our algorithm we assume that all PCA Policies have a policy set of size at least 2. This is a reasonable assumption since the case where the PCA Policy consists of a single policy in its policy set is uninteresting. This is because the evaluation of such a PCA Policy always returns the decision returned by this single policy in the policy set, irrespective of the combining algorithm used.

6.4 *PBS* Encoding

The *SAT* and *PBS* encodings of the *DIC* query are analogous to each other for all cases except for PCA Policies combined using voting algorithms. It would be possible to reuse the concepts from *DIC*2*SAT* for these similar cases in *DIC*2*PBS*. In order to do this, we need to be able to perform boolean combinations of *ICS*s. These operations on the *ICS*s should be semantically equivalent to boolean combinations of propositional formulae.

Each *ICS* $\langle CS, x \rangle$ encodes a condition that is either satisfied (when the pseudo boolean variable x is assigned 1) or not satisfied (when the pseudo boolean variable x is assigned 0). Given two *ICS*s, we want to construct another *ICS*, that captures the conjunction or disjunction of the two conjunctions. Similarly, given an *ICS*, we may want to construct another *ICS* that would capture the negation of the condition.

For the encoding of conjunction, disjunction and negation on *ICS*s I define two basic functions `AtLeast` and `AtMost`, that are described in the succeeding sections.

Figure 6.1: Skeleton of the PseudoCode of *DIC2PBS* (*pol*, *com*, *ds*)

(a) Skeleton for all *pol* types

```
1.  l := LABEL(pol)
2.  switch (pol) {
3.      case p:
4.          if (p ∈ ds) then
5.              ⋮
6.          else
7.              ⋮
8.      case d:
9.          if (d ∈ ds) then
10.             ⋮
11.         else
12.             ⋮
13.     case cond → pol′:
14.         if l =com then
15.             ⋮
16.         else /* com ⊏ l */
17.             ⋮
18.         if (n ∈ ds) then
19.             ⋮
20.     case po (pol1, pol2, ..., polk):
21.         ⋮
22.     case do (pol1, pol2, ..., polk):
23.         ⋮
24.     case fa (pol1, pol2, ..., polk):
25.         ⋮
26.     case oa (pol1, pol2, ..., polk):
27.         ⋮
28.     case smv (pol1, pol2, ..., polk):
29.         ⋮
30.     case amv (pol1, pol2, ..., polk):
31.         ⋮
32.     case spmv (pol1, pol2, ..., polk):
33.         ⋮
    }
```

(b) Skeleton of *pol* type PcaPolicy

```
1.  case pca (pol1, pol2, ..., polk):
2.      if l = com then
3.          if (p ∈ ds) then
4.              ⋮
5.          if (d ∈ ds) then
6.              ⋮
7.          if (i ∈ ds) then
8.              ⋮
9.          if (n ∈ ds) then
10.             ⋮
11.     else /* com ⊏ l */
12.         ⋮
```

6.4.1 `AtLeast`

In generating the *PBS* encoding, we may need to combine a group of conditions in a certain way. For example, we may need to generate a *PBS* encoding that would represent the satisfiability of k conditions from a set of n conditions, where $k \leq n$. The conjunction of these conditions would result in a *PBS* encoding that represents the satisfiability of all n conditions in a set. The disjunction of these conditions would result in a *PBS* encoding that represents the satisfiability of at least 1 condition from a set of n conditions.

In order to perform such combinations of conditions, we define the `AtLeast` function. This function takes a set of *ICS*s, where each *ICS* encodes a condition. It also takes a positive integer input that determining the combination to be performed. It generates an *ICS* as output, that captures the combination (conjunction or disjunction) of the input set. The inputs and output of the `AtLeast` function are described below.

The `AtLeast` function takes two inputs:

1. a set $\{\langle CS_1, x_1\rangle, \langle CS_2, x_2\rangle, \ldots, \langle CS_k, x_k\rangle\}$ of *ICS*s, and
2. a positive integer min.

On return, the `AtLeast` function returns an *ICS* $\langle OS, y\rangle$ such that:

- A solution to constraint set OS assigns 1 to pseudo boolean variable y iff the solution satisfies $\sum_{i=1}^{k} x_i \geq min$.

The pseudocode of the `AtLeast` function is given in Figure 6.2.

As could be seen from the algorithm presented (lines 4 and 5), the generated pseudo boolean constraint set OS includes the set union of the pseudo boolean constraint sets of each *ICS* in the input set. In addition, it contains two pseudo boolean constraints (line 6). These constraints are such that they ensure that at least min of the *ICS*s in the input set are satisfied when y is assigned 1. Whereas, when y is assigned 0 the constraints ensure that at most $min - 1$ of the *ICS*s in the input set can be satisfied.

Figure 6.2: PseudoCode of the `AtLeast` function

procedure `AtLeast` (s, min)
1. Let OS be an empty constraint set.
2. Let y be a fresh pseudo boolean variable.
3. $n := size(s)$
4. **for every** $\langle CS_i, x_i \rangle$ **in** s **do**
5. $\quad OS := OS \bigcup CS_i$
6. $OS := OS \bigcup \{\sum_{i=1}^{n} x_i \geq min \times y\} \bigcup \{\sum_{i=1}^{n} x_i \leq y \times n + (1 - y) \times (min - 1)\}$
7. **return** $\langle OS, y \rangle$

end procedure

To perform a conjunction of the set $\{\langle CS_1, x_1 \rangle \langle CS_2, x_2 \rangle \ldots, \langle CS_k, x_k \rangle\}$ of *ICS*s (each *ICS* encodes a condition), we pass in this set and a minimum threshold of k to the `AtLeast` function.

Given n conditions, each encoded as an *ICS*, their disjunction can be encoded by invoking `AtLeast` with a minimum threshold of 1.

6.4.2 `AtMost`

When generating the *PBS* encoding of a *DIC* query we may need to encode a combination of conditions involved. We may want to generate a *PBS* encoding that represents at most k conditions being satisfied from a set of n conditions. The negation of a condition should result in a *PBS* encoding that captures at most 0 of a singleton set containing only the condition to be negated.

We define the `AtMost` function to perform such combinations. This function takes a set of *ICS*s (each *ICS* represents a condition) and a positive integer determining the maximum threshold. The function returns an *ICS* that captures this combination of the input set of *ICS*s. The inputs and output of the `AtMost` algorithm follows.

The `AtMost` function takes two inputs:

1. a set $\{\langle CS_1, x_1 \rangle, \langle CS_2, x_2 \rangle, \ldots, \langle CS_k, x_k \rangle\}$ of *ICS*s, and

Figure 6.3: PseudoCode of the `AtMost` function

procedure `AtMost` (s, max)
1. Let OS be an empty constraint set.
2. Let y be a fresh pseudo boolean variable.
3. $n := size(s)$
4. **for every** $\langle CS_i, x_i \rangle$ **in** s **do**
5. $\quad OS := OS \bigcup CS_i$
6. $OS := OS \bigcup \{\sum_{i=1}^{n} x_i + y \geq 1 + (1 - y) \times max\} \bigcup \{\sum_{i=1}^{n} x_i \leq max \times y + (1 - y) \times n\}$
7. **return** $\langle OS, y \rangle$

end procedure

2. a positive integer max.

The `AtMost` function returns an *ICS* $\langle OS, y \rangle$ such that:

- A solution to constraint set OS assigns 1 to pseudo boolean variable y iff the solution satisfies $\sum_{i=1}^{k} x_i \leq max$.

The pseudocode for this `AtMost` function is given in Figure 6.3.

This algorithm generates an *ICS* as output (line 7). The output pseudo boolean constraint set OS includes the set union of the pseudo boolean constraint sets of all input *ICS*s (lines 4 and 5). In addition, two constraints that ensure the satisfaction of at most max of the input *ICS* is added to OS (line 6). The constraints ensure the following.

- When y is assigned 0 the constraints ensure that at least $max + 1$ of the input *ICS*s are satisfied.

- When y is assigned 1 these constraints ensure that no more than max number of *ICS*s of the input set are satisfied.

Given a condition encoded as an *ICS*, its negation can be encoded by invoking `AtMost` with a maximum threshold of 0.

6.4.3 Examples of Combinations of *ICS*s

Consider two *ICS*s as follows.

$$\begin{aligned}\langle CS_1, x_1\rangle &= \langle\{x_1 \leq 0\}, x_1\rangle \\ \langle CS_2, x_2\rangle &= \langle\{x_2 \geq 1\}, x_2\rangle\end{aligned}$$

$\langle CS_1, x_1\rangle$ and $\langle CS_2, x_2\rangle$ are *ICS*s encoding boolean *True* and *False* respectively, in our *PBS* encoding. The *ICS* $\langle CS_1, x_1\rangle$ is such that constraint set CS_1 is always unsatisfiable when $x_1 = 1$. CS_1 is always satisfiable when $x_1 = 0$. Similarly, the *ICS* $\langle CS_2, x_2\rangle$ is such that constraint set CS_2 is always satisfiable when $x_2 = 1$. CS_2 is always unsatisfiable when $x_2 = 0$. The examples below illustrate the `AND`, `OR` and `NOT` operations on these *ICS*s.

Example 6.4.1. `AND`

The conjunction of these two *ICS*s is obtained by invoking the `AtLeast` method on a minimum threshold of 2.

$$\begin{aligned}\langle CS_3, x_3\rangle &= \langle CS_1, x_1\rangle \texttt{ AND } \langle CS_2, x_2\rangle \\ \langle CS_3, x_3\rangle &= \langle CS_1 \bigcup CS_2 \\ &\quad \bigcup\{2x_3 + 1x_1 + 1x_2 \geq 0\} \\ &\quad \bigcup\{-1x_3 - 1x_1 + 1x_2 \geq -1\}, x_3\rangle\end{aligned}$$

From $\langle CS_3, x_3\rangle$ it is clear that the satisfiability of CS_3 depends on the value of x_3. Moreover, when $x_3 = 1$ the only possible assignments for x_1, x_2 are 1. When $x_1 = 1$ CS_1 becomes unsatisfiable. Hence, the constraint set CS_3 is unsatisfiable. This is the expected output since performing `AND` operation on *True* and *False* values results in *False*, which is always unsatisfiable.

Example 6.4.2. `OR`

The disjunction of these two *ICS*s is obtained by invoking the `AtLeast` method on a minimum threshold of 1.

$$
\begin{aligned}
\langle CS_3, x_3\rangle &= \langle CS_1, x_1\rangle \texttt{ OR } \langle CS_2, x_2\rangle \\
\langle CS_3, x_3\rangle &= \langle CS_1 \bigcup CS_2 \\
&\quad \bigcup\{-1x_3 + 1x_1 + 1x_2 \geq 0\} \\
&\quad \bigcup\{-1x_3 - 1x_1 + 2x_2 \geq 0\}, x_3\rangle
\end{aligned}
$$

From $\langle CS_3, x_3\rangle$ it is clear that the satisfiability of CS_3 depends on the value of x_3. Moreover, when $x_3 = 1$, the possible assignments for x_1 and x_2 are $x_1 = 0, x_2 = 1$, or $x_1 = 1, x_2 = 0$. When $x_1 = 1$ CS_1 becomes unsatisfiable. So if $x_2 = 1$ and $x_1 = 0$, all constraints in the set CS_3 are satisfiable. Hence, the constraint set CS_3 is satisfiable. This is the expected output since performing `OR` operation on *True* and *False* values results in *True*, which is always satisfiable.

Example 6.4.3. `NOT`

The negation of an *ICS* is obtained by invoking the `AtMost` method on a maximum threshold of 0.

$$
\begin{aligned}
\langle CS_3, x_3\rangle &= \texttt{NOT } \langle CS_1, x_1\rangle \\
\langle CS_3, x_3\rangle &= \langle CS_1 \bigcup\{+1x_3 + 1x_1 \geq 1, -1x_3 - 1x_1 \geq -1\}, x_3\rangle
\end{aligned}
$$

The negation of the *False* value should return a *True* value. This is reflected in the above example. If $x_3 = 1$ the CS_3 constraint set will be satisfied iff $x_1 = 0$. If $x_1 = 0$ then CS_1 is satisfiable. Note that CS_1 is unsatisfiable when $x_1 = 1$.

6.5 *PBS* encoding of Majority Voting Algorithms

This section elaborates the *PBS* encoding of the *DIC* query involving majority voting algorithms. This is one of the major contributions of this thesis work. We give a novel approach to implement the *DIC* query using pseudo boolean constraints. Previous work done in [32] used linear constraints to encode these voting algorithms in their composition framework.

In our approach to policy analysis, we compile all combining algorithms into pseudo boolean constraints.

The brute-force *SAT* encoding given in [15] had been described earlier in this chapter. The size of the resulting propositional formula has exponential growth, as the size of the number of policies in the policy set of the PCA Policy increases. Thus, there is a need for a more efficient encoding. The use of pseudo boolean constraints to encode the majority voting schemes seems like a better alternative. This is due to the natural way in which pseudo boolean constraints represent these kind of problems involving constraints.

I gave a technical description of the majority voting algorithms in Table 3.4. In this Table I describe the majority voting algorithms under the following assumption. Given a policy set $(pol_1, pol_2, \ldots, pol_k)$, each policy pol_j in this set (where j is the index of the policy in the policy set) is associated with four pseudo boolean variables $p_j, d_j, n_j,$ and i_j. Each variable represents a decision in the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$ respectively. When pol_j returns a decision, the pseudo boolean variable representing the decision is assigned value 1. The other three variables are assigned value 0.

In the *PBS* encoding of the majority voting schemes, the *DIC2PBS* algorithm recursively generates an *ICS* $\langle CS_j^{dec}, x_j^{dec} \rangle$ (where $dec \in \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$), for each policy pol_j in the policy set. Each of these *ICS*s represents the encoding of the query $\mathcal{DEC}(pol_j, dec)$. For example, for every pol_j in the policy set we capture p_j by an *ICS* $\langle CS_j^p, x_j^p \rangle$. When $x_j^p = 1$ the constraint set CS_j^p will be satisfied iff the query $\mathcal{DEC}(pol_j, \boldsymbol{p})$ is satisfiable. This step may be performed for the applicable decisions in $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$, for each policy in the policy set to generate the respective *ICS*s. These *ICS*s generated will be combined to generate the final *ICS* from the *DIC2PBS* algorithm.

The *PBS* encoding generated for the three majority voting schemes is outlined through the description below for some input cases.

Case 6.5.1. $pol = \texttt{smv}(pol_1, pol_2, \ldots, pol_k)$, $com = pol$, $ds = \{\boldsymbol{p}\}$

Let $\langle OS, y\rangle$ be the final *ICS* generated to encode a *DIC* query on these inputs. In order for the evaluation of *pol* to return $\boldsymbol{p}$ the following condition should hold.

$$\sum_{j=1}^{k} x_j^p > \sum_{j=1}^{k} x_j^d$$

Here, x_j^p and x_j^d are pseudo boolean variables in each of the *ICS*s generated for a policy pol_j in the policy set. Each pair of *ICS*s for a policy pol_j encode $\mathcal{DEC}(pol_j,\boldsymbol{p})$ and $\mathcal{DEC}(pol_j,\boldsymbol{p})$, respectively. x_j^p or x_j^d will be assigned 1 iff the corresponding $\mathcal{DEC}$ query is satisfiable.

(a) The constraint set *OS* is a set union of the following sets.

- The constraint set of each *ICS* in the set encoding the evaluation of policies in the policy set returning $\boldsymbol{p}$,
  $\{\langle CS_1^p, x_1^p\rangle, \langle CS_2^p, x_2^p\rangle, \ldots, \langle CS_k^p, x_k^p\rangle\}$.
- The constraint set of each *ICS* in the set encoding the evaluation of policies in the policy set returning $\boldsymbol{d}$,
  $\{\langle CS_1^d, x_1^d\rangle, \langle CS_2^d, x_2^d\rangle, \ldots, \langle CS_k^d, x_k^d\rangle\}$.
- A set of two constraints
  (1) $\{\sum_{j=1}^{k} x_p^j - \sum_{j=1}^{k} x_d^j \geq y + (1-y) \times -k\}$
  (2) $\{\sum_{j=1}^{k} x_p^j - \Sigma_{j=1}^{k} x_d^j \ \leq k \times y\}$
  These two constraints ensure that $\sum_{j=1}^{k} x_j^p > \sum_{j=1}^{k} x_j^d$.

(b) The pseudo boolean variable y is such that:

- When y is assigned 0 the constraint set *OS* is satisfiable iff,
  $-k \leq \sum_{j=1}^{k} x_j^p$ - $\sum_{j=1}^{k} x_j^d \leq 0$.
  This encodes the condition that the number of policies that return a $\boldsymbol{p}$ in the policy set is no more than the number of policies in the policy set that return a $\boldsymbol{d}$. This will ensure the unsatisfiability of the query *DIC*(*pol*, *com*, *ds*).

- When y is assigned 1 the constraint set OS is satisfiable iff,

  $1 \leq \sum_{j=1}^{k} x_j^p$ - $\sum_{j=1}^{k} x_j^d \leq k$.

  This encodes the condition that the number of policies that return a $\boldsymbol{p}$ in the policy set is strictly more than the number of policies in the policy set that return a $\boldsymbol{d}$. This will ensure the satisfiability of the query $DIC(pol,\ com,\ ds)$.

Case 6.5.2. $pol = \texttt{amv}(pol_1, pol_2, \ldots, pol_k),\ com = pol,\ ds = \{\boldsymbol{d}\}$

Let $\langle OS, y\rangle$ be the final ICS generated to encode a DIC query on these inputs. In order for the evaluation of pol to return $\boldsymbol{d}$ the following condition should hold.

$$\sum_{j=1}^{k} x_j^d > \tfrac{k}{2}$$

Here, x_j^d is a pseudo boolean variable in the ICS generated for a policy pol_j in the policy set. The ICS generated for this policy encodes the query $\mathcal{DEC}(pol_j, \boldsymbol{d})$. The pseudo boolean variable x_j^d can be assigned 1 iff the corresponding $\mathcal{DEC}$ query is satisfiable.

Let $a = \lfloor \frac{k}{2} \rfloor + 1$. The set CS^d contains ICSs, where each ICS corresponds to a policy pol_j in the policy set and encodes the query $\mathcal{DEC}(pol_j, \boldsymbol{d})$.

$CS^d = \{\langle CS_1^d, x_1^d\rangle, \langle CS_2^d, x_2^d\rangle, \ldots, \langle CS_k^d, x_k^d\rangle\}$.

The final generated ICS is given as follows.

$$\langle OS, y\rangle = \texttt{AtLeast}(CS^d, a).$$

The output can be described as follows.

- When y is assigned 0 the constraint set OS is satisfiable iff at most $a-1$ of the $\mathcal{DEC}$ queries encoded by the ICSs in the input set are satisfied. Intuitively, this is an encoding of the condition that the number of policies returning a $\boldsymbol{d}$ decision in the policy set is strictly less than a.
- When y is assigned 1 the constraint set OS is satisfiable iff at least a of the $\mathcal{DEC}$ queries encoded by the ICSs in the input set are satisfied. In other words,

this is an encoding of the condition that the number of policies returning a $\boldsymbol{d}$ decision in the policy set is no less than a.

Case 6.5.3. $pol = \texttt{spmv}(pol_1, pol_2, \ldots, pol_k)$, $com = pol$, $ds = \{\boldsymbol{p}\}$

Let $\langle OS, y\rangle$ be the final *ICS* generated to encode a *DIC* query on these inputs. In order for the evaluation of *pol* to return $\boldsymbol{p}$ the following condition should hold.

$$\sum_{j=1}^{k} x_j^p > \frac{2\times k}{3}$$

Here, x_j^p is a pseudo boolean variable in the *ICS* generated for a policy pol_j in the policy set. The *ICS* generated for this policy encodes the query $\mathcal{DEC}(pol_j,\boldsymbol{p})$. The pseudo boolean variable x_j^p can be assigned 1 iff the corresponding $\mathcal{DEC}$ query is satisfiable.

Let $a = \lfloor\frac{2\times k}{3}\rfloor + 1$. The set CS^p contains *ICS*s, where each *ICS* corresponds to a policy pol_j in the policy set, and encodes the query $\mathcal{DEC}(pol_j\boldsymbol{p})$.

$CS^p = \{\langle CS_1^p, x_1^p\rangle, \langle CS_2^p, x_2^p\rangle, \ldots, \langle CS_k^p, x_k^p\rangle\}$.

The final *ICS* generated is given below.

$$\langle OS, y\rangle = \texttt{AtLeast}(CS^p, a).$$

The output can be described as follows.

- When y is assigned 0 the constraint set OS is satisfiable iff at most $a-1$ of the $\mathcal{DEC}$ queries encoded by the *ICS*s in the input set are satisfied. Intuitively, this is an encoding of the condition that the number of policies returning a $\boldsymbol{p}$ decision in the policy set is strictly less than a.
- When y is assigned 1 the constraint set OS is satisfiable iff at least a of the $\mathcal{DEC}$ queries encoded by the *ICS*s in the input set are satisfied. In other words, this is an encoding of the condition that the number of policies returning a $\boldsymbol{p}$ decision in the policy set is no less than a.

Consider the Example 6.5.1.

Example 6.5.1. Let *DIC* (*pol*, *com*, *ds*) be the *DIC* query whose *PBS* encoding we are interested in. The input values of this query follows,

$pol =$ pcamv (pol_1, pol_2, pol_3),

$com =$ LABEL (pol), and

$ds = \{\ \boldsymbol{d}\ \}$.

In this *DIC* query pcamv is such that pcamv $\in \{$ `smv`, `amv`, `spmv` $\}$. Policies pol_1, pol_2, pol_3 are instances of the syntactic category $\mathcal{POLICY}$.

The *PBS* encoding of the above *DIC* query differs according to the type of pcamv used to combine individual policy decisions in *pol*. Each of these *PBS* encodings is described below.

(a) pcamv = smv

$$\begin{aligned}
\langle CS, x\rangle &= \langle CSD \bigcup CSP \bigcup \{x_1 + x_2 + x_3 - x_4 - x_5 - x_6 \geq x + (1-x)\times 3\} \\
&\quad \bigcup \{x_1 + x_2 + x_3 - x_4 - x_5 - x_6 \leq 3 \times x\}, x\rangle \\
CSD &= CS_1 \bigcup CS_2 \bigcup CS_3 \\
\langle CS_1, x_1\rangle &= DEC2PBS(pol_1, \{\boldsymbol{d}\}) \\
\langle CS_2, x_2\rangle &= DEC2PBS(pol_2, \{\boldsymbol{d}\}) \\
\langle CS_3, x_3\rangle &= DEC2PBS(pol_3, \{\boldsymbol{d}\}) \\
CSP &= CS_4 \bigcup CS_5 \bigcup CS_6 \\
\langle CS_4, x_4\rangle &= DEC2PBS(pol_1, \{\boldsymbol{p}\}) \\
\langle CS_5, x_5\rangle &= DEC2PBS(pol_2, \{\boldsymbol{p}\}) \\
\langle CS_6, x_6\rangle &= DEC2PBS(pol_3, \{\boldsymbol{p}\})
\end{aligned}$$

(b) pcamv = amv

$$\begin{aligned}
\langle CS, x\rangle &= \texttt{AtLeast}\ (CSD, 2) \\
CSD &= \{\langle CS_1, x_1\rangle, \langle CS_2, x_2\rangle, \langle CS_3, x_3\rangle\} \\
\langle CS_1, x_1\rangle &= DEC2PBS(pol_1, \{\boldsymbol{d}\}) \\
\langle CS_2, x_2\rangle &= DEC2PBS(pol_2, \{\boldsymbol{d}\}) \\
\langle CS_3, x_3\rangle &= DEC2PBS(pol_3, \{\boldsymbol{d}\})
\end{aligned}$$

(c) pcamv = spmv

$$
\begin{aligned}
\langle CS, x\rangle &= \texttt{AND}\left(\texttt{AtLeast}\ (CSD, 1),\ \texttt{AtMost}\ (CSP, 2)\right)\\
CSD &= \{\langle CS_1, x_1\rangle, \langle CS_2, x_2\rangle, \langle CS_3, x_3\rangle\}\\
\langle CS_1, x_1\rangle &= DEC2PBS(pol_1, \{\boldsymbol{d}\})\\
\langle CS_2, x_2\rangle &= DEC2PBS(pol_2, \{\boldsymbol{d}\})\\
\langle CS_3, x_3\rangle &= DEC2PBS(pol_3, \{\boldsymbol{d}\})\\
CSP &= \{\langle CS_4, x_4\rangle, \langle CS_5, x_5\rangle, \langle CS_6, x_6\rangle\}\\
\langle CS_4, x_4\rangle &= DEC2PBS(pol_1, \{\boldsymbol{p}\})\\
\langle CS_5, x_5\rangle &= DEC2PBS(pol_2, \{\boldsymbol{p}\})\\
\langle CS_6, x_6\rangle &= DEC2PBS(pol_3, \{\boldsymbol{p}\})
\end{aligned}
$$

In the above *PBS* encoding, the *DEC2PBS* encoding is generated, by, reducing the *DEC2PBS* to *DIC2PBS*, as follows,

$$DEC2PBS\ (pol,\ ds) \triangleq DIC2PBS\ (pol,\ \textsc{label}\ (pol),\ ds).$$

In case(a) above, for `smv` the *ICS* generated encodes the condition that the number of policies returning a $\boldsymbol{d}$ is less than the number of policies returning a $\boldsymbol{p}$.

In case(b) above, for `amv` the *ICS* generated encodes the condition that the number of policies returning a $\boldsymbol{d}$ in the policy set is at least 2 since $2 > \lfloor 3/2 \rfloor$.

In case(c) above, for `spmv` the *PBS* encoding generated encodes the conditions:

- The number of policies in the policy set returning a $\boldsymbol{p}$ is no more than 2 since $2 < (\lfloor 2/3 \rfloor \times 3) + 1$, and
- there is at least one policy in the policy set returning a $\boldsymbol{d}$.

The resulting *ICS* in each case is $\langle CS, x\rangle$. When $x = 1$, the satisfiability of the constraint set *CS* is comparable to the satisfiability of the *DIC* query on the same inputs.

On comparing the *SAT* and *PBS* encodings of the majority voting algorithms one can see that the *PBS* encoding of these majority voting algorithms is more natural than the

brute-force *SAT* encodings of these algorithms. The majority voting schemes return final decisions by comparing the number of policies returning a decision. Hence, it is easy to construct constraints that will capture these conditions. Also, it is easier to implement the *PBS* encoding of the voting algorithms. The size of the *PBS* encoding (number of *ICS*s combined in the final *ICS* encoding) has linear growth as the number of policies in the policy set of the PCA Policy increases. This is more efficient than the *SAT* encoding, since there is an exponential growth in the propositional formula generated with the increase in number of policies in the policy set. In the next chapter we prove our claim that *PBS* encoding of voting algorithms is more efficient than its *SAT* counterpart, through empirical analysis.

6.6 Implementation

The details of the implementation of the *DIC*2*PBS* algorithm, and the *PBS* solver used is given in Section 7.2.3.

Chapter 7

Empirical Study

In this chapter I report the empirical study that was conducted as part of this thesis work. The purpose of the study was to compare the relative efficiency for evaluating a *DIC* query via the *PBS* encoding versus the *SAT* encoding.

7.1 Study Aim

The empirical analysis was conducted to test the hypothesis below.

Hypothesis: *The PBS encoding of the DIC query is more efficient, in terms of time and space, compared to the corresponding SAT encoding, when analysing a complex PCA Policy that is combined using one of the majority voting schemes —* `smv, amv, spmv`.

7.2 Implementation Details

In this section, I describe the implementations that were made as part of the empirical study.

7.2.1 Random Instance Generation

I implemented a random instance generator that generates a *pol*, *com*, *ds* triple on each run. Each of these is described below.

- *pol*— a PCA Policy of the form, `pca`$(pol_1, pol_2, \ldots, pol_k)$. This policy has as its `pca` one of the three majority voting algorithms — `smv, amv, spmv`. The probability of selection of each of these algorithms is 1/3 (equal probability) on each run of the random instance generator.

- *com*— the label of a sub-policy of *pol*.

- *ds*— a non-empty subset of the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

The *DIC*2*SAT* and *DIC*2*PBS* algorithms operate on the above three inputs and encode the conditions under which the *DIC* query is satisfied into *SAT* and *PBS* instances, respectively.

Generation of *com*

In our empirical analysis, for all input instances generated by the random instance generator, the value of *com* was as follows.

$$com = \text{LABEL}\ (pol)$$

This was to ensure that the encodings (*SAT* and *PBS*) generated were of a *DIC* query containing majority voting algorithms. This may not be the case when $com \sqsubset pol$. Since *com* is always the label of *pol*, we are essentially encoding the query $\mathcal{DEC}(pol, ds)$ (Section 4.6), into *SAT* and *PBS* instances for this *com* value.

Generation of *ds*

A *non-empty* subset of the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$, will be generated randomly. We ensure that there is an equal probability of selecting each *non-empty* subset of this set. We enumerate all 15 non-empty subsets of the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$. A subset will then be selected at random.

Generation of *pol*

The *pol* input is a syntactically valid policy in our policy composition framework. We designed an algorithm for generating random instances of policies. This algorithm is given in Appendix C. The algorithm will associate an integer, with each policy generated, as its label. A description of this algorithm's inputs and outputs is given below. Every PCA Policy generated by this algorithm, contains at least two policies in its policy set. The reason being, the case where the PCA Policy contains only one policy is uninteresting for this empirical study.

Input: This algorithm takes the following two inputs:

- n — a positive integer that denotes the size of the policy to be generated, where, $n \geq 3$, and,

- m — a positive integer, representing the number of distinct request predicates that may occur in the generated policy. Here, $m \geq 1$.

Output: A policy *ranpol*, of size n (denoting the number of policy constructs in this policy). *ranpol* is a PCA Policy, whose `pca` $\in$ { `smv`, `amv`, `spmv`}. The remaining $n-1$ policies are generated by calling the algorithm recursively on smaller values of n. If this policy contains a sub-policy of type Conditional Policy, then, the condition of this sub-policy is a request predicate, denoted by a number between 1 and m. The selection of each request predicate, from 1 to m, by our algorithm, has equal probability.

The choice of m will affect the hardness of the satisfiability of the final encoding generated by $DIC2SAT$ and $DIC2PBS$. Small number of request predicates would result in a harder satisfiability problem, whereas large number of request predicates results in a relatively easier problem. This is because the satisfiability of the SAT and PBS encodings depend on the existence of a protection state where the request predicates do not conflict with each other. Hence, the interplay of the different request predicates is an important aspect in the satisfiability instance generated. We set m to be one-eighth of the policy size, n. We consider this number to be small enough, to generate satisfiability instances of moderate difficulty.

The described algorithm will be called recursively to build a policy of size n, by calling itself on smaller sizes. The behaviour of this algorithm differs for different values of n. Each of these cases is discussed below.

Case 7.2.1. $n = 1$

In this case, the random policy generated, consists of a single sub-policy. Hence, an atomic policy will be generated. This will be either `p` or `d`. There is an equal probability of selecting either of these atomic policies in our algorithm.

Case 7.2.2. $n = 2$

In this case, the random policy generated will be of size 2. Hence, a Conditional Policy, $rp \rightarrow pol$, will be generated in this case. The request predicate rp will be chosen in random from the m request predicates (each chosen with equal probability). The sub-policy pol will be an atomic policy — `p` or `d`, with either policy having an equal chance to be selected.

Case 7.2.3. $n \geq 3$

In this case, the generated random policy is a composite policy, namely, a PCA Policy or a Conditional Policy. Each kind of policy will be selected with 1/2 probability.

If a Conditional Policy is selected, then it has the form $rp \rightarrow pol$. The condition rp is generated randomly (with an equal chance of selecting one of the m request predicates). The sub-policy pol will be a random policy generated by recursively calling the algorithm with size $n - 1$.

If a PCA Policy is selected, then it has the form $\texttt{pca}(pol_1, pol_2, \ldots, pol_k)$. Firstly, `pca` will be selected from the set {`po`, `do`, `fa`, `oa`, `smv`, `amv`, `spmv`}. The probability of selecting a combining algorithm from this set is 1/7. Secondly, the size of the policy set k, and, the size of each policy in the policy set will be decided. For this purpose, we used the mathematical concept of the *composition* of an integer.

A composition of a positive integer q can be described as given in [41, page 190] as follows. A composition is an ordered partition of a positive integer q represented as a sequence of integers $c_1, c_2, \ldots, c_l$, such that, $c_1 + c_2 + \cdots + c_l = q$. A composition of an integer can be represented as follows. Let q be a positive integer, such that $q \geq 1$. Consider a straight line of length q. Let this line be divided into q parts of unit length by $q - 1$ points. Now, a composition can be generated for q by selecting t of these $q-1$ points, where, $0 \leq t \leq n-1$. The number of elements of the composition will be $t+1$, and, each element of the composition will be the distance between adjacent points selected on the line.

We can represent this selection of t points from $q-1$ points as a binary number. Let b be

a binary number of the form, $b_1 b_2 \ldots b_{q-1}$. This number has a binary digit representing each of the $q-1$ points. If the binary digit corresponding to a point is 1, it means the point is selected, or else, if it is 0 it means the point is not selected. Each of these possible selection of a set of points for generating the composition of an integer q can be represented as the binary representations of integers between 0 and 2^{q-1}, inclusive. Each binary representation is of length $q-1$ (achieved by prepending 0's). Thus, there are 2^{q-1} compositions of an integer q.

We define a *degenerate composition* of positive integer q to be the composition of length 1 (t=0, none of the $q-1$ points are selected). All other compositions of q are referred to as *non-degenerate compositions*.

In this case, in the second step of generating a PCA Policy, we need to decide on the size of the policy set and the size of each policy in this policy set. For this purpose, we generate a non-degenerate composition of the integer $n-1$ of the form $(c_1, c_2, \ldots, c_k)$, such that $c_1 + c_2 + \cdots + c_k = n-1$. The policy set generated for this PCA Policy contains k component policies. The i^{th} policy in the set $(pol_1, pol_2, \ldots, pol_k)$ is of size c_i, where $1 \leq i \leq k$. In Appendix C we present the algorithm to randomly select (with equal probability) and generate, a non-degenerate composition of an integer. We deal with only non-degenerate compositions, as PCA Policies generated by this algorithm consists of at least two policies in its policy set.

Example 7.2.1. The set of all compositions of 3 is as follows. The number of compositions of 3 is $2^{3-1} = 4$.

$Compositions(3) = \{(1,1,1), (1,2), (2,1), (3)\}$

In our algorithm we eliminate the degenerate composition (3), since we want the policy set of the generated PCA Policy to contain at least 2 policies.

Thus, this algorithm generates random input instances. This algorithm was implemented in JAVA, using 350 lines of code.

7.2.2 *DIC2SAT* Implementation

The *DIC2SAT* algorithm was implemented in JAVA. The input policy *pol* is represented as an abstract syntax tree. The label *com* is a reference to a node in the AST. The decision set *ds* is a non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$. On return, the abstract syntax tree of a propositional formula is generated. This algorithm performs a post processing step after the generation of a propositional formula. This step is to generate the formula's equivalent *CNF* form. This transformation is done based on the Tseitin transformation [47]. The generated CNF is written to a text file to be given as input to the SAT solver. This algorithm was implemented using 2210 lines of JAVA code, with 1176 lines of code being shared with the *DIC2PBS* implementation.

MiniSat Solver

The *SAT* Solver that we use for our empirical study is the MiniSat Solver [21, 4]. This solver was implemented in C++. This *SAT* solver was selected because it is easily implementable, open source and efficient. This solver receives the CNF file as input. It outputs whether this is a satisfiable instance or not. Also, the time in seconds taken by the solver to determine the satisfiability of this instance will be generated.

7.2.3 *DIC2PBS* Implementation

The *DIC2PBS* algorithm was implemented in JAVA. The input policy *pol* is represented as an Abstract Syntax Tree (AST). The label *com* is s reference to a node in the AST. The input *ds* is a non empty subset of the set $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$. A data-structure was constructed for the tuple of type $\langle CS, x\rangle$. A data-structure was constructed to represent each constraint in the set CS. A post processing step was included to convert the generated tuple $\langle CS, x\rangle$ into a set of pseudo boolean constraints CS' such that

$$CS' = CS \bigcup \{x = 1\}.$$

The constraint set CS' was written to a text file and fed to a *PBS* solver to determine its satisfiability. More information about the *PBS* solver used, is explained in the next chapter. The satisfiability of these set of pseudo boolean constraints is equivalent to the satisfiability of the corresponding *DIC* query. The implementation of *DIC*2*PBS* algorithm took 1949 lines of JAVA Code, of which, 1176 lines of JAVA code were used by both *DIC*2*SAT* and *DIC*2*PBS* implementations.

MiniSat+ Solver

The *PBS* solver that we use in our empirical study is the MiniSat+ solver [22, 5]. This solver is also implemented in C++ and developed by the same team that developed the MiniSat solver. The MiniSat+ solver receives a text file with pseudo boolean constraints as input. It outputs whether this set of pseudo boolean constraints is satisfiable or not. It also outputs the time taken by the solver to determine the satisfiability of this instance.

The MiniSat+ solver solves a *PBS* instance by an optimized conversion to a *SAT* instance [22]. Essentially, our contribution is on devising a compact *PBS* encoding to cope with the complexity of majority voting, and advocating the reliance on state-of-the-art *PBS* solvers to decide how to generate an optimized *SAT* encoding from the *PBS* encoding.

7.3 Challenges

Before conducting the experiment, we performed some pilot runs on both a MacBook with 4 GB RAM and an Intel Core 2 Duo processor, and on the hardware used to run the experiment (refer to Section 7.4.2). We observed the following behaviour during these pilot runs. We also describe how this observed behaviour has influenced our experimental setup.

- In terms of memory, the JVM was throwing an Out Of Memory error when *DIC*2*SAT* algorithm was executed due to insufficient Heap Space. This occurred for even small policy sizes of around 30, when we allocated 2 GB of

RAM to the JVM. We did not face any memory issues on implementing the *DIC*2*PBS* algorithm on a JVM with 2 GB memory for policy sizes around 30. To deal with this problem, we decided to run our experiment on a High Performance Computing Server of the University of Calgary, BigBox (further hardware details in Section 7.4.2), and allocate 10 GB of RAM to the JVM.

- When the *DIC*2*SAT* algorithm was executed, by allocating 10 GB of RAM space to the JVM, we had to wait for 6 hours before the JVM threw an Out Of Memory Error message, when trying to encode a *DIC* query handling a PCA Policy of size 190. This was a result of the JAVA Heap Space filling up to its maximum capacity of 10 GB before generation of the entire formula. Once this occurred, the Garbage Collector starts to work overtime to clear the unwanted memory space. This produced a GC Over limit Exceeded error in JAVA. For this reason, we decided to allocate a time-limit of 15 minutes each, for compiling the *DIC* query into *SAT* and *PBS* instances, respectively, when conducting the experiment.

7.4 Experiment Setup

This section illustrates the details of this empirical analysis.

7.4.1 Measurements

The following measurements were made as part of the study.

1. **CT-SAT** The time in seconds to compile a *DIC* query, on a random input instance, to a *SAT* instance, using the *DIC*2*SAT* implementation.

2. **CT-PBS** The time in seconds to compile a *DIC* query, on the same input instance, to a *PBS* instance.

3. **ST-SAT** The time in seconds taken by the MiniSat *SAT* solver, to determine the satisfiability of the generated *SAT* instance.

4. **ST-PBS** The time in seconds taken by the MiniSat+ *PBS* solver, to determine the satisfiability of the generated *PBS* instance.

7.4.2 Hardware

The experiment was conducted on a High Performance Computing Unix server at the University of Calgary. Specifically, the server on which the empirical analysis was conducted was named BigBox. It had a 16 core processor and a 128 GB RAM. I allocated 10 GB RAM to the Java Virtual Machine (JVM).

7.4.3 Software

A Perl script was written to automate the experiment.

7.4.4 Timing Device

The **CT-SAT** and **CT-PBS** were measured using the tool JETM, Java Execution Time Monitor. This tool was embedded in the code to determine the CPU time taken by $DIC2SAT$ and $DIC2PBS$ to generate the *SAT* and *PBS* encodings of the *DIC* query respectively. The tool measured the time taken in milliseconds (10^{-3} seconds) to generate the respective encodings. This time was converted to seconds for comparison with solver times, which was generated in seconds.

The **ST-SAT** and **ST-PBS** were gathered from the respective solvers. The *SAT* and *PBS* solvers that we use, in addition to determining the satisfiability of the instances, also give the time taken in seconds, by the solver to solve that instance.

7.4.5 Input Instances

An input instance was generated randomly using the random instance generator explained in the previous section. Input instances were generated for different values of n, where n is the number of sub-policies in the random policy generated by the random instance generator. The input range of n was specified to acquire the input instances in that range. The interval for each range was 17 for the first range $(3 - 20)$, and then, 20 for the subsequent ranges. The minimum value of n was 3, whereas the maximum value was 300. Input instances were generated for random values of n within each range. A total of 40 input instances were generated in each range.

7.4.6 Measurement of Performance

For each input instance that was generated by the random instance generator, in each range, the following was done:

- The input set *pol*, *com*, *ds* was given as input to $DIC2SAT$ and $DIC2PBS$ implementations. The method of input generation was given in Section 7.2.1.
- If the time taken for generating the respective encodings was within 15 minutes, the time taken to generate the encoding will be recorded. Otherwise, a timeout will be recorded for the respective implementation.
- The generated encodings were written to the text files and fed as input to the SAT and PBS solvers.
- The time taken for solving the satisfiability of the respective instances — SAT or PBS, was then recorded.

7.5 Experiment Results

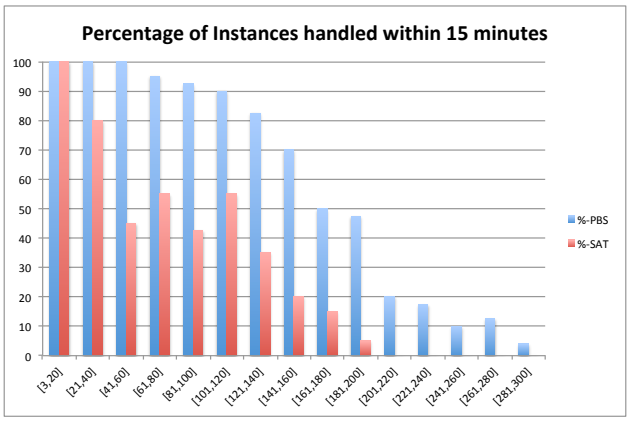
Each data point was collected in the above manner. The average compiler and solver times for *SAT* and *PBS* instances (only those *SAT* and *PBS* instances that do not timeout) were calculated for each input range. The purpose was to compare the average times taken, by the two encodings, in each interval. These values were plotted on graphs. Moreover, we also analysed the percentage of input instances that were handled by *DIC2SAT* and *DIC2PBS* within the time limit of 15 minutes. This was done to study the overall performance of *DIC2SAT* and *DIC2PBS* and, compare them. These results were also plotted on a graph. The results of this empirical study are given in Figure 7.1.

The X-axis, for all the graphs that were plotted, indicates the range of n, where, n is the size of the random policy generated by the random instance generator. The Y-axis of Graphs 7.1b and 7.1c are the average compiler and solver times for instances generated in each range, respectively. The X-axis of the Graph 7.1a gives the percentage of handled instances within the timeout period (of 15 minutes) for both *DIC2SAT* and *DIC2PBS*.

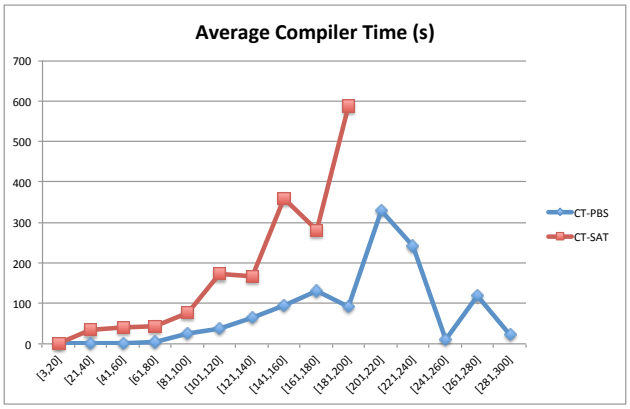
From the Graph 7.1a, we see that *DIC2SAT* handles instances generated for policy sizes up to 20 only, whereas *DIC2PBS* handles all generated instances of size up to 60. However, for larger policy sizes *DIC2PBS* consistently handles more input instances than *DIC2SAT* within the given timeout period of 15 minutes. Especially, the performance gap between *DIC2SAT* and *DIC2PBS* is noticeable for $n \geq 121$, where *DIC2PBS* handles more than half the number of instances handled by *DIC2SAT*. The *SAT* encodings generated as the policy size increases, are of exponential sizes and hence, there is a rapid increase in compile time (as could be seen from the graph). The *PBS* encodings are much more compact and hence, the comparatively shorter compile times that do not grow as fast. Thus, we may conclude that the overall performance of *DIC2PBS* is better than *DIC2SAT*.

From the Graph 7.1b, we observe that there is a very small difference in the performance of *DIC2SAT* and *DIC2PBS* for smaller instances ($n \leq 100$), with the *DIC2PBS* showing

(a) Comparison of Percentage of Handled Instances by *DIC2SAT* and *DIC2PBS*



(b) Comparison of Average Compile Times of *DIC2SAT* and *DIC2PBS*



(c) Comparison of Average Solver Times of *SAT* and *PBS* Instances

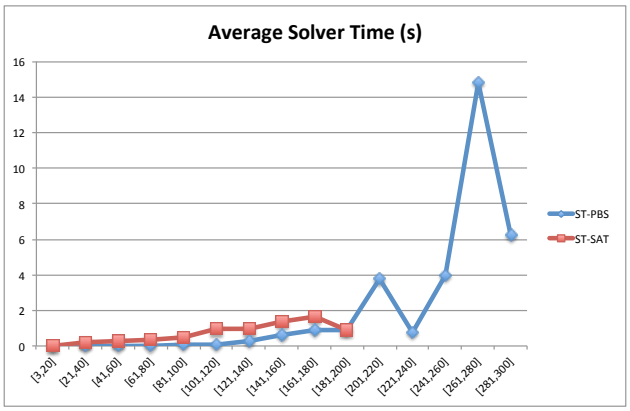


Figure 7.1: Empirical Results

better performance than $DIC2SAT$. However, for larger values of n $DIC2PBS$ is much more efficient than $DIC2SAT$. The longest time taken to compile an instance by $DIC2SAT$ is around 10 minutes, whereas for $DIC2PBS$ it is only around 5 minutes. There is a noticeable decrease in compile times for $DIC2PBS$ for $n \geq 221$, since the percentage of handled instances is very small.

From the Graph 7.1c, we observe that the SAT and PBS solvers exhibit linear performance for policy sizes up to 100. For $n \geq 200$, we notice a considerable drop in the performance of the PBS solver. This may partly be attributed to the fact that the number of SAT instances solved is quite small when compared to the number of PBS instances solved by the respective solvers. Also, the SAT solvers use techniques that have been under research for nearly 5 decades (the Davis Putnam algorithm [19] was given in 1960), whereas the techniques for solving PBS instances have seen some major breakthrough only in the recent few years. The longest time taken by the SAT solver to solve a SAT instance is around 2 seconds, whereas the longest time taken by the PBS solver is around 16 seconds. The time difference is around 14 seconds. This is not very steep, considering that the PBS solver handles instances of policy size up to 300. As the technology for solving PBS instances further develops in the future, we anticipate the performance gap will become narrower.

7.5.1 Summary of Findings

The following is a concise summary of our findings when using the SAT or PBS encodings of the DIC query handling pol, which is a PCA Policy combined using one of the three majority voting algorithms.

(i) Both SAT and PBS encodings of the DIC query are equally efficient in handling policies of size up to 20.

(ii) In terms of the compile time, $DIC2PBS$ is more efficient than $DIC2SAT$ for all instances.

(iii) In terms of the solver times for the respective encodings, *PBS* exhibits longer solver times for policy sizes greater than 200. This may be attributed to the large number of instances handled by *DIC*2*PBS* for larger values of n.

(iv) The *DIC*2*SAT* and *DIC*2*PBS* implementations handle all instances of policy size up to 20 for the timeout limit of 15 minutes.

(v) The number of instances handled by *DIC*2*PBS* is far greater than the number of handled instances of *DIC*2*SAT* for a timeout limit of 15 minutes for policies sizes greater than 20.

7.6 Conclusion from the Empirical Analysis

Thus from the empirical study, we may conclude the following:

(a) If we were to encode a *DIC* query containing majority voting schemes, *PBS* encoding is a better option than *SAT* encoding in terms of efficiency.

(b) If we have a time constraint when analysing large policies using *DIC* queries, *PBS* encoding is a wise option. This is because *DIC*2*PBS* handles large *DIC* instances more gracefully than its *DIC*2*SAT* counterpart. Hence, it is more likely that the large problem instance will be handled by *DIC*2*PBS* within the given time limit.

Chapter 8

Conclusion, Related Work, Future Work

8.1 Conclusion

In this section I give a summary of my contributions through this work.

(a) I introduce a new white-box policy analysis technique, for analysing a composite policy. I presented how the *DIC* analysis query determines the authorization decision returned by an access control policy, in the context of the composite policy. I then discussed the query's various applications.

(b) I presented the abstract syntax and semantics of our policy composition framework based on XACML's policy language model. I then illustrated the generality of this policy composition framework.

(c) I designed an algorithm *DIC*2*SAT* for the conversion of a *DIC* query to a *SAT* instance. I elucidated the need for a new encoding of the *DIC* query, as the *SAT* encoding is inefficient when majority voting combining algorithms are involved. For this reason, I designed another algorithm *DIC*2*PBS* that converts the *DIC* query to a *PBS* instance.

(d) I implemented the *DIC*2*SAT* and *DIC*2*PBS* algorithms, and compared the efficiency of these algorithms in compiling a *DIC* query into the respective encoding. I focused specifically on *DIC* queries analysing composite policies using a majority voting combining algorithm. I demonstrated the efficiency of the *PBS* encoding over the *SAT* encoding of *DIC* through empirical analysis.

8.2 Related Work

In this section I elaborate some of the work related to this research, and, how my work differs from them.

8.2.1 Policy Compostion Framework

In [39], Ni et al. give a D-Algebra that aids policy composition. This algebra presents a method to specify the combination techniques used to combine individual decisions in a composite policy. The specification of a composition function in this framework is as a function in D-Algebra. They specify a table whose row and column heads specify authorization decisions. Every row and column head intersection in this table gives the decision returned by combining these two decisions using a composition function. Such a table is provided in their framework as a representation of composition functions. They provide mechanisms to convert this representation to a D-Algebra function. They specify XACML rule and policy combining algorithms using tables (as described above), and their equivalent D-Algebra functions. D-Algebra uses values in Multi Valued Logic to represent individual decisions. They also provide a rational number interpretation of their Algebra to encode majority voting algorithms. They use integer variables to represent every authorization decision. Each integer variable represents the number of policies in the policy set that return a decision. The majority voting algorithms are specified as a comparison between these integer variables.

In [32], Li et al. give a Policy Combining Algorithm (PCA) framework, called PCL (Policy Combining Language). This language has two variants: (1) to represent majority voting combining algorithms, and (2) to represent combining algorithms other than these majority voting algorithms. The (1) variant is specified using linear constraints on variables representing the number of policies returning a decision. The (2) PCL variant specifies composition functions using DFA, or as tables similar to tables representing composition functions in [39]. They provide techniques to handle obligations, that may be part of policies

being combined in a composite policy.

The composition frameworks provided in [39] and [32] are both elegant and strive to eliminate some erratic behaviour seen in XACML. In our framework, we design a composition framework similar to XACML, since it is well known and widely implemented. Adapting our *DIC* analysis for analysing policies in the composition frameworks given in [39] and [32] is possible.

In [15], Bruns and Huth give a policy language based on the four valued Belnap Logic. They show that policy decision combinations in a composite policy can be performed by using the *meet* and *join* operators of the Belnap bilattice. They prove that it is possible to represent all known policy combining algorithms using these operators.

Our policy composition framework is similar to the policy composition framework given by the work of Bruns and Huth [15], in the following ways. A policy in their policy language returns one of the four Belnap values. In our framework a policy can return one of the four decisions "permit", "deny", "not-applicable", or "indeterminate". Their composition framework handles both voting and non-voting combining algorithms, similar to our composition framework. However, the difference lies in the way these voting algorithms are being handled in their framework (as a *SAT* instance encoding). We handle these voting algorithms much more efficiently (as *PBS* instance). It is possible to utilize our *DIC* analysis on policies in this framework.

8.2.2 Policy Analysis and Testing

There are various types of policy analysis techniques that has been proposed previously [27, 46, 28, 37, 12, 30, 49], the following are the ones that are related to the policy analysis technique proposed by us.

The work done in [34] presents the following types of policy analysis— analysis of information about the policy, analysis of information about the structure of the policy, analysis of the decision returned by a policy, and analysis of the type of access request to which the

policy is applicable. They provide a method to determine whether a policy applicable to requests satisfying a request predicate, can return a specific decision. They provide two techniques to analyse the similarity between two policies. In the first technique, they compare the policy specifications to retrieve common attributes based on how the policy is defined. Then, they decide a similarity score of 0 to 1 on this basis. In the second technique , two policies are analysed to determine if, each policy returns a specific decision, for a common set of access requests, satisfying a request predicate.

In [15], the following analysis techniques are presented. Firstly, they provide a technique to determine if a policy returns a specific decision on at least one access request. Secondly, they provide techniques to compare two policies: (1) they provide a technique to determine whether a policy is more permissive than another policy, and (2) they determine policy similarity by determining whether two policies return the same access decision for all access requests, or for all access requests that satisfy a request predicate.

In [25], Hughes and Bultan compare the extent of similarity between two access control policies, by determining, the number of common access requests to which these policies apply to.

In [23], Fisler et al. give two kinds of policy analyses. The first, is to determine whether a safety property is being satisfied by an access control policy. The second, is to perform change impact analysis on two policies. They use Multi Terminal Binary Decision Diagrams (MTBDDs) [11, pages 392–422] to represent policies. Then, they combine the MTBDDs of the policy before and after change. From this combined diagram, they effectively determine the access requests, for which, the policy returns a different authorization decision, before and after modification.

In [36], Martin and Xie provide a mechanism to provide the most effective test cases in the form of (access request, decision) pairs, to test a *Policy* in XACML. They target each rule in such a policy, and, determine an access request whose access decision is affected by

that specific rule. Then, they add the (access request, decision) pair to the set of test cases.

In all of the above analysis techniques mentioned, with the exception of [36], the analysis is similar to typical black-box testing. Here, a policy is analysed based on either the type of access request it is applicable to, or the authorization decision that is returned for a specific access request. The position of this work is that the structure of a policy is important when analysing access control policies, specifically composite policies. My work differs from all of the above in that it is possible for the policy analyst to check a single policy within a composite policy, in the context of execution of the composite policy. In [36], only rules can be targeted within a policy (in XACML) to produce effective test cases. The techniques provided for generating test cases for XACML rules in a XACML policy cannot be applied to policies in a composite policy.

8.2.3 Approach to Policy Analysis

In [15] and [25], the analysis approach is to transform the analysis of a policy to a *SAT* instance. This approach produces exponential encoding in some cases (analysis involving PCA Policies containing majority voting algorithms). In [23] and [34], change impact analysis is done by generating Multi-Terminal Binary Decision Diagrams (MTBDDs) for policies, and the comparing them. In my analysis approach, I transformed the analysis query to both *PBS* and *SAT* instances, and concluded the effectiveness of the approach to convert the analysis query to a PBS instance through empirical analysis. To my knowledge, this is the first analysis approach that transforms an analysis query to a PBS instance.

8.3 Future Work

The following are some future directions of this work.

I have made some assumptions regarding analysing policies that are local only. I do not support the dynamic retrieval of policies located at a URL. It is possible to explore further

in this direction, to extend the applicability of the analysis technique proposed.

One possible future work would be to implement the analysis framework in a small-scale access control system and study the usage of the framework in such a system. If the access control system uses XACML to specify its policies, adapting our policy language to support policies in the system will be easy. Otherwise, some changes may need to be done to the policy language and the compilation algorithm in our framework.

Another possible direction can be to extend the expressiveness of our policy language, and hence extend the spectrum of access control policies that can be analysed using the *DIC* query. One strategy is to include much complex constructs like boolean combinations of request predicates in the $\mathcal{COND}$ type of our abstract syntax.

Bibliography

[1] Borg *PBS* Solver. URL `http://nn.cs.utexas.edu/pages/research/borg/`.

[2] zChaff *SAT* Solver. URL `http://www.princeton.edu/~chaff/zchaff.html`.

[3] GRASP *SAT* Solver. URL `http://vlsicad.eecs.umich.edu/BK/Slots/cache/sat.inesc.pt/~jpms/grasp/`.

[4] MiniSat *SAT* Solver, . URL `http://minisat.se/MiniSat.html`.

[5] MiniSat+ *PBS* Solver, . URL `http://minisat.se/MiniSat+.html`.

[6] PBS v2.1 *PBS* Solver. URL `http://www.aloul.net/Tools/pbs/`.

[7] SAT4J Core *SAT* Solver, . URL `http://www.sat4j.org/products.php#core`.

[8] SAT4J Pseudo *PBS* Solver, . URL `http://www.sat4j.org/products.php#pseudo`.

[9] Siege *SAT* Solver. URL `http://www.cs.sfu.ca/research/groups/CL/software/siege/`.

[10] Claudio Agostino Ardagna, Sabrina Capitani di Vimercati, Tyrone Grandison, Sushil Jajodia, and Pierangela Samarati. Regulating Exceptions in Healthcare Using Policy Spaces. In *Data and Applications Security XXII*, volume 5094 of *Lecture Notes in Computer Science*, pages 254–267, London, UK, 2008. Springer.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[12] Moritz Y. Becker. Specification and Analysis of Dynamic Authorization Policies. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 203–217, Port Jefferson, New York, USA, 2009. IEEE Computer Society.

[13] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An Algebra for Composing Access Control Policies. *ACM Transactions on Information and System Security*, 5(1):1–35, February 2002.

[14] Glenn Bruns and Michael Huth. Access-Control Policies via Belnap Logic: Effective and Efficient Composition and Analysis. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, pages 163–176, Carnegie Mellon University, Pittsburgh - USA, 2008. IEEE Computer Society.

[15] Glenn Bruns and Michael Huth. Access Control via Belnap logic: Intuitive, Expressive, and Analyzable Policy Composition. *ACM Transactions on Information and System Security*, 14(1):9:1–9:27, June 2011.

[16] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, Shaker Heights, Ohio, USA, 1971. ACM.

[17] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[18] Jason Crampton and Michael Huth. An Authorization Framework Resilient to Policy Evaluation Failures. In *Proceedings of the 15th European Symposium on Research in Computer Security*, ESORICS'10, pages 472–487, Athens, Greece, 2010. Springer.

[19] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[20] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of ACM*, 5(7):394–397, July 1962.

[21] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lec-*

ture Notes in Computer Science, pages 502–518, Santa Margherita Ligure, Italy, 2004. Springer.

[22] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1–4):1–26, March 2006.

[23] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 196–205, St. Louis, Missouri, USA, 2005. ACM.

[24] Holger H. Hoos and Thomas Statzle. *Stochastic Local Search : Foundations & Applications (The Morgan Kaufmann Series in Artificial Intelligence)*. Morgan Kaufmann, Burlington, Massachusetts, USA, 1st edition, 2004.

[25] Graham Hughes and Tevfik Bultan. Automated Verification of Access Control Policies Using a SAT Solver. *International Journal of Software Tools for Technology Transfer*, 10(6):503–520, December 2008.

[26] Michael R. A. Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[27] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic Error Finding in Access-Control Policies. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 163–174, Chicago, IL, USA, 2011. ACM.

[28] Vahid R. Karimi. Formal Analysis of Access Control Policies for Pattern-Based Business Processes. In *Proceedings of the 2009 World Congress on Privacy, Security, Trust and the Management of e-Business*, CONGRESS '09, pages 239–242, Saint John, New Brunswick, Canada, 2009. IEEE Computer Society.

[29] Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of the Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, Yorktown Heights, New York, USA, 1972. Plenum Press, New York.

[30] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 677–686, Banff, Alberta, Canada, 2007. ACM.

[31] B. W. Lampson. Dynamic protection structures. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 27–38, Las Vegas, Nevada, 1969. ACM.

[32] Ninghui Li, Qihua Wang, Wahbeh Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access Control Policy Combinination: Theory Meets Practice. In *Proceedings of the 14th ACM symposium on Access Control Models And Technologies*, SACMAT '09, Stresa, Italy, 2009. ACM.

[33] Dan Lin, Prathima Rao, Elisa Bertino, and Jorge Lobo. An Approach to Evaluate Policy Similarity. In *Proceedings of the 12th ACM Symposium on Access Control Models And Technologies*, SACMAT '07, pages 1–10, Nice-Sophia Antipolis, France, 2007. ACM.

[34] Dan Lin, Prathima Rao, Elisa Bertino, Ninghui Li, and Jorge Lobo. EXAM: a comprehensive environment for the analysis of access control policies. *International Journal of Information Security*, 9(4):253–273, August 2010.

[35] Vasco M. Manquinho and João P. Marques Silva. On Using Cutting Planes in Pseudo-Boolean Optimization. *Journal on Satisfiability, Boolean Modeling and Computation*, 2 (1–4):209–219, March 2006.

[36] Evan Martin and Tao Xie. Automated Test Generation for Access Control Policies via

Change-Impact Analysis. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, SESS '07, page 5, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[37] Fabio Massacci and Nicola Zannone. A model-driven approach for the specification and analysis of access control policies. In *Proceedings of On the Move to Meaningful Internet Systems: OTM 2008, OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II*, OTM '08, pages 1087–1103, Monterrey, Mexico, 2008. Springer.

[38] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. Technical report, OASIS Access Control TC, 2005. URL `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[39] Qun Ni, Elisa Bertino, and Jorge Lobo. D-Algebra for Composing Access Control Policy Decisions. In *Proceedings of the 2009 ACM Symposium on Information, Computer, and Communications Security*, ASIACCS '09, Sydney, Australia, 2009. ACM.

[40] Behrooz Parhami. Voting Algorithms. *IEEE Transactions on Reliability*, 43(4):617–629, December 1994.

[41] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1st edition, 1977.

[42] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, August 1996.

[43] Thomas J. Schaefer. The Complexity of Satisfiability Problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, San Diego, California, USA, 1978. ACM.

[44] Bart Selman and Henry A. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *Proceedings of the 11th National Conference on Artificial Intelligence*, AAAI'93, pages 46–51, Washington, D.C., 1993. AAAI Press / The MIT Press.

[45] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A Hybrid Pseudo-Boolean SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1–4):165–189, March 2006.

[46] Scott D. Stoller, Ping Yang, C R. Ramakrishnan, and Mikhail I. Gofman. Efficient Policy Analysis for Administrative Role Based Access Control. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 445–455, Alexandria, Virginia, USA, 2007. ACM.

[47] G. S. Tseitin. On the Complexity of Derivations in the Propositional Calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[48] Joachim P. Walser. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 269–274, Providence, Rhode Island, 1997. AAAI Press.

[49] Dianxiang Xu, Lijo Thomas, Michael Kent, Tejeddine Mouelhi, and Yves Le Traon. A Model-Based Approach to Automated Testing of Access Control Policies. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 209–218, Newark, New Jersey, USA, 2012. ACM.

[50] E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. In *Proceedings of the 2005 IEEE International Conference on Web Services*, ICWS'05, pages 2 vol. (xxxiii+856), Orlando, Florida, USA, 2005. IEEE Computer Society.

Appendix A

DIC2SAT Compilation Algorithm

In this chapter I give the pseudocode of the *DIC2SAT* algorithm.

A.1 *DIC2SAT* description

The compilation of the *DIC* query into a *SAT* instance is given by the *DIC2SAT* algorithm. A description of the *DIC2SAT* algorithm's precondition, inputs and output follows.

Input *DIC2SAT* takes three inputs.

- *pol*: An access control policy that is an instance of the syntactic category $\mathcal{POLICY}$.
- *com*: A label of a sub-policy of *pol*.
- *ds*: A decision set which is a non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Precondition The following conditions on inputs *pol* and *com* should hold.

$$com \sqsubseteq \textsc{label}(pol)$$

The condition given is satisfied iff, *com* is the label of one of the sub-policies of *pol*.

Output *DIC2SAT* returns a propositional formula ϕ, such that the query *DIC*(*pol*, *com*, *ds*) is satisfiable iff ϕ is satisfiable.

A.2 Algorithm *DIC2SAT*

procedure *DIC2SAT*(*pol*, *com*, *ds*)

1. $l :=$LABEL(*pol*)

switch(pol) {

2. **case** p:

3. **if** $\boldsymbol{p} \in ds$ **then**

4. $\phi := \top$

else

5. $\phi := \bot$

6. **case** d:

7. **if** $\boldsymbol{d} \in ds$ **then**

8. $\phi := \top$

else

9. $\phi := \bot$

10. **case** $rp \rightarrow pol^{'}$:

11. Let rp be the boolean variable that encodes the truth value of the satisfiability of the condition rp.

12. $\phi := (rp \wedge \boldsymbol{DIC2SAT}\ (pol^{'}, com^{'} = (com = l)\ ?\ \textsc{label}(pol^{'})\ :\ com,\ ds)\ )$

13. **if** $(\ l = com)\ and\ (\boldsymbol{n} \in ds)$ **then**

14. $\phi := \phi \vee \neg rp$

/*$pol = \texttt{pca}(pol_1, pol_2, \ldots, pol_k)$*/

15. **case** $\texttt{po}(pol_1, pol_2, \ldots, pol_k)$:

16. **if** $l = com$ **then**

17. $\phi := \bot$

18. **if** $(\boldsymbol{p} \in ds)$ **then**

19. $\phi := \phi \vee\ OneOf\ ((pol_1, pol_2, \ldots, pol_k),\{\boldsymbol{p}\})$

20. **if** $(\boldsymbol{d} \in ds)$ **then**

21. $\phi := \phi \vee \Big(\big(\neg OneOf((pol_1, pol_2, \ldots, pol_k),\{\boldsymbol{p}\})\big)$
$\wedge\ OneOf(\ (pol_1, pol_2, \ldots, pol_k),\{\boldsymbol{d}\})\Big)$

22. **if** ($\boldsymbol{i} \in ds$) **then**

23. $\phi \;:=\; \phi \vee \Big(\big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}, \boldsymbol{d}\}) \big)$
$\wedge\; \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{i}\}) \Big)$

24. **if** ($\boldsymbol{n} \in ds$) **then**

25. $\phi \;:=\; \phi \vee \big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \big)$

else /* $\mathit{com} \sqsubset l$ */

26. Let j be the index in $\{1, \ldots, k\}$ such that $\mathit{com} \sqsubseteq \textsc{label}(\mathit{pol}_j)$

27. $\phi \;:=\; \big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_{j-1}), \{\boldsymbol{p}\}) \big) \wedge \boldsymbol{DIC2SAT}(\mathit{pol}_j, \mathit{com}, \mathit{ds})$

28. **case** `do`$(\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k)$:

29. **if** $l = \mathit{com}$ **then**

30. $\phi \;:=\; \bot$

31. **if** ($\boldsymbol{d} \in ds$) **then**

32. $\phi \;:=\; \phi \vee\; \mathit{OneOf}\; ((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}\})$

33. **if** ($\boldsymbol{i} \in ds$) **then**

34. $\phi \;:=\; \phi \vee \Big(\big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}\}) \big)$
$\wedge\; \mathit{OneOf}(\; (\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{i}\}) \Big)$

35. **if** ($\boldsymbol{p} \in ds$) **then**

36. $\phi \;:=\; \phi \vee \Big(\big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}, \boldsymbol{i}\}) \big)$
$\wedge\; \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}\}) \Big)$

37. **if** ($\boldsymbol{n} \in ds$) **then**

38. $\phi \;:=\; \phi \vee \big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \big)$

else /* $\mathit{com} \sqsubset l$ */

39. Let j be the index in $\{1, \ldots, k\}$ such that $\mathit{com} \sqsubseteq \textsc{label}(\mathit{pol}_j)$

40. $\phi \;:=\; \big(\neg \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_{j-1}), \{\boldsymbol{d}\}) \big) \wedge \boldsymbol{DIC2SAT}(\mathit{pol}_j, \mathit{com}, \mathit{ds})$

41. **case** `fa`$(\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k)$:

42. **if** $l = \mathit{com}$ **then**

43. $\phi := \bot$

44. **if** $(\boldsymbol{p} \in ds)$ **then**

45. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{p}\}) \big) \Big)$

46. **if** $(\boldsymbol{d} \in ds)$ **then**

47. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{d}\}) \big) \Big)$

48. **if** $(\boldsymbol{i} \in ds)$ **then**

49. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(F\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{i}\}) \big) \Big)$

50. **if** $(\boldsymbol{n} \in ds)$ **then**

51. $\phi := \phi \vee \big(\neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \big)$

else /* $com \sqsubset l$ */

52. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

53. $\phi := F\text{-}app(\ (pol_1, pol_2, \ldots, pol_j),\ com,\ ds)$

54. **case** `oa`$(pol_1, pol_2, \ldots, pol_k)$:

55. **if** $l = com$ **then**

56. $\phi := \bot$

57. **if** $(\boldsymbol{p} \in ds)$ **then**

58. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(O\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{p}\}) \big) \Big)$

59. **if** $(\boldsymbol{d} \in ds)$ **then**

60. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(O\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{d}\}) \big) \Big)$

61. **if** $(\boldsymbol{i} \in ds)$ **then**

62. $\phi := \phi \vee \Big(\bigvee_{j=1}^{k} \big(O\text{-}app((pol_1, pol_2, \ldots, pol_j), \textsc{label}(pol_j), \{\boldsymbol{i}\}) \big) \Big)$
$\bigvee \Big(\bigvee_{p=1}^{k-1} \bigvee_{q=p+1}^{k} \big(DEC2SAT(pol_p, \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \wedge DEC2SAT(pol_q, \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \big) \Big)$

63. **if** $(\boldsymbol{n} \in ds)$ **then**

64. $\phi := \phi \vee \big(\neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}) \big)$

else /* $com \sqsubset l$ */

65. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

66. ϕ := **DIC2SAT**(pol_j, com, ds)

67. **case** `smv` ($pol_1, pol_2, \ldots, pol_k$):

68. $\phi := \bot$

69. **if** ($\boldsymbol{p} \in ds$) *or* ($\boldsymbol{d} \in ds$) **then**

70. Let r be the number of elements in ($pol_1, pol_2, \ldots, pol_k$).

71. $a := \lfloor \frac{r}{2} \rfloor + 1$

72. Let C_a be a set containing all $\binom{r}{a}$ combinations of policies in the ($pol_1, pol_2, \ldots, pol_k$).

73. **for every** i **in** $\{1, 2, \ldots, a-1\}$ **do**

74. Let C_i be a set containing all $\binom{r}{i}$ combinations of policies in the ($pol_1, pol_2, \ldots, pol_k$).

75. **for every** $c_j \in C_i$ **do**

76. $s = \{pol_1, pol_2, \ldots, pol_k\} \setminus c_j$

77. $l = size(s)$

78. **for every** $m \in \{i, i+1, \ldots, l\}$ **do**

79. Let D_{j_m} be a set containing all $\binom{l}{m}$ combinations of s.

80. $D_j = D_{j_i} \cup D_{j_{i+1}} \cup \cdots \cup D_{j_l}$

81. **if** $\boldsymbol{p} \in ds$ **then**

82. **for every** combination $c_j \in C_a$ **do**

83. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{p}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{p}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_a, \{\boldsymbol{p}\})$

84. $\phi_{C_a} := \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

85. **for every** i **in** $\{1, 2, \ldots, a-1\}$ **do**

86. **for every** combination $c_j \in C_i$ **do**

87. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{p}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{p}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_i, \{\boldsymbol{p}\})$

88. **for every** combination $c_y \in D_j$ **do**

89. $\phi_y := DEC2SAT(p_1, \{\boldsymbol{d}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{d}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_o, \{\boldsymbol{d}\})$

90. $\phi_{D_j} := \neg\phi_1 \bigwedge \neg\phi_2 \bigwedge \cdots \bigwedge \neg\phi_n$

91. $\phi_j := \phi_j \bigwedge \phi_{D_j}$

92. $\phi_i := \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

93. $\phi := \phi \bigvee \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_a$

else

94. **for every** combination $c_j \in C_a$ **do**

95. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{d}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{d}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_a, \{\boldsymbol{d}\})$

96. $\phi_{C_a} := \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

97. **for every** i **in** $\{1, 2, \ldots, a-1\}$ **do**

98. **for every** combination $c_j \in C_i$ **do**

99. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{d}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{d}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_i, \{\boldsymbol{d}\})$

100. **for every** combination $c_y \in D_j$ **do**

101. $\phi_y := DEC2SAT(p_1, \{\boldsymbol{p}\})$
$\bigwedge DEC2SAT(p_2, \{\boldsymbol{p}\})$
$\bigwedge \cdots \bigwedge DEC2SAT(p_o, \{\boldsymbol{p}\})$

102. $\phi_{D_j} := \neg\phi_1 \bigwedge \neg\phi_2 \bigwedge \cdots \bigwedge \neg\phi_n$

103. $\phi_j := \phi_j \bigwedge \phi_{D_j}$

104. $\phi_i := \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

105. $\phi := \phi \bigvee \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_a$

106. **if** $\boldsymbol{i} \in ds$ **then**

107. $\phi := \phi \bigvee \big(\neg DEC2SAT(pol, \{\boldsymbol{p}, \boldsymbol{d}\}) \bigwedge OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\})\big)$

108. **if** $\boldsymbol{n} \in ds$ **then**

109. $\phi := \phi \vee \big(\neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\})\big)$

110 **case** amv$(pol_1, pol_2, \ldots, pol_k)$:

111. $\phi := \bot$

112. **if** $(\boldsymbol{p} \in ds)$ *or* $(\boldsymbol{d} \in ds)$ **then**

113. Let r be the number of elements in $\{pol_1, pol_2, \ldots, pol_k\}$

114. $a := \lfloor \frac{r}{2} \rfloor + 1$

115. Let C be a set containing all $\binom{r}{a}$ combinations of policies in the set $\{pol_1, pol_2, \ldots, pol_k\}$.

116. Let n be the number of combinations in set C.

117. **if** $\boldsymbol{p} \in ds$ **then**

118. **for every** combination $c_j \in C$ **do**

/* such that every $c = (p_1, p_2, \ldots, p_a)$, where, each $p_i \in \{pol_1, pol_2, \ldots, pol_k\}$ and $1 \leq i \leq a$ */

119. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{p}\}) \bigwedge DEC2SAT(p_2, \{\boldsymbol{p}\}) \bigwedge \cdots \bigwedge DEC2SAT(p_a, \{\boldsymbol{p}\})$

120. $\phi := \phi \bigvee \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

121. **if** $\boldsymbol{d} \in ds$ **then**

122. **for every** combination $c_j \in C$ **do**

/* such that every $c = (p_1, p_2, \ldots, p_a)$, where, each $p_i \in \{pol_1, pol_2, \ldots, pol_k\}$ and $1 \leq i \leq a$ */

123. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{d}\})$

$\bigwedge DEC2SAT(p_2, \{\boldsymbol{d}\})$

$\bigwedge \cdots \bigwedge DEC2SAT(p_a, \{\boldsymbol{d}\})$

124. $\phi := \phi \bigvee \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

125. **if** $\boldsymbol{i} \in ds$ **then**

126. $\phi := \phi \bigvee \big(\neg DEC2SAT(pol, \{\boldsymbol{p}, \boldsymbol{d}\}) \bigwedge OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\})\big)$

127. **if** $\boldsymbol{n} \in ds$ **then**

128. $\phi := \phi \vee \big(\neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\})\big)$

129**case** `spmv`$(pol_1, pol_2, \ldots, pol_k)$:

130. $\phi := \bot$

131. **if** $(\boldsymbol{p} \in ds)$ **then**

132. Let r be the number of elements in $\{pol_1, pol_2, \ldots, pol_k\}$

133. $a := \lfloor \frac{2 \times r}{3} \rfloor + 1$

134. Let C be a set containing all $\binom{r}{a}$ combinations of policies in the set $\{pol_1, pol_2, \ldots, pol_k\}$.

135. Let n be the number of combinations in set C.

136. **for every** combination $c_j \in C$ **do**

/* such that every $c = (p_1, p_2, \ldots, p_a)$, where, each $p_i \in \{pol_1, pol_2, \ldots, pol_k\}$ and $1 \leq i \leq a$ */

137. $\phi_j := DEC2SAT(p_1, \{\boldsymbol{p}\})$

$\bigwedge DEC2SAT(p_2, \{\boldsymbol{p}\})$

$\bigwedge \cdots \bigwedge DEC2SAT(p_a, \{\boldsymbol{p}\})$

138. $\phi := \phi \bigvee \phi_1 \bigvee \phi_2 \bigvee \cdots \bigvee \phi_n$

139. **if** $\boldsymbol{d} \in ds$ **then**

140. $\phi := \phi \bigvee \big(\neg DEC2SAT(pol, \{\boldsymbol{p}\}) \bigwedge OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{d}\})\big)$

141. **if** $\boldsymbol{i} \in ds$ **then**

142. $\phi := \phi \bigvee \big(\neg DEC2SAT(pol, \{\boldsymbol{p}\}) \bigwedge \neg OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{d}\})$

$\bigwedge OneOf((pol_1, pol_2, \ldots, pol_k),\ \{\boldsymbol{p},\boldsymbol{i}\})\big)$

143. **if** $(\boldsymbol{n} \in ds)$ **then**

144. $\phi := \phi \vee \big(\neg OneOf((pol_1, pol_2, \ldots, pol_k),\{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\})\big)$

}

145**return** ϕ

end procedure

A.3 Sub-Formulae

$$
\begin{aligned}
OneOf((pol_1, pol_2, \ldots, pol_k),ds) &= DEC2SAT(pol_1,\ ds) \\
&\quad \vee\ DEC2SAT(pol_2,ds) \vee \cdots \\
&\quad \vee\ DEC2SAT(pol_k,\ ds) \\
F\text{-}app((pol_1, pol_2, \ldots, pol_j),com,ds) &= \big(\textstyle\bigwedge_{i=1}^{j-1} DEC2SAT(pol_i, \{\boldsymbol{n}\})\big) \\
&\quad \wedge \boldsymbol{DIC2SAT}(pol_j,\ com,\ ds) \\
O\text{-}app((pol_1, pol_2, \ldots, pol_k),\ com,\ ds) &= F\text{-}app((pol_1, pol_2, \ldots, pol_j),\ com,\ ds) \\
&\quad \wedge \big(\textstyle\bigwedge_{i=j+1}^{k} DEC2SAT(pol_i, \{\boldsymbol{n}\})\big)
\end{aligned}
$$

Appendix B

DIC2PBS Compilation Algorithm

In this chapter I give the pseudocode of the *DIC2PBS* algorithm.

B.1 *DIC2PBS* description

The compilation of the *DIC* query using the *PBS* encoding is given by the *DIC2PBS* algorithm. A description of *DIC2PBS* algorithm's inputs, precondition, output, and postconditions follows.

Input *DIC2PBS* takes three inputs.

- *pol*: An access control policy that is an instance of the syntactic category $\mathcal{POLICY}$.
- *com*: The label of a sub-policy of *pol*.
- *ds*: A decision set which is a non-empty subset of $\{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$.

Precondition The following condition on inputs *com* and *pol* should hold.

$$com \sqsubseteq \textsc{label}(pol)$$

The condition given is satisfied if and only if *com* identifies a sub-policy of *pol*.

Output *DIC2PBS* returns a $\langle CS, x\rangle$ pair. Here *CS* is a set of pseudo boolean constraints, and x is a pseudo boolean variable. This output satisfies the following postconditions.

Postcondition The following are the two postconditions.

(a) The output $\langle CS, x\rangle$ is such that

- *CS* is always satisfiable; let *RV* be the set of variables in *CS* that correspond to request predicates; let *AV* be the set of other variables in *CS*;

- $x \in AV$;
- Suppose σ is a variable assignment for the request predicates in *pol*. If σ satisfies the input *DIC* query, then every extension of σ to a satisfying assignment of *CS* will assign 1 to x. If σ does not satisfy the input *DIC* query, then every extension of σ to a satisfying assignment of *CS* will assign 0 to x. Consequently, the input *DIC* query is satisfiable iff there is a variable assignment for *CS* for which x is assigned 1.

(b) Let CS' be the set of pseudo boolean constraints such that

$$CS' = CS \bigcup \{x = 1\}, \text{ where } \langle CS, x \rangle \text{ is the output of } DIC2PBS.$$

The satisfiability of this set of pseudo boolean constraints CS' reflects the satisfiability of the *DIC* query on the same set of input values.

B.2 Basic Subroutines

The following are some of the basic subroutines that I utilize in the $DIC2PBS$ algorithm.

B.2.1 *Uniform*

This routine takes as input (a) a set of policies that are instances of the syntactic category $\mathcal{POLICY}$, and (b) a singleton set $\{dec\}$, where $dec \in \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{n}, \boldsymbol{i}\}$. The routine returns a set of *ICS*s. An *ICS* is generated for each policy pol_j in the input policy set, such that this *ICS* is satisfied iff the query $DEC(pol_i, \{dec\})$ is satisfiable.

procedure $Uniform(\{pol_1, pol_2, \ldots, pol_k\}, \{\text{dec}\})$

1. Let S be an empty set.
2. **for every** pol_i **in** $(pol_1, pol_2, \ldots, pol_k)$ **do**
3. $\quad S := S \bigcup DEC2PBS(pol_i, \{dec\})$

4. **return** S

end procedure

B.2.2 `AtLeast`

This routine takes as input (a) a set of *ICS*s s, and (b) a positive integer min. It returns an *ICS* $\langle OS, y\rangle$, such that when $y = 1$, OS is satisfied iff at least min of the input *ICS*s are satisfied.

procedure `AtLeast`(s, min)

1. Let OS be an empty constraint set.
2. Let y be a fresh pseudo boolean variable.
3. $n := size(s)$
4. **for every** $\langle CS_i, x_i\rangle$ **in** s **do**
5. $\quad OS := OS \bigcup CS_i$
6. $OS := OS \bigcup \{\sum_{i=1}^{n} x_i \geq min \times y\} \bigcup \{\sum_{i=1}^{n} x_i \leq y \times n + (1 - y) \times (min - 1)\}$
7. **return** $\langle OS, y\rangle$

end procedure

B.2.3 `AtMost`

This routine takes as input (a) a set of *ICS*s s, and (b) a positive integer max. It returns an *ICS* $\langle OS, y\rangle$, such that when $y = 1$, OS is satisfied iff at most max of the input *ICS*s are satisfied.

procedure `AtMost`(s, max)

1. Let OS be an empty constraint set.
2. Let y be a fresh pseudo boolean variable.

3. $n := size(s)$

4. **for every** $\langle CS_i, x_i \rangle$ **in** s **do**

5. $\quad OS := OS \bigcup CS_i$

6. $OS := OS \bigcup \{\sum_{i=1}^{n} x_i + y \geq 1 + (1 - y) \times max\} \bigcup \{\sum_{i=1}^{n} x_i \leq max \times y + (1 - y) \times n\}$

7. **return** $\langle OS, y \rangle$

end procedure

B.2.4 Boolean Combinations of *PBS* Encodings

The following routines demonstrate how boolean combinations are performed on *ICS*s, using the `AtLeast` and `AtMost` routines. Each of the following three routines takes a set of *ICS*s as input, and returns a single *ICS* as output.

procedure `OR`(s)

1. **return** `AtLeast`$(s, 1)$

end procedure

procedure `AND`(s)

1. Let n be the number of *ICS*s in s.

2. **return** `AtLeast`(s, n)

end procedure

procedure `NOT`(s)

1. **return** `AtMost`$(s, 0)$

end procedure

B.2.5 *PBS* Encodings of Boolean Values

The routine *ForceZero* gives the *PBS* encoding of the boolean *False* value, and the routine *ForceOne* gives the *PBS* encoding of the boolean *True* value. Both these routines take a pseudo boolean variable as input, and return an *ICS* as output.

procedure *ForceZero*(x)

1. Let f be a fresh pseudo boolean variable.
2. **return** $\langle\{f - 2x \geq 0\}, x\rangle$

end procedure

procedure *ForceOne*(x)

1. Let t be a fresh pseudo boolean variable.
2. **return** $\langle\{t + 2x \geq 2\}, x\rangle$

end procedure

B.3 Algorithm *DIC2PBS*

procedure *DIC2PBS*(*pol*, *com*, *ds*)

1. $l :=$ LABEL(*pol*)
2. Let *CS* be an empty set and x be a fresh pseudo boolean variable.

   switch(*pol*) {
3. **case** p:
4. **if** $\boldsymbol{p} \in ds$ **then**
5. $\langle CS, x\rangle :=$ *ForceOne*(x)

   else
6. $\langle CS, x\rangle :=$ *ForceZero*(x)

7. **case** d:

8. **if** $\boldsymbol{d} \in ds$ **then**

9. $\langle CS, x\rangle := \ ForceOne(x)$

else

10. $\langle CS, x\rangle \ := ForceZero(x)$

11. **case** $rp \rightarrow pol^{'}$:

12. Let r be the pseudo boolean variable encoding the truth value of the satisfiability of condition rp, and x_1 be a fresh pseudo boolean variable.

13. **if** $l = com$ **then**

14. $\langle CS, x\rangle := \texttt{AND}\big(\langle\{r - x_1 = 0\}, x_1\rangle, DEC2PBS(pol^{'}, ds)\big)$

else

15. $\langle CS, x\rangle := \texttt{AND}\big(\langle\{r - x_1 = 0\}, x_1\rangle, \boldsymbol{DIC2PBS}\ (pol^{'}, com, ds\ )\big)$

16. **if** $\boldsymbol{n} \in ds$ **then**

17. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle,\ \langle\{r + x_1 = 1\}, x_1\rangle\big)$

/* $pol = \texttt{pca}(pol_1, pol_2, \ldots, pol_k)$ */

18. **case** $\texttt{po}(pol_1, pol_2, \ldots, pol_k)$:

19. **if** $l = com$ **then**

20. $\langle CS, x\rangle \ := ForceZero(x)$

21. **if** $\boldsymbol{p} \in ds$ **then**

22. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle,\ OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}\})\big)$

23. **if** $\boldsymbol{d} \in ds$ **then**

24. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle,\ \texttt{AND}(\texttt{NOT}(OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}\})),$
$OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{d}\}))\big)$

25. **if** $\boldsymbol{i} \in ds$ **then**

26. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{AND}(\texttt{NOT}(OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}\})),$
$OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{i}\}))\big)$

27. **if** $\boldsymbol{n} \in ds$ **then**

28. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}))\big)$

else /* $\mathit{com} \sqsubset l$ */

29. Let j be the index in $\{1, \ldots, k\}$ such that $\mathit{com} \sqsubseteq \textsc{label}(\mathit{pol}_j)$

30. $\langle CS, x\rangle := \texttt{AND}\big(\texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_{j-1}), \{\boldsymbol{p}\})),$
$\boldsymbol{DIC2PBS}(\mathit{pol}_j, \mathit{com}, \mathit{ds})\big)$

31. **case** $\texttt{do}(\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k)$:

32. **if** $l = \mathit{com}$ **then**

33. $\langle CS, x\rangle := \mathit{ForceZero}(x)$

34. **if** $\boldsymbol{p} \in ds$ **then**

35. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle,\ \mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}\})\big)$

36. **if** $\boldsymbol{i} \in ds$ **then**

37. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle,\ \texttt{AND}(\texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}\})),$
$\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{i}\}))\big)$

38. **if** $\boldsymbol{p} \in ds$ **then**

39. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{AND}(\texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{d}, \boldsymbol{i}\})),$
$\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}\}))\big)$

40. **if** $\boldsymbol{n} \in ds$ **then**

41. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}))\big)$

else /* $\mathit{com} \sqsubset l$ */

42. Let j be the index in $\{1, \ldots, k\}$ such that $\mathit{com} \sqsubseteq \textsc{label}(\mathit{pol}_j)$

43. $\langle CS, x\rangle := \texttt{AND}\big(\texttt{NOT}(\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_{j-1}), \{\boldsymbol{d}\})),$
$\boldsymbol{DIC2PBS}(\mathit{pol}_j, \mathit{com}, \mathit{ds})\big)$

44. **case** $\texttt{fa}(\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k)$:

45. **if** $l = \mathit{com}$ **then**

46. $\langle CS, x\rangle := \mathit{ForceZero}(x)$

47. **if** $\boldsymbol{p} \in ds$ **then**

48. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (F-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{p}\}))\big)$

49. **if** $\boldsymbol{d} \in ds$ **then**

50. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (F-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{d}\}))\big)$

51. **if** $\boldsymbol{i} \in ds$ **then**

52. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (F-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{i}\}))\big)$

53. **if** $\boldsymbol{n} \in ds$ **then**

54. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{NOT}(OneOf((pol_1, pol_2, \ldots, pol_k), \{\boldsymbol{p}, \boldsymbol{d}, \boldsymbol{i}\}))\big)$

else /* $com \sqsubset l$ */

55. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq$ LABEL(pol_j)

56. $\langle CS, x\rangle := F\text{-}app((pol_1, pol_2, \ldots, pol_j), com, ds)$

57. **case** $\texttt{oa}(pol_1, pol_2, \ldots, pol_k)$:

58. **if** $l = com$ **then**

59. $\langle CS, x\rangle := ForceZero(x)$

60. **if** $\boldsymbol{p} \in ds$ **then**

61. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (O-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{p}\}))\big)$

62. **if** $\boldsymbol{d} \in ds$ **then**

63. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (O-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{d}\}))\big)$

64. **if** $\boldsymbol{i} \in ds$ **then**

65. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{OR}_{j=1}^{k}$ (O-app ($(pol_1, pol_2, \ldots, pol_j)$, LABEL$(pol_j), \{\boldsymbol{i}\}))$,

$\texttt{OR}_{p=1}^{k-1}(\texttt{OR}_{q=p+1}^{k}(\texttt{AND}(DEC2PBS(pol_p, \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\}), DEC2PBS(pol_q, \{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\})))))\big)$

66. **if** $\boldsymbol{n} \in ds$ **then**

67. $\langle CS, x\rangle := \texttt{OR}\big(\langle CS, x\rangle, \texttt{NOT}(OneOf((pol_1, pol_2, \ldots, pol_k),\{\boldsymbol{p},\boldsymbol{d},\boldsymbol{i}\}))\big)$

else /* $com \sqsubset l$ */

68. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

69. $\langle CS, x\rangle := \boldsymbol{DIC2PBS}(pol_j, com, ds)$

70. **case** $\texttt{smv}(pol_1, pol_2, \ldots, pol_k)$:

71. **case** $\texttt{amv}(pol_1, pol_2, \ldots, pol_k)$:

72. **case** $\texttt{spmv}(pol_1, pol_2, \ldots, pol_k)$:

$\langle CS, x\rangle := \boldsymbol{DIC2PBS\text{-}MV}(pol, com, ds)$

}

73.**return** CS

end procedure

procedure $\boldsymbol{DIC2PBS\text{-}MV}(pol, com, ds)$

/* $pol = \texttt{pca}(pol_1, pol_2, \ldots, pol_k)$ */

1. $l := \textsc{label}(pol)$

Global:

begin

2. $pol\_set := \{pol_1, pol_2, \ldots, pol_k\ \}$

3. Let k be the number of policies in pol_set.

4. $csp := Uniform(pol\_set, \{\boldsymbol{p}\})$

5. $csd := Uniform(pol\_set, \{\boldsymbol{d}\})$

6. $csi := Uniform(pol\_set, \{\boldsymbol{i}\})$

end

switch(pol) {

7. **case** $\texttt{smv}(pol_1, pol_2, \ldots, pol_k)$:

8. **if** $l = com$ **then**

9. $\langle CS, x\rangle :=$***DIC2PBS-SMV***(pol_set, ds)

else $/* \ com \sqsubset l \ */$

10. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

11. $\langle CS, x\rangle :=$ ***DIC2PBS***(pol_j, com, ds)

12. **case** $\texttt{amv}(pol_1, pol_2, \ldots, pol_k)$:

13. **if** $l = com$ **then**

14. $\langle CS, x\rangle :=$***DIC2PBS-AMV***(pol_set, ds)

else $/* \ com \sqsubset l \ */$

15. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

16. $\langle CS, x\rangle :=$ ***DIC2PBS***(pol_j, com, ds)

17. **case** $\texttt{spmv}(pol_1, pol_2, \ldots, pol_k)$:

18. **if** $l = com$ **then**

19. $\langle CS, x\rangle :=$***DIC2PBS-SPMV***(pol_set, ds)

else $/* \ com \sqsubset l \ */$

20. Let j be the index in $\{1, \ldots, k\}$ such that $com \sqsubseteq \textsc{label}(pol_j)$

21. $\langle CS, x\rangle :=$ ***DIC2PBS***(pol_j, com, ds)

}

22. **return** $\langle CS, x\rangle$

end procedure

procedure ***DIC2PBS-SMV***(pol_set, ds)

1. Let CS_1, CS_2, CS_3 and CS_4 be empty constraint sets.
2. Let y_1, y_2, y_3 and y_4 be fresh pseudo boolean variables.
3. **if** $\boldsymbol{p} \in ds$ **then**

4. **for every** $\langle CS_{p_i}, x_{p_i} \rangle$ **in** csp **do**

5. $\quad CS_1 := CS_1 \bigcup CS_{p_i}$

6. **for every** $\langle CS_{d_i}, x_{d_i} \rangle$ **in** csd **do**

7. $\quad CS_1 := CS_1 \bigcup CS_{d_i}$

8. $\langle CS_1, y_1 \rangle := \langle \{ CS_1 \bigcup \{ \sum_{j=1}^{k} x_{p_j} - \sum_{j=1}^{k} x_{d_j} \ \geq y_1 + (1 - y_1) \times -k \} \bigcup$
$\{ \sum_{j=1}^{k} x_{p_j} - \sum_{j=1}^{k} x_{d_j} \ \leq k \times y_1 \},\ y_1 \rangle$

else

9. $\langle CS_1, y_1 \rangle \ := \ ForceZero(y_1)$

10. **if** $\boldsymbol{d} \in ds$ **then**

11. **for every** $\langle CS_{p_i}, x_{p_i} \rangle$ **in** csp **do**

12. $\quad CS_2 := CS_2 \bigcup CS_{p_i}$

13. **for every** $\langle CS_{d_i}, x_{d_i} \rangle$ **in** csd **do**

14. $\quad CS_2 := CS_2 \bigcup CS_{d_i}$

15. $\langle CS_2, y_2 \rangle := \langle \{ CS_2 \bigcup \{ \sum_{j=1}^{k} x_{d_j} - \sum_{j=1}^{k} x_{p_j} \ \geq y_2 + (1 - y_2) \times -k \} \bigcup$
$\{ \sum_{j=1}^{k} x_{d_j} - \sum_{j=1}^{k} x_{p_j} \ \leq k \times y_2 \},\ y_2 \rangle$

else

16. $\langle CS_2, y_2 \rangle \ := ForceZero(y_2)$

17. **if** $\boldsymbol{i} \in ds$ **then**

18. $\langle CS_3, y_3 \rangle :=$ `AND` (`NOT` (***DIC2PBS-SMV***$(pol\_set, \{\boldsymbol{p}, \boldsymbol{d}\})$),
`AtLeast` $(csp \bigcup csd \bigcup csi, 1))$

else

19. $\langle CS_3, y_3 \rangle \ := ForceZero(y_3)$

20. **if** $\boldsymbol{n} \in ds$ **then**

21. $\langle CS_4, y_4 \rangle :=$ `AtMost`$(csp \bigcup csd \bigcup csi, 0)$

else

22. $\langle CS_4, y_4 \rangle \ := ForceZero(y_4)$

23. **return** OR $(\langle CS_1, y_1\rangle, \langle CS_2, y_2\rangle, \langle CS_3, y_3\rangle, \langle CS_4, y_4\rangle)$

end procedure

procedure *DIC2PBS-AMV* (pol_set, ds)

1. $a := \lfloor \frac{k}{2} \rfloor + 1$
2. Let CS_1, CS_2, CS_3 and CS_4 be empty constraint sets.
3. Let y_1, y_2, y_3 and y_4 be fresh pseudo boolean variables.
4. **if** $\boldsymbol{p} \in ds$ **then**
5. $\quad \langle CS_1, y_1\rangle := \texttt{AtLeast}(csp, a)$

   else
6. $\quad \langle CS_1, y_1\rangle := ForceZero(y_1)$
7. **if** $\boldsymbol{d} \in ds$ **then**
8. $\quad \langle CS_2, y_2\rangle := \texttt{AtLeast}(csd, a)$

   else
9. $\quad \langle CS_2, y_2\rangle := ForceZero(y_2)$
10. **if** $\boldsymbol{n} \in ds$ **then**
11. $\quad \langle CS_3, y_3\rangle := \texttt{AtMost}(csp \bigcup csd \bigcup csi, 0)$

    else
12. $\quad \langle CS_3, y_3\rangle := ForceZero(y_3)$
13. **if** $\boldsymbol{i} \in ds$ **then**
14. $\quad \langle CS_4, y_4\rangle := \texttt{AND}(\texttt{AtMost}(csp, a-1),\ \texttt{AtMost}(csd, a-1),$

    $\texttt{AtLeast}(csp \bigcup csd \bigcup csi, 1))$

    else
15. $\quad \langle CS_4, y_4\rangle := ForceZero(y_4)$
16. **return** OR $(\langle CS_1, y_1\rangle, \langle CS_2, y_2\rangle, \langle CS_3, y_3\rangle, \langle CS_4, y_4\rangle)$

end procedure

procedure *DIC2PBS-SPMV* (pol_set, ds)

1. $a := \lfloor \frac{2 \times k}{3} \rfloor + 1$
2. Let CS_1, CS_2, CS_3 and CS_4 be empty constraint sets.
3. Let y_1, y_2, y_3 and y_4 be fresh pseudo boolean variables.
4. **if** $\boldsymbol{p} \in ds$ **then**
5. $\quad \langle CS_1, y_1 \rangle := \texttt{AtLeast}(csp, a)$

   else
6. $\quad \langle CS_1, y_1 \rangle := ForceZero(y_1)$
7. **if** $\boldsymbol{d} \in ds$ **then**
8. $\quad \langle CS_2, y_2 \rangle := \texttt{AND}(\texttt{AtMost}(csp, a-1), \texttt{AtLeast}(csd, 1))$

   else
9. $\quad \langle CS_2, y_2 \rangle := ForceZero(y_2)$
10. **if** $\boldsymbol{n} \in ds$ **then**
11. $\quad \langle CS_3, y_3 \rangle := \texttt{AtMost}(csp \bigcup csd \bigcup csi, 0)$

    else
12. $\quad \langle CS_3, y_3 \rangle := ForceZero(y_3)$
13. **if** $\boldsymbol{i} \in ds$ **then**
14. $\quad \langle CS_4, y_4 \rangle := \texttt{AND}\ (\texttt{AtMost}(csp, a-1),\ \texttt{AtMost}(csd, 0),$

    $\quad \texttt{AtLeast}(csp \bigcup csi, 1))$

    else
15. $\quad \langle CS_4, y_4 \rangle := ForceZero(y_4)$
16. **return** $\texttt{OR}\ (\langle CS_1, y_1 \rangle, \langle CS_2, y_2 \rangle, \langle CS_3, y_3 \rangle, \langle CS_4, y_4 \rangle)$

end procedure

B.4 SubFormulae

$$
\begin{aligned}
\mathit{OneOf}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \mathit{ds}) \quad &= \quad \mathtt{OR}\big(\mathit{DEC2PBS}(\mathit{pol}_1, \mathit{ds}), \\
&\qquad \mathit{DEC2PBS}(\mathit{pol}_2, \mathit{ds}), \ldots, \\
&\qquad \mathit{DEC2PBS}(\mathit{pol}_k, \mathit{ds})\big) \\
\mathit{F\text{-}app}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_j),\ \mathit{com},\ \mathit{ds}) \quad &= \quad \mathtt{AND}\big(\mathtt{OR}_{i=1}^{j-1}\big(\mathit{DEC2PBS}(\mathit{pol}_i, \{n\})\big), \\
&\qquad \boldsymbol{\mathit{DIC2PBS}}\ (\mathit{pol}_j, \mathit{com}, \mathit{ds}\ )\big) \\
\mathit{O\text{-}app}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_k), \mathit{com}, \mathit{ds}) \quad &= \quad \mathtt{AND}\big(\mathit{F\text{-}app}((\mathit{pol}_1, \mathit{pol}_2, \ldots, \mathit{pol}_j), \\
&\qquad \mathit{com}, \mathit{ds}), \\
&\qquad \mathtt{OR}_{i=j+1}^{k}\big(\mathit{DEC2PBS}(\mathit{pol}_i, \{\boldsymbol{n}\})\big)\big)
\end{aligned}
$$

Appendix C

Random Policy Generator Algorithm

A description of the algorithm's inputs and output is detailed below.

Input: This algorithm takes the following two inputs:

- n — a positive integer that denotes the size of the policy to be generated, where $n \geq 3$, and
- m — a positive integer representing the number of distinct request predicates that may occur in the generated policy. Here, $m \geq 1$.

Output: A policy *ranpol*, of size n (denoting the number of policy constructs in this policy). *ranpol* is a PCA Policy, whose `pca` $\in$ {`smv`, `amv`, `spmv`}. The remaining $n-1$ policies are generated by calling the algorithm recursively on smaller values of n. If this policy contains a sub-policy of type Conditional Policy, then the condition of this sub-policy is a request predicate, denoted by a number between 1 and m. The selection of each request predicate from 1 to m by our algorithm, has equal probability.

procedure *Generate_PCAPolicy*(n, m)

1. vot_pca := {`smv`, `amv`, `spmv`}
2. $alg := random(vot\_pca)$
3. $C := Generate\_Composition(n-1)$

   /* C is a sequence of integers denoting a composition of integer $n-1$ */
4. **for every** int_i **in** C **do**
5. $\quad pol_i := Generate\_Policy(int_i, m)$
6. **return** $alg(pol_1, pol_2, \ldots, pol_k)$

   /* k is the length of sequence C */

end procedure

procedure *Generate_Policy*(n, m)

1. $atom := \{\texttt{p}, \texttt{d}\}$
2. $con := \{1, 2, \ldots, m-1, m\}$
3. $all\_pca := \{\texttt{po}, \texttt{do}, \texttt{fa}, \texttt{oa}, \texttt{smv}, \texttt{amv}, \texttt{spmv}\}$
4. **if** $n = 1$ **then**
5. **return** $random(atom)$
6. **else if** $n = 2$ **then**
7. $rp := random(con)$
8. $pol^{'} := random(atom)$
9. **return** $rp \rightarrow pol^{'}$

else /* $n \geq 3$ */

10. $type := random(\{1, 2\})$
11. **if** $type = 1$ **then**
12. $rp := random(con)$
13. $pol^{'} := Generate\_Policy(n-1, m)$
14. **return** $rp \rightarrow pol^{'}$

else $/ * type = 2 * /$

15. $alg := random(all\_pca)$
16. $C := Generate\_Composition(n-1)$
17. **for every** int_i **in** C **do**
18. $pol_i := Generate\_Policy(int_i, m)$
19. **return** $alg(pol_1, pol_2, \ldots, pol_k)$

end procedure

procedure *Generate_Composition*(n)

1. $num := random(\{1, 2, \ldots, 2^{n-1} - 1, 2^{n-1}\})$

2. Let *bin* be the binary representation of *num*.

   /* *bin* has length $n-1$. For shorter *bin* lengths, we prepend 0 to make length of *bin* equal to $n-1$ */

3. $dist := 0$
4. $j = 1$
5. **for every** $digit_i$ **in** *bin* **do**

   /* $1 \leq i \leq n-1$ */

6. **if** $digit_i = 1$ **then**
7. $int_j := i - dist$
8. $dist := i$
9. $j = j + 1$
10. $int_j := n - dist$
11. **return** $(int_1, int_2, \ldots, int_{count})$

    /* If *onecount* indicates the number of 1's in binary *bin*, then *count* = *onecount* + 1 */

end procedure

C.0.1 SubRoutines:

In the above procedures, the routine *random*(s) can be described as follows.

- **Input**: A set of items s.
- **Output**: A single item *sel_item*, selected from s. Each item in s has an equal likelihood of being selected.