A White-Box Policy Analysis and its Efficient Implementation

Jayalakshmi Balasubramaniam Philip W. L. Fong
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{ jbalasub, pwlfong }@ucalgary.ca

ABSTRACT

In policy composition frameworks, such as XACML, composite policies can be formed by the application of policy composition algorithms (PCAs), which combine authorization decisions of component policies. Understanding the behaviour of composite policies is a non-trivial endeavour, but instrumental in the engineering of correct access control policies. Existing policy analyses take a black-box approach, in which the global behaviour of the composite policy is assessed. A black-box approach is useful for detecting the presence of erroneous behaviour, but not particularly useful for locating the source of the error.

In this work, we propose a white-box policy analysis, known as Decision in Context (DIC), that assesses the behaviour of component policies situated in a composite policy. We show that the DIC query can be applied to facilitate policy change impact analysis, break-glass reduction analysis, dead policy identification, as well as the pruning of redundant subpolicies. For generality, the DIC query is defined in an XACML-style policy composition framework that is agnostic of the underlying access control model. The DIC query is implemented via a reduction to either propositional satisfiability (SAT) or pseudo boolean satisfiability (PBS) 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 a more effective encoding, especially for composite policies containing majority-voting PCAs.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access Controls

General Terms

Security, Theory

SACMAT'13, June 12–14, 2013, Amsterdam, The Netherlands.

Keywords

Policy analysis, white-box testing, change impact analysis, break-glass reduction, dead policy, policy pruning, policy composition, XACML, propositional satisfiability, pseudo boolean satisfiability

1. INTRODUCTION

A typical organization is divided into a number of work units. For example, a hospital has different departments, such as cardiology, neurology, anaesthetics, critical care, etc; a bank is composed of units such as legal, financial management, marketing, etc. Each work unit may have a different set of policies governing the disclosure and usage of data. In such an organizational setting, an access control system must (i) consult the policies of the many work units, and (ii) combine, weigh and prioritize their individual authorization decisions in order to reach a final authorization decision. This creates the need for a policy composition framework [25, 10, 26, 21, 8, 11]. A typical policy composition framework, such as XACML [25], offers policy combinators called Policy Combining Algorithms (PCAs) for constructing composite policies out of component policies. These PCAs specify procedures for harmonizing the potentially conflicting decisions of their component policies.

Being able to comprehend and assess the behaviour of a composite policy specified in a policy composition framework is both important and challenging. Important because the formulation of correct policy depends on it. Challenging because of the sheer size of organizational policies, as well as the fact that, in an organizational setting, components policies may undergo constant revisions motivated by changing organizational needs. Various policy analyses have been proposed to alleviate this need [14, 20, 29, 23, 24, 17, 19, 7, 22, 18, 10, 31].

Many of the previously proposed policy analyses [14, 17, 22, 10] are ***black-box analyses***. They assess the global behaviour of a composite policy: e.g., under such and such a condition, would the composite policy return such and such authorization decisions? That is, the composite policy is seen as a black box, in the sense that the concern is on the global behaviour of the entire policy rather than the local behaviour of subpolicies.

Black-box analyses is important, as they help us identify any problematic behaviour of the composite policy. Yet they are not designed for locating the source of error: i.e., determining which subpolicy of the composite policy produces the error. To facilitate the location of error sources, we need analysis techniques that would assess the behaviour

of subpolicies in the execution context of the composite policy. In other words, the composite policy is now seen as a white box. ***White-box analyses*** are analogous to debugging tools. It is the position of this work that we need to develop white-box policy analysis techniques.

Yet the behaviour of subpolicies are difficult to understand due to at least two reasons. First, realistic policy analysis frameworks offer PCAs that have procedural semantics. For example, the First-Applicable PCA of XACML may opportunistically terminate the evaluation of subpolicies when an applicable subpolicy is found. Reasoning about such behaviour is challenging. Second, a breed of PCAs known as majority-voting algorithms have not been given proper treatment in previous work on policy analyses [14, 17, 22, 10]. Inefficient implementation, by way of brute-force encoding, is usually adopted.

To bridge the above gaps, this work proposes a novel white-box policy analysis, as well as corresponding implementation techniques that can gracefully cope with the presence of majority-voting PCAs. Specifically, the contributions of this work are the following.

1. A new policy analysis, Decision in Context (DIC), has been proposed. The DIC analysis can be used for understanding the behaviour of a subpolicy that is situated within a composite policy, when the latter is being evaluated. Specifically, given a subpolicy contained within a composite policy, a DIC query determines (a) whether the subpolicy will be evaluated when the composite policy is evaluated, and (b) the potential authorization decisions that the subpolicy may return. The DIC analysis has a wide range of applications, including that of change impact analysis, reduction of break-glass accesses, dead policy identification, and complex policy pruning. The definition and applications of DIC analysis are discussed in §4.

2. The DIC analysis is defined in terms of a policy composition language called μ-XACML, which is a light-weight, idealized version of XACML 2.0 [25]. Defining DIC in terms of μ-XACML demonstrates that the analysis can be conducted against policy composition frameworks with complex policy combinators, including those that exhibit procedural characteristics as well as those that employ majority-voting schemes. This language is also independent of the choice of access control models, thereby rendering the DIC analysis widely applicable. Details of μ-XACML can be found in §3.

3. We demonstrate how the DIC query can be answered by a reduction to either propositional satisfiability (SAT) or pseudo boolean satisfiability (PBS) instances. While the SAT compilation is now a standard technique for implementing policy analyses, the application of PBS as an encoding technique is a novel contribution of this work. The motivation is to cope with the complexity introduced by majority-voting PCAs. A standard SAT-based compilation would lead to an exponential growth in encoding size, whereas a PBS encoding is much more compact. Details of these compilation can be found in §5 and §6. An empirical study for demonstrating the efficiency of the PBS encoding over the SAT encoding, especially in the presence of majority-voting PCAs, is reported in §7.

2. RELATED WORK

We compare our work to related research in three aspects: (i) policy composition frameworks, (ii) policy analysis and testing, and (iii) policy analysis implementation approaches.

2.1 Policy Composition Frameworks

D-Algebra [26] is an algebraic framework for policy composition. Algebraic operations have been designed for policy composition. A technique has been developed to convert tabular specification of policy combinators into D-Algebra functions. The framework is expressive enough for capturing standard PCAs of XACML. The algebra can also be interpreted in such a way to support majority voting.

The Policy Combining Language (PCL) [21] is another policy composition framework. The language has two variants: one for representing majority-voting PCAs, and the other for standard PCAs. Majority voting PCAs are specified using linear constraints on variables representing the number of policies returning a specific decision. Standard PCAs are represented as finite-state automata. PCL also handles obligations.

Bruns and Huth [10] base their policy composition language on Belnap Logic (a four-valued logic). The ***meet*** and ***join*** operators of the Belnap bilattice are taken as policy combinators. The language is capable of representing every possible four-valued functions (and thus all PCAs). The language is parameterized on abstract access control systems, which abstract away the idiosyncrasies of individual access control model.

The goal of this present work is not to innovate in the design of policy composition languages. Our policy composition language, μ-XACML, is designed only for capturing the essential complexity of XACML 2.0 [25]. It forms the basis for demonstrating the feasibility of defining the DIC analysis for realistic policy composition languages. We also borrowed the idea of abstract access control systems from [10], to render our work applicable to a wide range of access control models.

2.2 Policy Analysis and Testing

Various policy analyses have been proposed previously [20, 29, 24, 19, 7, 18, 31]. The following discusses the ones that are particularly related to the DIC analysis.

A number of types of policy analyses have been presented in [22]: analysis of information about the policy, analysis of information about the structure of the policy, analysis of the decisions returned by a policy, and analysis of the type of access requests to which the policy is applicable. They provide a method to determine, given requests satisfying a request predicate, whether a policy applicable to such requests can return a specific decision. They also provide techniques for analyzing the similarity between two policies.

Two kinds of policy analyses are considered in [10]. First, they provide a technique to determine if a policy returns a specific decision on at least one access request. Second, they provide techniques to compare two policies: (a) they provide a technique to determine whether a policy is more permissive than another policy, and (b) 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.

Hughes and Bultan [17] compare the extent of similarity between two access control policies, by determining the

number of access requests to which these policies are applicable.

Two kinds of policy analyses are proposed in [14] The first one determines whether an access control policy satisfies a safety property. The second is change impact analysis on two versions of a policy. Multi Terminal Binary Decision Diagrams (MTBDDs) [5, pages 392–422] are adopted for representing policies. The MTBDDs of the policy before and after change are combined to represent the change in behaviour. From this combined diagram, one can effectively determine the access requests for which the policy returns a different authorization decision before and after the change.

In [23], a mechanism has been devised to generate test cases for XACML policies. Each test case is a pair composed of an access request and an authorization decision. Each rule in a policy is examined in turn, to determine an access request for which the corresponding authorization decision is affected by that rule. Then, a new request-decision pair is introduced into the set of test cases.

All the above analyses (with the exception of [23]) are black-box analyses. A policy is analyzed based on either the type of access requests to which it is applicable, or the authorization decision that is returned for a specific access request. The position of this work is that assessing the behaviour of individual subpolicies is important during policy debugging. This work differs from all of the above in that it is possible for the policy analyst to examine the behaviour of a subpolicy situated within a composite policy.

2.3 Approach to Implement Policy Analyses

The policy analyses of [10] and [17] are realized by translating analysis instances to SAT instances. This approach produces SAT instances of exponential size in cases involving majority voting. In [14] and [22], change impact analysis is conducted by generating Multi-Terminal Binary Decision Diagrams (MTBDDs) to represent the difference between two policies. In our work, the DIC query is translated into either SAT or PBS instances. PBS is employed to produce a compact encoding for policies involving majority-voting PCAs, and we rely on state-of-the-art PBS solvers to solve PBS instance efficiently, or to convert an PBS instance to an optimized SAT instance [13]. To the best of our knowledge, our work is the first to implement policy analysis queries via a PBS encoding.

3. μ-XACML

Our proposed policy analysis will be specified in terms of a policy composition language called μ-XACML, which is a lightweight, idealized version of XACML 2.0 [25]. The goal here is not to capture the entirety of XACML. Instead, we want to demonstrate that, even in the presence of complex policy combinators such as XACML PCAs, which feature imperative characteristics as well as majority-voting schemes, our policy analysis can still be properly defined.

Following the approach of Bruns and Huth [10], the definition of μ-XACML is parameterized by abstract access control systems. The interface between an abstract access control system and the policy composition language takes the form of request predicates. By following this design, it allows the μ-XACML and the proposed policy analysis to be independent of the choice of the underlying access control models, thereby widening the applicability of the policy analysis.

3.1 Abstract Access Control System

An access control system S is a 4-tuple $\langle PS, R, RP, \Vdash \rangle$. PS is a set of ***protection states***. R is a set of ***requests***. RP is a set of ***request predicates***. Typical members of PS, R and RP are denoted respectively by ps, r and rp. A request predicate rp describes a property of a request that is issued in a given protection state. Specifically, each pair (ps, r) either satisfies a request predicate rp, or not. The ternary relation $\Vdash \subseteq PS \times R \times RP$ specifies the semantics of a request predicate. We write $ps, r \Vdash rp$ whenever $(ps, r, rp) \in \Vdash$. Every state-request pair (ps, r) induces a variable assignment $\sigma_{ps,r} : RP \to \{0, 1\}$ such that $\sigma_{ps,r}(rp) = 1$ iff $ps, r \Vdash rp$. An access control system S is ***uniform*** iff, for every variable assignment $\sigma : RP \to \{0, 1\}$, there exists a state-request pair (ps, r) such that $\sigma_{ps,r} = \sigma$.

As one can easily see, the above notion of access control systems abstracts away the idiosyncrasy of individual access control model. So long as an access control model can be abstracted into a system of the above form, then our policy composition framework and policy analysis techniques will be applicable.

3.2 Policy Evaluation

The policy language μ-XACML offers syntax for specifying policy *expressions*. Each expression is evaluated against a state-request pair (ps, r). The result of evaluation is one of four ***authorization decisions***. The decisions "***permit***" and "***deny***" have obvious meaning. The decision "***indeterminate***" is returned when a PCA fails to harmonize the conflicting decisions returned by its component policies. The decision "***not applicable***" is returned when a policy is not applicable for the given state-request pair. Let $DS = \{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\}$ be the set of authorization decision values.

3.3 Syntax

The abstract syntax of μ-XACML is given in the following context free grammar:

$$
\begin{aligned}
pol &::= \mathsf{permit} \mid \mathsf{deny} \mid con \to pol \mid pca(pol^+) \\
pca &::= \mathsf{po} \mid \mathsf{do} \mid \mathsf{fa} \mid \mathsf{oa} \mid \mathsf{smv} \mid \mathsf{amv} \mid \mathsf{spmv} \\
con &::= rp \mid \top \mid \bot
\end{aligned}
$$

where $rp \in RP$ is a request predicate. The atomic policies permit and deny return fixed decision values. The conditional policy $con \to pol$ restricts the applicability of pol based on the satisfiability of the request predicate con. The PCA policy $pca(pol_1, \ldots, pol_k)$ combines the decisions of the subpolicies $pol_1, \ldots, pol_k$ using the PCA pca. Seven PCAs are supported in μ-XACML: (a) four standard XACML policy combining algorithms — permit-overrides (po), deny-overrides (do), first-applicable (fa), and only-applicable (oa); (b) three majority voting algorithms — simple majority voting (smv), absolute majority voting (amv), and super permit majority voting (spmv).

We write $vars(pol)$ to denote the set of request predicates appearing in policy pol.

3.4 Semantics and Derived Forms

The evaluation function $eval(\sigma, pol)$ returns the decision of policy pol when it is evaluated against the variable assignment σ. Its definition is given below.

- $eval(\sigma, \mathsf{permit}) = \mathbf{p}$.

- $eval(\sigma, \mathsf{deny}) = \mathbf{d}$.
- $eval(\sigma, con \rightarrow pol) = eval(\sigma, pol)$ if either $con = \top$, or $con = rp$ and $\sigma(rp) = 1$; otherwise, $eval(\sigma, con \rightarrow pol) = \mathbf{n}$.

The decision value returned by $eval(\sigma, pca(pol_1, \ldots, pol_k))$ depends on pca.

- Case $pca = \mathsf{po}$: If $eval(\sigma, pol_i) = \mathbf{p}$ for some $1 \leq i \leq k$, then return $\mathbf{p}$. Otherwise, if $eval(\sigma, pol_i) = \mathbf{d}$ for some $1 \leq i \leq k$, then return $\mathbf{d}$. Otherwise, if $eval(\sigma, pol_i) = \mathbf{i}$ for some $1 \leq i \leq k$, then return $\mathbf{i}$. Otherwise, return $\mathbf{n}$. Operationally, the subpolicies pol_i are evaluated from left to right, and evaluation stops as soon as one of the subpolicies evaluates to a $\mathbf{p}$.
- Case $pca = \mathsf{do}$: Similar to the case of po, except that the order of precedence is: $\mathbf{d}$, $\mathbf{i}$, $\mathbf{p}$, $\mathbf{n}$.
- Case $pca = \mathsf{fa}$: If $eval(\sigma, pol_i) = \mathbf{n}$ for all $1 \leq i \leq k$, then return $\mathbf{n}$. Otherwise, let i be the smallest index for which (a) $eval(\sigma, pol_j) = \mathbf{n}$ for $1 \leq j < i$, and (b) $eval(\sigma, pol_i) = v$, where $v \neq \mathbf{n}$. Return v. Operationally, the subpolicies pol_i are evaluated from left to right, and evaluation stops as soon as one of the subpolicies evaluates to a decision that is not $\mathbf{n}$.
- Case $pca = \mathsf{oa}$: If $eval(\sigma, pol_i) = v$ for some $1 \leq i \leq k$, where $v \in \{\mathbf{p}, \mathbf{d}\}$, but $eval(\sigma, pol_j) = \mathbf{n}$ for $j \neq i$, then return v. Otherwise, if $eval(\sigma, pol_i) = \mathbf{n}$ for all $1 \leq i \leq k$, then return $\mathbf{n}$. Otherwise, return $\mathbf{i}$. Operationally, all subpolicies pol_i are evaluated before the final decision is computed.

To articulate the decision value returned by the majority voting algorithms smv, amv and spmv, we define additional notation. We associate with each subpolicy pol_i and each decision value v a variable x_i^v, such that variable can take on a value of either 0 or 1, and $x_i^v = 1$ iff $eval(\sigma, pol_i) = v$.

- Case $pca = \mathsf{smv}$: If $\Sigma_{i=1}^k x_i^{\mathbf{p}} > \Sigma_{i=1}^k x_i^{\mathbf{d}}$, then return $\mathbf{p}$. If $\Sigma_{i=1}^k x_i^{\mathbf{d}} > \Sigma_{i=1}^k x_i^{\mathbf{p}}$, then return $\mathbf{d}$. If $\Sigma_{i=1}^k x_i^{\mathbf{n}} = k$, then return $\mathbf{n}$. Otherwise, return $\mathbf{i}$.
- Case $pca = \mathsf{amv}$: If $\Sigma_{i=1}^k x_i^{\mathbf{p}} \geq \lfloor k/2 \rfloor + 1$, then return $\mathbf{p}$. If $\Sigma_{i=1}^k x_i^{\mathbf{d}} \geq \lfloor k/2 \rfloor + 1$, then return $\mathbf{d}$. If $\Sigma_{i=1}^k x_i^{\mathbf{n}} = k$, then return $\mathbf{n}$. Otherwise, return $\mathbf{i}$.
- Case $pca = \mathsf{spmv}$: Similar to the case of amv, except that the threshold is $\lfloor 2k/3 \rfloor + 1$ rather than $\lfloor k/2 \rfloor + 1$.

Operationally, majority voting algorithms evaluate all their subpolicies pol_i before a final decision is computed.

We write $eval(ps, r, pol)$ to denote $eval(\sigma_{ps,r}, pol)$, that is, the evaluation of pol against the state-request pair (ps, r). We also define the following derived forms.

$$\mathsf{in} \stackrel{def}{=} \mathsf{oa}(\mathsf{permit}, \mathsf{deny}) \qquad \mathsf{na} \stackrel{def}{=} \bot \rightarrow \mathsf{permit}$$

The above derived forms essentially evaluate to the decision values $\mathbf{i}$ and $\mathbf{n}$ respectively.

3.5 Comparing with XACML

As disclaimed before, μ-XACML is never intended to capture the entirety of XACML. Instead, μ-XACML is designed to capture only the essential complexities of XACML so as to demonstrate that our policy analysis can be applied to realistic policy languages. We outline below the major ways in which μ-XACML deviates from XACML.

1. XACML feature rules, policies and policy sets. We flatten the hierarchy in μ-XACML, and consider only one syntactic category — policies. Therefore, we do not differentiate between rule combining algorithms and policy combining algorithms.
2. In XACML, applicability testing and decision combination are merged in a single construct (e.g., policy). In μ-XACML, we separate them into two constructs — conditional and PCA policies. This produces a more composeable and uniform language design.
3. In XACML, an indeterminate decision may be returned for two reasons: (a) a PCA fails to harmonize contradicting decisions of the component policies, or (b) an error occurs during the evaluation of a policy. In μ-XACML, we ignore possibility (b).
4. XACML supports dynamic retrieval of policies at the time of their evaluation, as well as the specification of obligations; μ-XACML supports neither.

4. DECISION IN CONTEXT

We define in this section a white-box policy analysis, Decision in Context (DIC), and showcase some useful applications of the analysis.

4.1 Preliminaries

We assume that policies are represented as abstract syntax trees (ASTs) [3] according to the grammar of §3.3. Subpolicies correspond to subtrees.

We further assume that every tree node is uniquely labelled within the AST. Thus, we identify a subpolicy by the label of the root of the corresponding subtree. We write $\mathsf{root}(pol)$ for the label of the root node of pol. Let l_1 and l_2 be the labels of two AST nodes. We write $l_1 \sqsubseteq l_2$ if l_1 is either l_2 or one of its descendants, and say that l_1 is a subpolicy of l_2. We write $l_1 \sqsubset l_2$ if $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$, and we say that l_1 is a ***proper*** subpolicy of l_2.

4.2 The DIC Query

An ***atomic DIC query*** has the form:

$$DIC(pol, l, ds)$$

where pol is a policy to be analyzed, l is the label of a subpolicy of pol, and ds is a *non-empty* subset of DS.

The above query is said to be ***satisfied*** by a variable assignment σ iff, during the evaluation of $eval(\sigma, pol)$, (a) the subpolicy referenced by l is recursively evaluated, and (b) the evaluation of this subpolicy returns one of the authorization decisions in ds. The query is said to be ***satisfied*** by a state-request pair (ps, r) iff it is satisfied by the variable assignment $\sigma_{ps,r}$. The query is ***satisfiable*** iff it is satisfied by some state-request pair. The query is ***valid*** iff it is satisfied by every state-request pair.

Note that, in order for a DIC query to be satisfied by a variable assignment, both conditions (a) and (b) must hold. That is, if a variable assignment fails to satisfy the DIC query, then either the subpolicy is not evaluated, or else its evaluation yields a decision value that is not in ds.

Condition (a) is significant because of the operational semantics of μ-XACML. Specifically, the conditional as well as the PCAs fa, po and do may not evaluate all its subpolicies due to its opportunistic semantics.

Note also that, in order for the DIC query to be well-formed, l must reference a subpolicy of pol. That is, if pol is itself a subpolicy of some bigger policy pol', and l references some subpolicy of pol' that is not a subpolicy of pol, then the DIC query is not well defined.

A ***composite DIC query*** is a boolean combination of atomic DIC queries:

$$q ::= DIC(pol, l, ds) \mid \neg q \mid q \vee q$$

The composite query $q_1 \vee q_2$ is satisfied by variable assignment σ iff σ satisfies either q_1 or q_2. The query $\neg q$ is satisfied by σ iff q is not satisfied by σ. The satisfaction of composite DIC queries by state-request pairs and the satisfiability and validity of composite DIC queries can be defined in a straightforward manner. Also, we define the derived form $q_1 \wedge q_2 \stackrel{def}{=} \neg(\neg q_1 \vee \neg q_2)$.

4.3 Applications

As a white-box policy analysis is like a debugger: it checks whether a subpolicy will be evaluated, and if so what decision values it will return. As such, the DIC query has a wide range of applications. The following are four examples.

4.3.1 Change Impact Analysis

Complex policies are prone to frequent changes. When the administrator adds new policy components, or remove or modify existing policy components, there is bound to be unanticipated changes to the behaviour of other policy components. DIC queries can be formulated to facilitate the effective identification of such unanticipated changes.

Suppose a policy pol_1 has been revised to pol_2. More specifically, the subpolicy l_1 in pol_1 has been revised into subpolicy l_2 in pol_2. We further suppose that the labels of the two ASTs are disjoint.

An policy analyst wishing to perform change impact analysis proceeds in two steps.

Step 1: Expansion or contraction of applicability.

Given a state-request pair (ps, r), a subpolicy l of policy pol is said to be ***applicable*** if the evaluation of $eval(ps, r, pol)$ results in the recursive evaluation of subpolicy l. The analyst begins by determining if the policy revision results in a change of applicability of l_1. If there has been an increase (resp. decrease) in the number of state-request pairs in which l_1 is applicable, then it is called an ***expansion (resp. contraction) of applicability***. The analyst usually has a prior expectation of whether the revision should result in expansion or contraction of applicability for the target subpolicy. A DIC query can be formulated to confirm this expectation.

Consider the following atomic DIC queries:

$$q_1^{app} = DIC(pol_1, l_1, \{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\})$$
$$q_2^{app} = DIC(pol_2, l_2, \{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\})$$

There is no contraction of applicability iff $\neg q_1^{app} \vee q_2^{app}$ is valid. Actual expansion of applicability can then be confirmed by further checking that $q_2^{app} \wedge \neg q_1^{app}$ is satisfiable.

Similarly, there is no expansion of applicability iff $\neg q_2^{app} \vee q_1^{app}$ is valid. Actual contraction of applicability can then be confirmed by further verifying that $q_1^{app} \wedge \neg q_2^{app}$ is satisfiable.

Step 2: Change of authorization decision.

Once the policy analyst has determined whether the applicability of the target subpolicy has changed, he may wish to determine whether the decision returned by that subpolicy has changed between the old and new versions. The policy analyst usually has prior expectation on whether there is a change in authorization, or what the change should be like.

Suppose $q = DIC(pol, l, ds)$, where $\emptyset \subset ds \subseteq DS$. The ***complement query*** q^c is defined to be $DIC(pol, l, DS \setminus ds)$. Note that q^c is not equivalent to $\neg q$. The query q^c checks if there can ever be a case when subpolicy l is evaluated, but the evaluation returns a decision outside of ds.

Consider the following atomic DIC queries:

$$q_1^v = DIC(pol_1, l_1, \{v\}) \qquad q_2^v = DIC(pol_2, l_2, \{v\})$$

The DIC query $q_1^v \wedge (q_2^v)^c$ is satisfiable iff there is a case when both the old and new version of the target subpolicy are applicable, but the old version returns the decision v while the new version returns a different decision.

4.3.2 Break-Glass Reduction Analysis

In [4], Ardagna *et al.* introduced the concept of ***policy spaces*** for reducing the number of emergency accesses (aka break-glass accesses) made to healthcare information systems to access patient health records.

Policy spaces are partitions (i.e., disjoint subsets) of policies in an access control system. Four such policy spaces are described in [4]:

- The ***positive authorization policy space***, denoted by ρ^+, is the set of policies that grant access under normal circumstances.
- The ***negative authorization policy space***, denoted by ρ^-, consists of those policies denying access under normal circumstances.
- The ***planned exception policy space***, denoted by ε^P, contains policies that grant access under known exceptional circumstances, which under normal circumstances will not be granted.
- The ***unplanned exception policy space***, denoted by ε^U, contains a single policy that grants access to all access requests. This covers circumstances 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.

Such a scheme can be modelled in our policy composition framework using the following policy:

$$\mathsf{fa}(\rho^+, \rho^-, \varepsilon^P, \varepsilon^U) \tag{1}$$

The atomic policy deny (resp. permit) is not allowed to appear in ρ^+ and ε^P (resp. ρ^-). The policy space ε^U is simply the atomic policy permit.

Ardagna *et al.* propose to reduce the occurrence of exceptions (requests to which access is granted by ε^P and ε^U) 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 frequently applicable conditions under which requests are granted from ε^U (by reviewing access logs).

A policy analyst may want to verify that the two kinds of periodic policy updates advocated by Ardagna *et al.* actually achieve the intended effects. This can be summarized by the following.

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

A checkmark (✓) indicates that we would like to see if there exists state-request pair in which the old version of either ε^P or ε^U is applicable, and the new version of either ρ^+ or ρ^- is applicable. This check gauges if the update produces actual benefits.

A cross (×) indicates that we would like to ensure that there is no state-request pair in which the old version of either ρ^+ or ρ^- is applicable, but the new version of either ε^P or ε^U is applicable. This check ensures that the update produces no error.

Let pol_1 and pol_2 be the old and new version of the policy in (1). Let l_1^+, l_1^-, l_1^P, l_1^U, l_2^+, l_2^-, l_2^P and l_2^U be the labels of the four policy spaces in the two policy versions. Define the following DIC query:

$$q_i^j = DIC(pol_i, l_i^j, \{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\})$$

where $i \in \{1, 2\}$ and $j \in \{+, -, P, U\}$.

To gauge if there is any real benefit for the update, the policy analyst can test if the following is satisfiable: $q_1^i \wedge q_2^j$, where $i \in \{P, U\}$ and $j \in \{+, -\}$. To verify if there is no error produced by the update, the policy analyst can test if the following is valid: $\neg(q_1^i \wedge q_2^j)$, where $i \in \{+, -\}$ and $j \in \{P, U\}$.

4.3.3 Dead Policy Identification

Another application of the DIC query is in the identification of dead policies. By dead policies (analogous to dead code [3]) we refer to those subpolicies of a composite policy that are never evaluated. For example, in the policy $\mathsf{po}(pol_1, pol_2, pol_3)$, if, for every state-request pair, at least one of pol_1 and pol_2 evaluates to $\mathbf{p}$, then pol_3 is a dead policy. A dead policy may be accidentally produced as a by-product of policy revision, or it may reflect a design flaw. Once a dead policy has been identified, the composite policy can usually be simplified.

Formally, a subpolicy referenced by l is a dead policy iff the following DIC query is *not* satisfiable (i.e., its negation is valid):

$$DIC(pol, l, \{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\})$$

In a sizeable composite policy, the search for dead policies can be performed in a top-down manner. More specifically, the AST of the composite policy can be traversed in a pre-order fashion, and each node is checked to see if it is a dead policy.

4.3.4 Complex Policy Pruning

Another useful application of the DIC query is in determining the parts of the composite policy that can be pruned (to be replaced by simpler policies). Specifically, a subpolicy that always returns the same decision can be replaced by a constant policy (i.e., permit, deny, in, na). This is comparable to constant folding in compilers [3] as well as the policy tree pruning method in [11]. Again, the presence of such a subpolicy may be the result of policy revision or design flaw.

Suppose we are to check if the subpolicy l of policy pol always return the same decision v. This can be achieved by checking if the *complement* (§4.3.1) of $DIC(pol, l, \{v\})$ is *not* satisfiable (i.e., the negation of the complement is valid). When this is the case, we know that either l is not evaluated, or else its evaluation never yields any decision other than v.

Again, the search for subpolicies to prune can be performed in a top-down manner: i.e., the AST of the composite policy is traversed in a pre-order fashion.

4.4 Complexity

The satisfiability of DIC queries naturally induces a decision problem.

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

THEOREM 1. *The DIC-SAT problem is NP complete.*

That DIC-SAT is in NP is elementary. The NP-hardness of DIC-SAT is demonstrated in Appendix A through a reduction from monotone one-in-three 3SAT [28].

5. REDUCTION TO SAT

To test the satisfiability of a DIC query, a standard implementation is to compile the DIC query into an equivalent propositional satisfiability (SAT) problem instance, and then employ a standard SAT solver to solve the SAT instance. We derived such a compilation algorithm, DIC2SAT, the design of which is the topic of this section. Our goal here is to outline the baseline implementation technique, and then point out its weakness in handling majority-voting PCAs, so as to prepare for the presentation of the alternative reduction in the next section.

5.1 The DIC2SAT Algorithm

We assume that the underlying access control system is uniform (that is, every variable assignment is realized by some state-request pair). §5.2 explains how this assumption can be relaxed.

The input and output of DIC2SAT is given below.

Input A policy pol, a label l, and a set ds of decision values, such that $DIC(pol, l, ds)$ is a well-formed DIC query.

Output A propositional formula ϕ such that $DIC(pol, l, ds)$ is satisfiable iff ϕ is satisfiable.

Specifically, the request predicates that appear in pol are taken as propositional variables in ϕ. Since the access control system is assumed to be uniform, the existence of a variable assignment that satisfies the DIC query implies that the DIC query is satisfiable.

5.2 Alternative Usages

The DIC2SAT algorithm can be used for testing the satisfiability of a composite DIC query. Suppose $q_1, \ldots, q_k$ are atomic DIC queries, and $f(q_1, \ldots, q_k)$ is a boolean combination of the atomic queries. Suppose further that ϕ_i is the SAT encoding of q_i (as returned by DIC2SAT). Then the

satisfiability of $f(q_1, \ldots, q_k)$ (as a composite query) can be determined by testing the satisfiability of $f(\phi_1, \ldots, \phi_k)$ (as a propositional formula).

While the DIC2SAT algorithm can be applied directly to uniform systems, it can be easily adapted to handle non-uniform systems. Given a DIC query q, there may be dependencies among the truth values of the request predicates appearing in q, because not every variable assignment is realized by some state-request pair. In such cases, it is usually possible to capture these dependencies in a propositional formula ψ (because there can only be finitely many request predicates appearing in q). If DIC2SAT returns a formula ϕ, then we test the satisfiability of $\phi \wedge \psi$ to find out if q is satisfiable.

As an example, suppose resource predicates rp_1 and rp_2 respectively indicate if the resource being accessed is a departmental or university resource. Suppose further that every departmental resource is considered a university resource. Then we can capture this dependency by the propositional formula $\neg rp_1 \vee rp_2$. If DIC2SAT returns ϕ, then we test the satisfiability of $\phi \wedge (\neg rp_1 \vee rp_2)$.

Generalizing the above technique, one can actually perform what-if analysis using DIC2SAT. By what-if analysis, we mean to ask if a DIC query is satisfiable by specific kinds of state-request pairs for which certain conditions (expressed in terms of a boolean combination of some request predicates) hold. For example, "is this DIC query satisfiable if the requested resource is a university equipment but the user is not an academic staff?" Suppose the condition can be captured by a propositional formula ψ, and DIC2SAT converts the DIC query to ϕ, then we test the satisfiability of $\psi \wedge \phi$.

5.3 The Overall Structure of DIC2SAT

DIC2SAT is syntax directed. That is, given a policy *pol*, DIC2SAT performs case analysis to see which variant of policy *pol* is, based on the grammar of §3.3. If *pol* is atomic, DIC2SAT returns a simple propositional formula as the encoding of *pol*. If *pol* is composite, DIC2SAT recursively invokes itself to process the subpolicies, and then combines the returned formulas to build an encoding for *pol*.

The full pseudocode of the DIC2SAT algorithm is given in the companion technical report [6]. Here we illustrate how the algorithm behaves by listing the pseudocode for the case of conditional policy.

case $pol = rp \rightarrow pol'$:
  if $l = \mathsf{root}(pol)$ **then**
    if $\mathbf{n} \in ds$ **then**
      return $\neg rp \vee \text{DIC2SAT}(pol', \mathsf{root}(pol'), ds)$;
    else
      return $rp \wedge \text{DIC2SAT}(pol', \mathsf{root}(pol'), ds)$;
  else /* $l \sqsubset \mathsf{root}(pol)$ */
    return $rp \wedge \text{DIC2SAT}(pol', l, ds)$;

We explain the intuition behind the pseudocode. The outer **if-then-else** indicates that there are two main cases: (1) the subpolicy l is *pol* itself, or (2) l is a proper subpolicy of *pol*. In case (1), subpolicy l is obviously evaluated when *pol* is evaluated. So the question is whether *pol* returns a member of *ds*. Now there are two subcases: (a) the decision **n** is one of the decisions in *ds*, or (b) otherwise. In subcase (a), there are two ways in which a member of *ds* is returned: either *rp* does not hold, and thus the conditional policy returns **n**, or else the evaluation of the subpolicy pol' yields a member of *ds*. In subcase (b), $\mathbf{n} \notin ds$, and thus the only way for a member of *ds* to be returned is when the condition *rp* holds, and also when the subpolicy pol' returns such a member.

In main case (2), l is a proper subpolicy of *pol*. We need *rp* to hold in order for pol' to be evaluated, and recursively we need subpolicy l to be evaluated to a member of *ds* when pol' is evaluated.

5.4 Compiling Standard PCAs

The compilation of the four standard PCAs is more involved. We illustrate this by considering the compilation of the first-applicable (fa) PCA.

We begin by defining some shorthands. Suppose $\langle pol_1, \ldots, pol_k \rangle$ is a sequence of policies, l is a label for which $l \sqsubseteq \mathsf{root}(pol_k)$, and *ds* is a non-empty set of decisions.

$$AllN(\langle pol_1, \ldots, pol_k \rangle) \stackrel{def}{=} \bigwedge_{i=1}^{k} \text{DIC2SAT}(pol_i, \mathsf{root}(pol_i), \{\mathbf{n}\})$$

Intuitively, the formula $AllN(\langle pol_1, \ldots, pol_k \rangle)$ is satisfiable iff every pol_i evaluates to **n**.

$$FA(\langle pol_1, \ldots, pol_k \rangle, l, ds) \stackrel{def}{=} AllN(\langle pol_1, \ldots, pol_{k-1} \rangle) \wedge \text{DIC2SAT}(pol_k, l, ds)$$

Intuitively, the formula $FA(\langle pol_1, \ldots, pol_k \rangle, l, ds)$ is satisfiable iff (a) each of $pol_1, \ldots, pol_{k-1}$ evaluates to **n**, and (b) subpolicy l of pol_k is evaluated to a decision in *ds*.

The compilation of fa is shown below:

case $pol = \mathsf{fa}(pol_1, \ldots, pol_k)$:
  if $l = \mathsf{root}(pol)$ **then**
    $\phi := \bot$;
    if $ds \cap \{\mathbf{p}, \mathbf{d}, \mathbf{i}\} \neq \emptyset$ **then**
      $\phi := \phi \vee \bigvee_{j=1}^{k} FA(\langle pol_1, \ldots, pol_j \rangle, \mathsf{root}(pol_j), ds \cap \{\mathbf{p}, \mathbf{d}, \mathbf{i}\})$;
    if $\mathbf{n} \in ds$ **then**
      $\phi := \phi \vee AllN(pol_1, \ldots, pol_k)$;
    return ϕ;
  else /* $l \sqsubset \mathsf{root}(pol)$ */
    let j be the index for which $l \sqsubseteq \mathsf{root}(pol_j)$;
    return $FA(\langle pol_1, \ldots, pol_j \rangle, l, ds)$;

The case handled by the **else** is easier to explain, so we start with that. In this case, l refers to a proper subpolicy of *pol*. So, l must be pointing to a subpolicy of one of $pol_1, \ldots, pol_k$. Let's say it is pointing to a subpolicy of pol_j. Then a simple invocation of $FA(\langle pol_1, \ldots, pol_j \rangle, l, ds)$ gives us exactly what we need.

In case l is referring to *pol*, the evaluation of l is guaranteed. We just need to ensure that the evaluation of *pol* returns a member of *ds*. The pseudocode initializes a formula ϕ to $\bot$ (the boolean constant false). The intention is that this formula will be a disjunction of further subformulas. As the construction progresses, more disjuncts will be put into the disjunction, and ϕ will be returned when the construction is complete. The first nested **if** builds disjuncts that account for *pol* returning a non-**n** member of *ds*. The second nested **if** builds disjuncts that account for *pol* returning **n** (in case the latter is a member of *ds*).

5.5 Compiling Majority Voting Algorithms

In [10], Bruns and Huth give a brute force SAT encoding of policies involving majority voting. The compilation techniques presented in the previous subsections can be applied to produce a brute-force compilation for the three majority-voting PCAs smv, amv and spmv. Rather than detailing the compilation here, we discuss below the problem of such an encoding.

Majority-voting PCAs consider the *number* of subpolicies returning a decision when a final decision is computed. Consider as an example the PCA policy $\mathsf{amv}(pol_1, \ldots, pol_n)$. A final decision of $\mathbf{p}$ is returned if at least $k = \lfloor \frac{n}{2} \rfloor + 1$ subpolicies return $\mathbf{p}$. A brute-force encoding will have to formulate a disjunction of k sub-formulas, with each sub-formula representing the condition under which one of the $\binom{n}{k}$ combinations of subpolicies return $\mathbf{p}$. So the size of the output formula produced by such an encoding will be at least $\binom{n}{k} = \Omega(2^{\frac{n}{2}})$.

The number of combinations that need to be generated by smv, amv and spmv are respectively $\Omega(2^n)$, $\Omega(2^{\frac{n}{2}})$ and $\Omega((\frac{3}{2})^{\frac{2\times n}{3}})$. This exponential growth in the size of the output formula underlines the need for a more compact encoding of DIC satisfiability, a topic to which we now turn.

6. REDUCTION TO PBS

A more compact encoding of DIC satisfiability can by obtained by PBS. This section begins with an introduction of PBS, and then proceeds to present the compilation techniques behind our DIC2PBS compiler.

6.1 Pseudo Boolean Satisfiability

A ***pseudo boolean variable*** is an integer variable that can be assigned a value of 0 or 1. A (linear) ***pseudo boolean constraint*** has one of the following general forms.

$$a_1y_1 + a_2y_2 + \ldots + a_ny_n \leq b$$
$$a_1y_1 + a_2y_2 + \ldots + a_ny_n = b$$
$$a_1y_1 + a_2y_2 + \ldots + a_ny_n \geq b$$

where a_i's and b are integer constants, and y_i's are pseudo boolean variables. A ***pseudo boolean constraint set*** is a finite set of pseudo boolean constraints. An assignment of 0 or 1 to each variable appearing in the constraint set is called a ***variable assignment***. If the variable assignment satisfies all the constraints in the constraint set, then it is a ***solution*** to the constraint set. A constraint set is ***satisfiable*** iff it has a solution. Pseudo boolean satisfiability (PBS), that is, deciding if a given pseudo boolean constraint set is satisfiable, is known to be NP complete.

6.2 The DIC2PBS Algorithm

We again assume that the underlying access control system is uniform. (Non-uniform systems can be handled in exactly the same way as discussed in §5.2.) We devised a compilation algorithm, DIC2PBS, that reduces a DIC satisfiability instance to a PBS instance. The request predicates in the DIC query appear as pseudo boolean variables in the PBS instance.

We define further notions before presenting DIC2PBS. Suppose CS is a *satisfiable* pseudo boolean constraint set, and x is a pseudo boolean variable that appears in CS. Then the pair (CS, x) is called an ***instrumented constraint set***. That CS is satisfiable is already given; the question is whether there exists a solution to CS that assigns 1 to x. An instrumented constraint set is used for representing a boolean condition: the condition holds iff there exists a solution to CS such that x is set to 1.

The input and output of DIC2PBS is given below.

Input A policy pol, a label l, and a set ds of decision values, such that $DIC(pol, l, ds)$ is a well-formed DIC query.

Output An instrumented constraint set (CS, x) such that:

- The request predicates appearing in pol will appear in CS as pseudo boolean variables. Let RV be the set of variables in CS that correspond to request predicates. Let AV be the rest of variables in CS.
- $x \in AV$.
- Every variable assignment for the variables in RV can be extended to a solution of CS.
- 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 solution of CS will assign 1 to x. If σ does not satisfy the input DIC query, then every extension of σ to a solution of CS will assign 0 to x. Consequently, the input DIC query is satisfiable iff there is a solution for CS for which x is set to 1.

Suppose the instrumented constraint set (CS, x) is returned by DIC2PBS. Consider the pseudo boolean constraint set $CS' = CS \cup \{x = 1\}$. The constraint set CS' is satisfiable iff the input DIC query is satisfiable.

6.3 Compiling Majority-Voting PCAs

To demonstrate the compactness of the PBS encoding, we here outline the compilation for the query $DIC(pol, l, ds)$, where:

- $pol = \mathsf{smv}(pol_1, \ldots, pol_n)$,
- $l = \mathsf{root}(pol)$, and
- $ds = \{\mathbf{p}\}$.

Recall that the simple majority voting PCA (smv) returns $\mathbf{p}$ iff the number of subpolicies returning $\mathbf{p}$ is strictly larger than the number of subpolicies returning $\mathbf{d}$. The compilation proceeds in three steps.

1. For $1 \leq i \leq n$, let $(CS_i^{\mathbf{p}}, x_i^{\mathbf{p}})$ be the return value of $DIC2PBS(pol_i, \mathsf{root}(pol_i), \{\mathbf{p}\})$. (That is, $(CS_i^{\mathbf{p}}, x_i^{\mathbf{p}})$ captures the condition that pol_i evaluates to $\mathbf{p}$.)
2. For $1 \leq i \leq n$, let $(CS_i^{\mathbf{d}}, x_i^{\mathbf{d}})$ be the return value of $DIC2PBS(pol_i, \mathsf{root}(pol_i), \{\mathbf{d}\})$. (That is, similar to Step 1, the variable $x_i^{\mathbf{d}}$ will assume the value 1 iff pol_i evaluates to $\mathbf{d}$.)
3. Return (CS, y), where y is a fresh variable, and CS is the union of the following three sets.
   (a) $CS_1^{\mathbf{p}} \cup \ldots \cup CS_n^{\mathbf{p}}$,
   (b) $CS_1^{\mathbf{d}} \cup \ldots \cup CS_n^{\mathbf{d}}$, and

(c) the set containing the following two constraints:

$$\sum_{i=1}^{n} x_i^{\mathbf{p}} - \sum_{i=1}^{n} x_i^{\mathbf{d}} \geq y + (1-y) \times (-n) \quad (2)$$

$$\sum_{i=1}^{n} x_i^{\mathbf{p}} - \sum_{i=1}^{n} x_i^{\mathbf{d}} \leq y \times n \quad (3)$$

To understand Step 3, note that $\sum_{i=1}^{n} x_i^{\mathbf{p}}$ is the number of subpolicies returning $\mathbf{p}$, $\sum_{i=1}^{n} x_i^{\mathbf{d}}$ is the number of subpolicies returning $\mathbf{d}$, constraint (3) forces y to 1 when $\sum_{i=1}^{n} x_i^{\mathbf{p}} > \sum_{i=1}^{n} x_i^{\mathbf{d}}$, and constraint (2) forces y to 0 when $\sum_{i=1}^{n} x_i^{\mathbf{p}} \leq \sum_{i=1}^{n} x_i^{\mathbf{d}}$.

Note that the growth in size of the output is polynomial rather than exponential.

The compilation of other cases of majority-voting PCAs are similar. For details please consult the companion technical report [6].

6.4 Compiling the Rest of the Policy Language

While PBS provides a more compact encoding for majority-voting PCAs, the compilation strategy of DIC2SAT for conditional policies and standard PCA policies can be reused as is in DIC2PBS. The role of a propositional formula in DIC2SAT is now assumed by an instrumented constraint set in DIC2PBS. DIC2SAT constructs boolean combinations of formulas returned by recursive calls. Therefore, for DIC2PBS, we need a way to formulate instrumented constraint sets that express the conjunction, disjunction and negation of boolean conditions encoded as instrumented constraint sets.

In the following, we present two algorithms, *AtLeast* and *AtMost*, that can be used for encoding boolean combinations.

Suppose $\langle (CS_1, x_1), \ldots, (CS_n, x_n) \rangle$ is a *non-empty* sequence of instrumented constraint sets (with distinct variables $x_1, \ldots, x_n$). Suppose further than m is a *non-negative* integer. The function $AtLeast(\langle (CS_1, x_1), \ldots, (CS_n, x_n) \rangle, m)$ returns an instrumented constraint set (CS, y). The variable y is a fresh variable. The constraint set CS contains all the constraints in $CS_1 \cup CS_2 \cup \ldots CS_k$, plus the following two constraints.

$$\sum_{i=1}^{n} x_i \geq y \times m \quad (4)$$

$$\sum_{i=1}^{n} x_i \leq y \times n + (1-y) \times (m-1) \quad (5)$$

If $\sum_{i=1}^{n} x_i$ is at least m, then constraint (5) forces y to 1. If $\sum_{i=1}^{n} x_i$ is smaller than m, then constraint (4) forces y to 0.

The intention is that, if each instrumented constraint set (CS_i, x_i) represents a boolean condition, then the output (CS, y) expresses the boolean condition that at least m of the input variables (x_i) are set to 1. The *AtLeast* function can be used for encoding disjunction (with m set to 1) and conjunction (with m set to n).

We also devised a function *AtMost* for ensuring that no more than a certain number of conditions hold. This function can be used for encoding negation (with a single instrumented constraint set as input and with the threshold set to zero).

In summary, with the use of instrumented constraint sets, one can encode boolean combinations, and thus the rest of the policy language can be compiled using exactly the same compilation strategy previously used in DIC2SAT.

7. EMPIRICAL STUDY

This section reports an empirical study that was conducted to compare the relative efficiency of testing DIC satisfiability via the PBS encoding versus the SAT encoding.

7.1 Prototype Implementation

The DIC2SAT and DIC2PBS algorithms were implemented in respectively 2210 and 1949 lines of Java code, among which 1176 lines are shared between the two implementations.

The DIC2SAT implementation returns the abstract syntax tree of a propositional formula. It further performs a post-processing step to convert the generated formula into CNF form. This transformation is based on the Tseitin transformation [30]. The CNF formula is then fed to the MiniSat Solver [12, 2]. The MiniSat Solver is an open-source SAT solver implemented in C++. The conversion to CNF is necessary because MiniSat expects its input in CNF. The solver reports the satisfiability of its input, and the time in seconds taken to determine satisfiability.

The DIC2PBS implementation constructs an instrumented constraint set (CS, x) as specified in §6. It then passes the constraint set $CS \cup \{x = 1\}$ to the MiniSat+ Solver [13, 1], which is an open-source PBS solver implemented in C++. Again, the solver reports satisfiability and timing in seconds.

7.2 Experiment Setup

Overall Design.

Random instances of DIC queries were generated for various ranges of policy size. For each randomly generated DIC query, the aforementioned DIC2SAT and DIC2PBS implementations were invoked to compile the DIC query into a corresponding SAT and PBS instance. MiniSat and MiniSat+ were invoked to solve that instance. The average compile time and the average solver time for each size range is computed. Details are given in the following.

Hardware.

The experiment was conducted on a High Performance Computing Unix server, named BigBox, at the University of Calgary. The server features a 16-core processor and 128 GB RAM. A total of 10 GB RAM has been allocated to the Java Virtual Machine (JVM) for this experiment.

DIC Instances.

Random instances of the atomic DIC query were generated according to the scheme sketched below. DIC instances are generated for different policy sizes, ranging from 3 to 300. We divide the size range into intervals. The first interval is 3–20. Thereafter, each interval covers 20 policy sizes, such that the last interval is 281–300. A total of 50 random DIC instances were generated for each interval.

Generation of DIC Instances.

Each randomly generated DIC instance has the form $(pol, \mathsf{root}(pol), ds)$, where pol is a policy, and ds is a non-empty subset of $\{\mathbf{p}, \mathbf{d}, \mathbf{i}, \mathbf{n}\}$. Each of the 15 possible non-empty subsets is selected with equal probability.

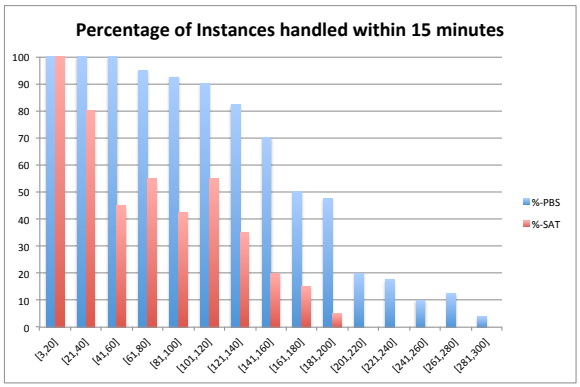

Figure 1: Comparison of Percentage of Handled Instances by DIC2SAT and DIC2PBS.

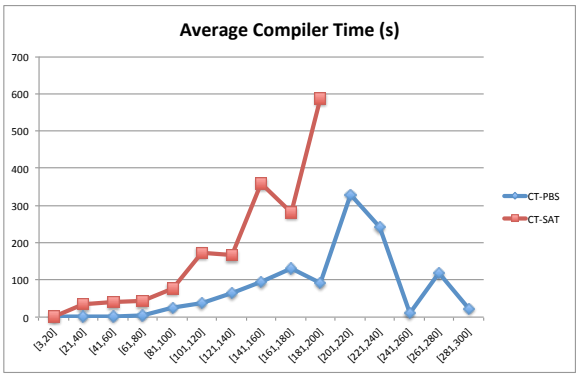

Figure 2: Comparison of Average Compile Times of DIC2SAT and DIC2PBS.

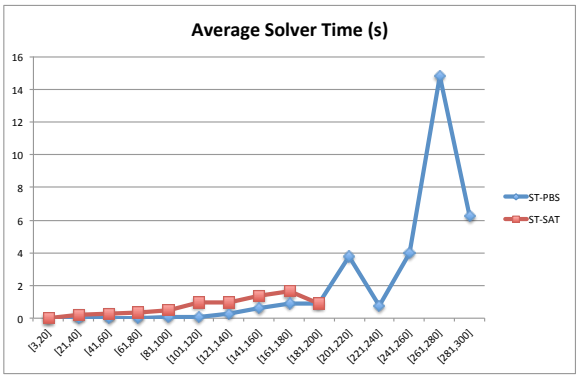

Figure 3: Comparison of Average Solver Times of SAT and PBS Instances.

Given a size parameter n, a random policy of size n is generated using the algorithm *PolGen* in Appendix B, with minor variations to be discussed below. The policies generated by *PolGen* contains both standard as well as majority-voting PCAs. We adapt *PolGen* so that the top-level AST node of the generated policy must of the form $pca(pol_1, \ldots, pol_k)$ (i.e., it can be neither an atomic nor conditional policy). In addition, *pca* must be one of the three majority-voting algorithms (each chosen with probability 1/3). Note that these restrictions apply only to the top-level AST node. The other subpolicies are generated with the probability distribution specified in Appendix B, and thus they contain either standard or majority-voting PCAs.

We introduced the above bias to the top-level AST node because we want to compare the relative performance of the two encodings in the case of majority-voting PCAs.

Time Out.

During pilot runs of the experiment (with a less capable hardware environment), we noticed that DIC2SAT would completely exhaust its heap space when dealing with DIC queries with large SAT encodings, resulting either in an out-of-memory error (usually accompanied by numerous invocations of the garbage collector), or in thrashing of memory pages. In either cases, the compile time is either unavailable or prohibitively large. To prevent this from happening in the real experiment, we set a time-out bound of 15 minutes for the compile time. If a DIC query can be compiled within the time-out bound, then that query is said to be "***handled***" by the compiler. We recorded the percentage of randomly generated DIC queries that were successfully handled by each of DIC2SAT and DIC2PBS.

Measurements.

The following four measurements were made for each randomly generated DIC instance that could be properly handled by a compiler within the time-out bound.

- **CT-SAT** (resp. **CT-PBS**) is the time in seconds to compile a DIC query to a SAT (resp. PBS) instance, using the DIC2SAT (resp. DIC2PBS) implementation.

- **ST-SAT** (resp. **ST-PBS**) is the time in seconds taken by the MiniSat SAT solver (resp. MiniSat+ PBS solver) to determine the satisfiability of the compiled SAT (resp. PBS) instance.

CT-SAT and **CT-PBS** were measured using JETM, the Java Execution Time Monitor. **ST-SAT** and **ST-PBS** were reported by MiniSat and MiniSat+.

7.3 Experiment Results

Figure 1 depicts, for each policy size interval, the percentage of DIC instances that were successfully handled by each of DIC2SAT and DIC2PBS without triggering time out. The vertical axis corresponds to percentages of handled DIC instances, and the horizontal axis corresponds to policy size intervals. In all input intervals, DIC2PBS handled more DIC instances than DIC2SAT. For instance, DIC2SAT began not to be able to handle all instances when policy size reaches 21. After than point, percentage of handled instances dropped to around 50% or below in the next few intervals. DIC2SAT failed to handle any instances of size 201 or more. In contrast, DIC2PBS demonstrated a more graceful degradation. The percentage of handled instances remains roughly 50% even for input sizes of 181–200, and remains non-zero even up to input sizes of 281–300. This is explained by noticing that the SAT instances produced by DIC2SAT are much bigger than those generated by DIC2PBS.

Figure 2 depicts the average compile time (i.e., **CT-SAT** and **CT-PBS**), in seconds, for each policy size interval. Note that as DIC2SAT failed to handle any DIC instances of policy size 200 or above, the line for **CT-SAT** broke off after

that point. Except for very small DIC instances, **CT-SAT** consistently exceeded **CT-PBS**. This, again, is explained by difference in encoding size produced by the two compilers. Note that the line for CT-PBS actually decreased for large DIC instances. The reason is that, for bigger input size, many of the difficult instances are not handled by DIC2PBS, and it is likely that the remaining instances are not particularly challenging.

Figure 3 depicts the average solver time (i.e., **ST-SAT** and **CT-PBS**), in seconds, for each policy size interval. Note that, if the DIC instance can be handled by the compiler, the solver time is actually insignificant.

8. CONCLUSION AND FUTURE WORK

The DIC analysis has been proposed as a white-box policy analysis. We demonstrated both the wide range of applications of the analysis, as well as the feasibility of the analysis via SAT and PBS encodings. Of particular interest is our novel application of PBS to achieve a compact encoding for policies containing majority-voting PCAs.

We are currently extending our work to adapt DIC analysis to Relationship-Based Access Control [15, 16, 9]. Research challenges include the choice of request predicates as well as formalizing their dependencies.

Acknowledgments

This work is supported in part by an NSERC Discovery Grant and a Canada Research Chair.

9. REFERENCES

[1] MiniSat+ PBS Solver. `http://minisat.se/MiniSat+.html`.

[2] MiniSat SAT Solver. `http://minisat.se/MiniSat.html`.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2nd edition, 2006.

[4] 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 *LNCS*, pages 254–267, London, UK, 2008. Springer.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[6] Jayalakshmi Balasubramaniam and Philip W. L. Fong. A white-box policy analysis and its efficient implementation. Technical Report 2013-1042-09, Department of Computer Science, University of Calgary, Alberta, Canada, 2013.

[7] Moritz Y. Becker. Specification and analysis of dynamic authorization policies. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF'09)*, pages 203–217, Port Jefferson, New York, USA, 2009.

[8] 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.

[9] Glenn Bruns, Philip W. L. Fong, Ida Siahaan, and Michael Huth. Relationship-based access control: Its expression and enforcement through hybrid logic. In *Proceedings of the 2nd ACM Conference on Data and Application Security (CODASPY'2012)*, San Antonio, TX, USA, February 2012.

[10] 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.

[11] 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.

[12] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 502–518, Santa Margherita Ligure, Italy, 2004. Springer.

[13] 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.

[14] 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.

[15] Philip W. L. Fong. Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy (CODASPY'11)*, pages 191–202, San Antonio, Texas, USA, February 2011.

[16] Philip W. L. Fong and Ida Siahaan. Relationship-based access control policies and their policy languages. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT'11)*, pages 51–60, Innsbruck, Austria, June 2011.

[17] 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.

[18] 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.

[19] 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*, pages 239–242, Saint John, New Brunswick, Canada, 2009. IEEE Computer Society.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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*, pages 1087–1103, Monterrey, Mexico, 2008. Springer.

[25] Tim Moses. eXtensible Access Control Markup Language (XACML) Version 2.0. Technical report, OASIS Access Control TC, 2005.

[26] 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.

[27] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms*. Prentice Hall, 1977.

[28] 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, CA, USA, May 1978.

[29] 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.

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

[31] 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.

APPENDIX

A. NP-HARDNESS OF DIC-SAT

To demonstrate the NP-hardness of DIC-SAT, we outline below a reduction from monotone one-in-three 3SAT [28].

An instance of monotone one-in-three 3SAT problem is similar to an instance of 3SAT, except that all literals are positive. That is, every clause is a disjunction of three variables. Monotone one-in-three 3SAT decides whether there exists a truth assignment that satisfies exactly one variable in each of the clauses. The problem is known to be NP-complete [28].

Given an instance of montone one-in-three 3SAT, $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$, where each C_i is of the form $x_1^i \vee x_2^i \vee x_3^i$, we construct a corresponding DIC-SAT instance $q_\phi = DIC(pol, \mathsf{root}(pol), \{\mathbf{p}\})$, where pol is defined as follows

$$\mathsf{amv}(pol_1, \ldots, pol_k, \overbrace{\mathsf{deny}, \ldots, \mathsf{deny}}^{k-1 \text{ times}})$$

and each pol_i is defined as follows

$$\mathsf{oa}(x_1^i \to \mathsf{permit}, x_2^i \to \mathsf{permit}, x_3^i \to \mathsf{permit})$$

It is easy to see that q_ϕ is satisfiable iff there is a truth assignment that satisfies exactly one variable in each clause of ϕ.

B. GENERATION OF RANDOM POLICIES

The following specifies an algorithm *PolGen* for generating a random policy of size n, with RP as the set of request predicates. We write m as a shorthand for the size of RP. Note that the policy returned *PolGen* does not contain $\top$ and $\bot$.

Algorithm $PolGen(n, RP)$:

Case $n = 1$**.** Each of deny and permit is generated with probability 1/2.

Case $n = 2$**.** A conditional policy $rp \to pol$ will be generated. One of the m request predicates from RP is selected to be rp, each with probability $1/m$. Either deny or permit is selected to be pol, each with probability 1/2.

Case $n \geq 3$**.** Either a conditional policy $rp \to pol$ or a PCA policy $pca(pol_1, \ldots, pol_k)$ is generated, each with probability 1/2. The following specifies the procedure for each of the two subcases.

- **Subcase** $rp \to pol$**.** One of the m members of RP is selected to be rp, with probability $1/m$. The subpolicy pol is generated by a recursive call to $PolGen(n-1, RP)$.
- **Subcase** $pca(pol_1, \ldots, pol_k)$**.** Firstly, pca will be selected from one of the seven combining algorithms (i.e., po, do, fa, oa, smv, amv, and spmv), each with probability 1/7. Secondly, we generate k and $pol_1, \ldots, pol_k$. To understand how this is done, we digress to introduce the idea of ***compositions***.
  A composition of a positive integer n is a sequence of positive integers $n_1, \ldots, n_k$ such that $k \leq n$ and $n = n_1 + \cdots + n_k$. There is a total of 2^{n-1} compositions. The composition with $k = 1$ is said to be ***degenerate***. So there is a total of $2^{n-1} - 1$ nondegenerate compositions. See [27, §5.3.1] for an algorithm for generating random compositions.
  We randomly generate a nondegenerate composition of $n-1$, so that each nondegenerate composition is generated with equal probability. Suppose the selected composition is $n_1, \ldots, n_k$. Then we recursively invoke $PolGen(n_i, m)$ to generate pol_i.