Specification and Analysis of ABAC Policies in a Rule-Based Framework

Besik Dundua, Temur Kutsia, Mircea Marin, and Mikheil Rukhaia

Abstract Attribute-based access control (ABAC) is a logical access control paradigm whereby access rights to system resources are granted through the use of policies that are evaluated against the attributes of entities (user, subject, and object), operations, and the environment relevant to a request. Many ABAC models, with different variations, have been proposed and formalized.

Since the access control policies that can be implemented in ABAC have inherent rule-based specifications, it is natural to adopt a rule-based framework to specify and analyse their properties. We describe the design and implementation of a software tool implemented in Mathematica. Our tool makes use of the rule-based capabilities of a rule-based package developed by us, can be used to specify configurations for the foundational model ABAC_α of ABAC, and to check safety properties.

1 Introduction

Access (authorization) control is a fundamental security technique concerned with determining the allowed activities of legitimate users, and mediating every attempt by a user to access a resource in a computing envionment. Over the years, many access control models have been developed to address various aspects of computer security, including: Mandatory Access Control (MAC) [13], Discretionary Access

Besik Dundua
VIAM, Ivane Javakhishvili Tbilisi State University and FBT, International Black Sea University, Georgia, e-mail: bdundua@gmail.com

Temur Kutsia
RISC, Johannes Kepler University Linz, Austria, e-mail: kutsia@risc.jku.at

Mircea Marin
West University of Timişoara, Romania, e-mail: mircea.marin@e-uvt.ro

Mikheil Rukhaia
VIAM, Ivane Javakhishvili Tbilisi State University, Georgia, e-mail: mrukhaia@yahoo.com

Control (DAC) [14], and Role-based Access Control (RBAC) [4]. Attribute-Based Access Control (ABAC) has received significant attention recently, although the concept has existed for more than twenty years. According to NIST [5]

> ABAC is an access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject, assigned attributes of the object, environment conditions, and a set of policies that are specified in terms of those attributes and conditions.

ABAC is is considered a next generation authorization paradigm which eliminates many limitations of the previous access control paradigms. It is dynamic: access control permissions are determined when the access control request is made; it is fine-grained: attributes can be added, to form detailed rules for access control policies; it has support for contextual/environmental conditions; and last but not least: it is easy to administer, flexible, and scalable. In fact, the access control policies that can be implemented in ABAC are limited only by the computational language and the richness of the available attributes. In particular, ABAC policies can be easily configured to simulate DAC, MAC and RBAC.

Until recently, there were no widely accepted formal models for ABAC. The foundational operational models ABAC_α and ABAC_β, and the administrative model GURA were proposed recently [6] as models with "just sufficient" features that can be used to easily and naturally configure the traditional access control models and some advanced features and extensions of RBAC.

The (efficient) implementation and analysis of these formal operational models of ABAC is of great importance. For this purpose, we designed and implemented a software tool that allows to specify configurations of ABAC_α policies, and to analyse them. The tool is implemented in Mathematica [16] and is based on the capabilities of ρLog [11, 9], a rule-based system implemented by us on top of the rule-based capabilities of Mathematica. We highlight the main features that make our rule-based system adequate to specify and analyze the configurations of the access control policies of ABAC_α.

The rest of this chapter is structured as follows. Section 2 contains a brief description of ρLog and the foundational model ABAC_α. In Sect. 3 we describe the rule-based tool designed by us for the specification and analysis of ABAC_α. In Sect. 4 we draw some conclusions and directions for future work.

2 Preliminaries

2.1 The ρLog system

ρLog is a system for rule-based programming with strategies and built-in support for constraint logic programming (CLP). This is a programming style similar to Constraint Logic Programming, where programs consist of rules which are used

to answer queries using a calculus based on a variation of SLDnf resolution [2] combined with constraint solving. There are, however, some significant differences.

The specification language has an alphabet $\mathscr{A}$ consisting of the following pairwise disjoint sets of symbols:

- $\mathscr{V}_{\mathsf{T}}$: term variables, denoted by $x, y, z, \ldots$,
- $\mathscr{V}_{\mathsf{S}}$: hedge variables, denoted by $\bar{x}, \bar{y}, \bar{z}, \ldots$,
- $\mathscr{V}_{\mathsf{F}}$: function variables, denoted by $X, Y, Z, \ldots$,
- $\mathscr{V}_{\mathsf{C}}$: context variables, denoted by $\overline{X}, \overline{Y}, \overline{Z}, \ldots$,
- $\mathscr{F}$: unranked function symbols, denoted by $f, g, h, \ldots$.

and distinguishes the following syntactic categories:

$t ::= x \mid f(\tilde{s}) \mid X(\tilde{s}) \mid \overline{X}(t)$	Term
$\tilde{t} ::= t_1, \ldots, t_n \quad (n \geq 0)$	Sequence of terms
$s ::= t \mid \bar{x}$	Hedge element
$\tilde{s} ::= s_1, \ldots, s_n \quad (n \geq 0)$	Hedge
$C ::= \circ \mid f(\tilde{s}_1, C, \tilde{s}_2) \mid X(\tilde{s}_1, C, \tilde{s}_2) \mid \overline{X}(C)$	Context

Hence, hedges are sequences of hedge elements, hedge variables are not terms, term sequences do not contain hedge variables, contexts (which are not terms either) contain a single occurrence of the hole. We do not distinguish between a singleton hedge and its sole element.

We denote the set of terms by $\mathscr{T}(\mathscr{F}, \mathscr{V})$, hedges by $\mathscr{H}(\mathscr{F}, \mathscr{V})$, and contexts by $\mathscr{C}(\mathscr{F}, \mathscr{V})$. Ground (i.e., variable-free) subsets of these sets are denoted by $\mathscr{T}(\mathscr{F})$, $\mathscr{H}(\mathscr{F})$, and $\mathscr{C}(\mathscr{F})$, respectively.

We make a couple of conventions to improve readability. We put parentheses around hedges, writing, e.g., $(f(a), \bar{x}, b)$ instead of $f(a), \bar{x}, b$. The empty hedge is written as $()$. The terms $a()$ and $X()$ are abbreviated as a and X, respectively, when it is guaranteed that terms and symbols are not confused. For hedges $\tilde{s} = (s_1, \ldots, s_n)$ and $\tilde{s}' = (s'_1, \ldots, s'_m)$, the notation $(\tilde{s}, \tilde{s}')$ stands for the hedge $(s_1, \ldots, s_n, s'_1, \ldots, s'_m)$. We use $\tilde{s}$ and $\tilde{r}$ for arbitrary hedges, and $\tilde{t}$ for sequences of terms.

We will also need anonymous variables for each variable category. They are variables without name, well-known in declarative programming. We write just $\_$ for an anonymous term or function variable, and $\_\_$ for an anonymous hedge or context variable. The set of anonymous variables is denoted by $\mathscr{V}_{\mathsf{An}}$.

A syntactic expression (or, just an expression) is an element of the set $\mathscr{F} \cup \mathscr{V} \cup \mathscr{T}(\mathscr{F}, \mathscr{V}) \cup \mathscr{H}(\mathscr{F}, \mathscr{V}) \cup \mathscr{C}(\mathscr{F}, \mathscr{V})$. We denote expressions by E. Atoms are reducibility formulas $t :: t_1 \Longrightarrow t_2$ with the intended reading "t_1 reduces to t_2 with strategy t." The negation of this atom is written as $t :: t_1 \not\Longrightarrow t_2$.

The rules of ρLog are of the form

$$f(\tilde{s}) :: t' \Longrightarrow t'' \leftarrow cond_1, \ldots, cond_n. \tag{1}$$

with the intended reading "$f(\tilde{s}) :: t' \Longrightarrow t''$ holds whenever $cond_1$ and … and $cond_n$ hold", and provide declarative semantics for reducibility formulas. f is the identifier

of the strategy and $\tilde{s}$ is its argument: If $\tilde{s}$ is $()$, the strategy is *atomic*, otherwise it is *parametric*. We view (1) as a partial definition of f.

Some strategies with frequent applications are predefined:

- `id`:: $s \Longrightarrow t$ holds if $s = t$.
- `elem`:: $l \Longrightarrow e$ holds if e is an element of list l.
- `subset`:: $l \Longrightarrow s$ holds if s is subset of set l.
- `fmap`$(t) :: f(s_1,\ldots,s_n) \Longrightarrow f(t_1,\ldots,t_n)$ holds if $t :: s_i \Longrightarrow t_i$ for $1 \leq i \leq n$.

Another way to specify strategies is by using the predefined combinators:

- $t_1 \circ t_2 :: t' \Longrightarrow t''$ holds if $t_1 :: s \Longrightarrow u$ and $t_2 :: u \Longrightarrow t$ hold for some u.
- $t_1 | t_2 :: t' \Longrightarrow t''$ holds if either $t_1 :: t' \Longrightarrow t''$ or $t_2 :: t' \Longrightarrow t''$ holds.
- $t^* :: t' \Longrightarrow t''$ holds if either $t' = t''$ or there exist $u_1,\ldots,u_n$ such that $u_1 = t'$, $u_n = t''$ and $t :: u_i \Longrightarrow u_{i+1}$ for all $1 \leq i < n$.
- `first_one`$(t_1,\ldots,t_n) :: t' \Longrightarrow t''$ holds if there exists $1 \leq i \leq n$ such that $t_i :: t' \Longrightarrow t''$ and $t_j :: t' \not\Longrightarrow t''$ hold for $1 \leq j < i$.
- `nf`$(t) :: t' \Longrightarrow t''$ holds if both $t^* :: t' \Longrightarrow t''$ and $t :: t'' \not\Longrightarrow \_$ hold.

ρLog can answer queries of the form $cond_1 \wedge \ldots \wedge cond_m$ where the variables are (implicitly) existentially quantified. The constraints $cond_i$ in queries and programs are of three kinds: reducibility atoms $t :: t' \Longrightarrow t''$, irreducibility literals $t :: t' \not\Longrightarrow t''$; and (3) boolean formulas that can be properly interpreted by the constraint solving component of ρLog. To instruct our system to compute one (resp. all) substitution(s) for the variables in the query $cond_1 \wedge \ldots \wedge cond_n$ for which it holds, we can submit requests of the form

Request($cond_1 \wedge \ldots \wedge cond_n$) or RequestAll($cond_1 \wedge \ldots \wedge cond_n$)

Another use of ρLog is to compute one or all reducts of a term with respect to a strategy. The request

ApplyRule(t, t')

instructs ρLog to compute one (if any) reduct of t' with respect to strategy t, that is, a term t'' such that formula $t :: t' \Longrightarrow t''$ holds. ρLog reports `"no solution found."` if there is no reduct of t' with t. ρLog can also be instructed to find all reducts of a term with respect to a strategy, with

ApplyRuleList(t, t')

To illustrate, consider the rule-based solutions to the following problems:

1. To eliminate all duplicates of elements in a list `L`, we submit the request ApplyRule(`nf(elim2)`,L) where strategy `elim2` is defined by the rule

   `elim2`:: $\{\overline{x}, x, \overline{y}, x, \overline{z}\} \Longrightarrow \{\overline{x}, x, \overline{y}, \overline{z}\} \leftarrow .$

   For example, ApplyRule(`nf(elim2)`,$\{1,2,7,2,3,1\}$) yields answer $\{1,2,7,3\}$.
2. To find out if (or which) `e` is an element of a list L, we can submit the request Request(elem::$L \Longrightarrow x$) where strategy `elim` is defined by the rule

   `elem`:: $\{\_\_, x, \_\_\} \Longrightarrow x \leftarrow .$

For example, Request(`elem`:: $\{1,2,3\} \Longrightarrow x$) can return the answer $\{x \mapsto 1\}$, and RequestAll(`elem`:: $\{1,2,3\} \Longrightarrow x$) returns $\{\{x \mapsto 1\},\{x \mapsto 2\},\{x \mapsto 3\}\}$.

3. To find all function symbols from a list L that occur in an expression E, we can submit the request ApplyRuleList(`getF`$(L),E$), where the parametric strategy `getF` is defined by the rule

   `getF`$(y) :: \_\_(F(\_\_)) \Longrightarrow F \leftarrow (\texttt{elem} :: y \Longrightarrow F)$.

   For example, `{f,g}` is the answer to the query

   ApplyRuleList`(getF({f,g,u,v,w}),f(g(a(),h(),b())))`

Sequence and context variables permit matching to descend to arbitrary depth and width in a term represented as a tree. The downside of using these kinds of variables in full generality is infinitary unification, and thus the impossibility to find a sound and complete calculus for ρLog. To avoid this problem, we adopted a natural syntactic restriction, called *determinism* [9], that ensure that all inference steps of our underlying calculus can be performed by computing matchers instead of most general unifiers. The good news is that matching with sequence and context variables is finitary [3].

2.2 The operational model of ABAC$_\alpha$

ABAC$_\alpha$ is a formal model of ABAC proposed by X. Jin in his PhD thesis [6] with a minimal set of features to configure the well-known access control models DAC, MAC, and RBAC. The core components of this operational model are: : users (U), subjects (S), objects (O), user attributes (UA), subject attributes (SA), object attributes (OA), permissions (P), authorization policy, creation and modification policy, and policy languages.

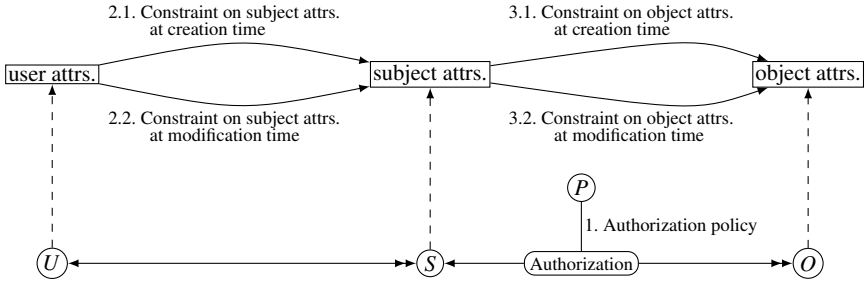

Fig. 1 The structure of ABAC$_\alpha$ model (adapted from [7])

Users represent human beings who create and modify subjects, and access resources through subjects. Subjects represent processes created by users to perform

some actions in the system. Objects represent system entities that should be protected. Users, subjects and objects are mutually disjoint in ABAC$_\alpha$, and are collectively called **entities**. Each user, subject, object is associated with a finite set of user attributes (UA), subject attributes (SA) and object attributes (OA) respectively. Every attribute *att* has a type, scope, and range of possible values. The sets of attributes specific to each kind of entity, together with their corresponding type, scope, and range, are specified in a **configuration type** of ABAC$_\alpha$: there will be one configuration type for DAC, and others for MAC, RBAC, etc.

In ABAC$_\alpha$, the type of an attribute is either atomic or set. The scope of each attribute is a finite set of values SCOPE(at). If at is of atomic type, then at can assume any value from SCOPE(at), otherwise it can assume any subset of values from SCOPE(at). Formally, this means that the range Range(at) of possible values of an attribute at is either SCOPE(at) if at is atomic or $2^{\text{SCOPE}(at)}$ if at is set, where each SCOPE(at) is either an unordered, a totally ordered, or a partially ordered finite set. There are six policies that control the operational behaviour of an ABAC$_\alpha$-based system, and each of them involves the interaction of two entities:

- authorization policies, which control the permissions that a user can hold on objects and exercise through subjects. Every configuration specifies a finite set P of permissions, and an authorization policy for every $p \in P$,
- policies to control the creation of a subject by a user, or of an object by a subject,
- policies for attribute value assignment: to a subject by the user who created it; or to an object by a subject,
- policies to control subject deletion by its creator.

All these policies grant/deny the corresponding operation based on the result of a boolean function which depends on the old and new attribute values of the interacting entities. According to [6, 1], each of these six boolean functions can be specified as a boolean formula in an instance of a language scheme called Common Policy Language (CPL). In CPL, the syntax of any formula ϕ is of the form

$$\begin{aligned}
\phi \ &::= \phi \wedge \phi \mid \phi \vee \phi \mid (\phi) \mid \neg\phi \\
&\quad \mid \exists x \in set.\phi \mid \forall x \in set.\phi \mid set\ \text{setcompare}\ set \\
&\quad \mid atomic \in set \mid atomic\ \text{atomiccompare}\ atomic \\
\text{setcompare} \ &::= \subset \mid \subseteq \mid \not\subseteq \\
\text{atomiccompare} \ &::= < \mid = \mid \leq
\end{aligned}$$

where *set* is a finite set of values, and *atomic* are concrete values.

3 A rule-based framework for ABAC$_\alpha$

In this section, we describe the rule-based tool designed by us for the specification and analysis of ABAC$_\alpha$.

Every entity (user, subject, or object) is completely described by its attribute values. Therefore, we chose to represent every entity as a term $K(at_1(v_1), \ldots, at_m(v_m))$

where $K \in \{\texttt{U},\texttt{S},\texttt{O}\}$ indicates the kind of entity, and every subterm $at_i(v_i)$ indicates that attribute at_i has value v_i. Every user has a unique identifier given by the value of its attribute `id`. Subjects are created by users and retain the identifier of their creator in the value of subject attribute `id`. From now on, we will assume the existence of a function UId(e) which returns the value of attribute `id` for every entity $e \in U \cup S$. Apart from this, the attribute names, their types and scope are characteristic to a particular configuration of ABAC$_\alpha$.

With our tool we can specify a configuration type for every configuration of interest, with the command

DeclareCfgType(*typeId*,
$\{\texttt{UA}\rightarrow \{uAt_1,\ldots,uAt_m\}, \texttt{SA}\rightarrow \{sAt_1,\ldots,sAt_n\}, \texttt{OA}\rightarrow \{oAt_1,\ldots,oAt_p\},$
$\texttt{Scope}\rightarrow \{at_1 \rightarrow \{sId_1,\tau_1\},\ldots,at_r \rightarrow \{sId_r,\tau_r\}\}\})$

This declaration specifies a *configuration type* with identifier *typeId*, where

- $\{uAt_1,\ldots,uAt_m\}$ is the set of user attributes; $\{sAt_1,\ldots,sAt_n\}$ is the set of subject attributes, and $\{oAt_1,\ldots,oAt_p\}$ is the set of object attributes;
- the scope of every attribute at_i is the set bound to identifier sId_i in a particular configuration (see below), and its type is $\tau_i \in \{\texttt{elem},\texttt{subset}\}$, where `elem` stands for atomic and `subset` for set.

A *configuration* is an instance of a configuration type, which specifies (1) the configuration type which it instantiates; (2) the sets of values for the identifiers sId_i from the specification of the configuration type, and (3) the initial sets U, S, and O of entities (users, subjects, objects) in the configuration. In our system, the declaration of a concrete configuration of ABAC$_\alpha$ has the syntax

DeclareConfiguration(*cId*,
$\{\texttt{CfgType}\rightarrow \textit{typeId}, \texttt{Users}\rightarrow \{uId_1\rightarrow u_1,\ldots,uId_m\rightarrow u_m\},$
$\texttt{Range}\rightarrow \{\texttt{UId}\rightarrow \{uId_1,\ldots,uId_m\},$
$sId_2 \rightarrow \text{SCOPE}(at_2),\ldots,sId_r \rightarrow \text{SCOPE}(at_r)\},$
$\texttt{Subjects}\rightarrow \{s_1,\ldots,s_n\}, \texttt{Objects}\rightarrow \{o_1,\ldots,o_q\}\})$

Its side effect is to instantiate some globally visible entries:
CfgType(*cId*) with *typeId*,
Users(*cId*) with the set $\{u_1,\ldots,u_m\}$ of terms for users,
every User(*cId*,uId_i) with the term u_i,
Subjects(*cId*) with the set $\{s_1,\ldots,s_n\}$ of terms for subjects, and
Objects(*cId*) with the set $\{o_1,\ldots,o_q\}$ of terms for objects.

To illustrate, consider the mandatory access control model model (MAC). Users and subjects have a `clearance` attribute of type `elem`, whose value is a number from a finite set of integers $L = \{1,2,\ldots,N\}$, which indicates the security level of the corresponding entity. Objects have a `sensitivity` attribute of type `elem` whose value is also from L, and represents the sensitivity degree of the information in that object. When read and write are the only permissions on objects, we can assume the set of permissions P to be $\{\texttt{read},\texttt{write}\}$.

A configuration type for MAC can be defined as follows:

DeclareCfgType(MAC,
 {UA→ {id,clearance}, SA→ {id,clearance}, OA→ {sensitivity},
  Scope→{id →{uId,elem}, clearance →{level,elem},
                         sensitivity →{level,elem}}})

A particular MAC configuration can be defined by

DeclareConfiguration(MAC-Cfg01,
 {CfgType→MAC,
  Users→ {u1 → U(id(u),clearance(3)),
          u2 → U(id(u2),clearance(4))},
  Range→{uId→ {u1,u2}, level→ {1,2,3,4,5}},
  Subjects→ {S(id(u1),clearance(3)),
             S(id(u2),clearance(2))},
  Objects→ {O(sensitivity(1)),O(sensitivity(4))}})

This configuration allows five security levels for users and subjects, and five sensitivity levels for objects.

3.1 Rules for the policies of the configuration points

The constraint solving component of ρLog allows to specify and interpret correctly all formulas written in instances of the CPL scheme. Therefore, for every configuration type *typeId*, we can write rule-based specifications of the policies for the interaction between system entities:

- A user u can create a subject s if ConstrS(*typeId*)$::\{u,s\} \Longrightarrow$ true holds, where the defining rule of strategy ConstrS is of the form

  ConstrS(*typeId*)$::\{\mathtt{U}(\tilde{s}_1),\mathtt{S}(\tilde{s}_2)\} \Longrightarrow$ true $\leftarrow \phi_1$.

- A subject s can create an object o if ConstrO(*typeId*)$::\{s,o\} \Longrightarrow$ true holds, where the defining rule of strategy ConstrO is of the form

  ConstrO(*typeId*)$::\{\mathtt{S}(\tilde{s}_1),\mathtt{O}(\tilde{s}_2)\} \Longrightarrow$ true $\leftarrow \phi_2$.

- A user u can modify a subject s to become a subject s' if the reducibility formula ConstrModS(*typeId*)$::\{u,s,s'\} \Longrightarrow$ true holds, where the defining rule of strategy ConstrModS is of the form

  ConstrModS(*typeId*)$::\{\mathtt{U}(\tilde{s}_1),\mathtt{S}(\tilde{s}_2),\mathtt{S}(\tilde{s}_3)\} \Longrightarrow$ true $\leftarrow \phi_3$.

- A subject s can modify an object o to become an object o' if the reducibility formula ConstrModO(*typeId*)$::\{s,o,o'\} \Longrightarrow$ true holds, where the defining rule of strategy ConstrModO is of the form

  ConstrModO(*typeId*)$::\{\mathtt{S}(\tilde{s}_1),\mathtt{O}(\tilde{s}_2),\mathtt{O}(\tilde{s}_3)\} \Longrightarrow$ true $\leftarrow \phi_4$.

- A subject s is authorized to hold permission $p \in P$ on an object o if the reducibility formula $\texttt{Auth}(\textit{typeId}, p) :: \{s, o\} \Longrightarrow \texttt{true}$ holds, where the defining rule of strategy Auth is of the form

  $\texttt{Auth}(x, z) :: \{\texttt{S}(\tilde{s}_1), \texttt{O}(\tilde{s}_2)\} \Longrightarrow \texttt{true} \leftarrow \phi_{5,p}$.

In these rule-based specifications, ϕ_i and $\phi_{5,p}$ are formulas written in the instance of the CPL scheme for the values of the attributes of the interacting entities mentioned in the left-hand side of the corresponding rule.

For example, the mandatory access control (MAC) configuration type with read and write permissions can have the following rule-based specifications

$$\texttt{ConstrS}(\texttt{MAC}) :: \{\texttt{U}(x, \texttt{clearance}(y)), \texttt{S}(x, \texttt{clearance}(z))\} \Longrightarrow \texttt{true} \leftarrow (z \leq y).$$
$$\texttt{ConstrO}(\texttt{MAC}) :: \{\texttt{S}(\_, \texttt{clearance}(x)), \texttt{O}(\texttt{sensitivity}(y))\} \Longrightarrow \texttt{true} \leftarrow (x \leq y).$$
$$\texttt{ConstrModS}(\texttt{MAC}) :: \{\_, \_, \_\} \Longrightarrow \texttt{false} \leftarrow .$$
$$\texttt{ConstrModO}(\texttt{MAC}) :: \{\_, \_, \_\} \Longrightarrow \texttt{false} \leftarrow .$$
$$\texttt{Auth}(\texttt{MAC}, \texttt{read}) :: \{\texttt{S}(\_, \texttt{clearance}(x)), \texttt{O}(\texttt{sensitivity}(y))\} \Longrightarrow \texttt{true} \leftarrow (y \leq x).$$
$$\texttt{Auth}(\texttt{MAC}, \texttt{write}) :: \{\texttt{S}(\_, \texttt{clearance}(x)), \texttt{O}(\texttt{sensitivity}(y))\} \Longrightarrow \texttt{true} \leftarrow (x \leq y).$$

These policies do not allow to modify the attribute values of subjects and objects.

3.2 Rules for the operational model

3.2.1 Subject and object creation

These are nondeterministic operations: at any time, a user can create any subject whose attribute values satisfy the CPL-formula for the subject creation policy; similarly, a subject can create any object whose attribute values satisfy the CPL-formula for the object creation policy. These operations are implemented in two steps:

1. We use the auxiliary functions sSeed($\textit{cId}$) to compute the term
   $\texttt{S}(sAt_1(\text{SCOPE}(sAt_1), \tau_1), \ldots, sAt_n(\text{SCOPE}(sAt_n), \tau_n))$
   and oSeed($\textit{cId}$) which computes the term
   $\texttt{O}(oAt_1(\text{SCOPE}(oAt_1), \tau_1), \ldots, oAt_p(\text{SCOPE}(oAt_p), \tau_p))$,
   where τ_i is the corresponding attribute type.
   For example, for the MAC configuration MAC-Cfg01 illustrated before, the terms computed by sSeed(MAC-Cfg01) and oSeed(MAC-Cfg01) are
   $\texttt{S}(\texttt{id}(\{\texttt{u1}, \texttt{u2}\}, \texttt{elem}), \texttt{clearance}(\{1, 2, 3, 4, 5\}, \texttt{elem}))$ and
   $\texttt{O}(\texttt{sensitivity}(\{1, 2, 3, 4, 5\}, \texttt{elem}))$.
2. We use the terms computed by sSeed($\textit{cId}$) and oSeed($\textit{cId}$) as "seeds" to create any entity allowed by the creation policies. In rule-based thinking, an entity (subject or object) $K(att_1(v_1), \ldots, att_k(v_k))$ can be generated from the "seed" term

$K(att_1(scope_1,\tau_1),\ldots,att_k(scope_k,\tau_k))$ if and only if the reducibility formulas $scope_i \to_{\tau_i} v_i$ hold. Its implementation in ρLog is easy: If we define the auxiliary strategy

$\texttt{setAt}:: F_{at}(y_{scope},x_{type}) \Longrightarrow F_{at}(x) \leftarrow (x_{type} :: y_{scope} \Longrightarrow x)$.

then the set of entities that can be generated from a seed term st is the set of all e for which the reducibility formula $\texttt{fmap(setAt)} :: st \Longrightarrow e$ holds. Therefore, for a given ABAC$_\alpha$ configuration $\texttt{cId}$:

1) a user u can create a subject s if $\texttt{createS(}\texttt{cId}\texttt{)} :: u \Longrightarrow s$ holds, where the defining rule of the parametric strategy `createS` is

$$\begin{aligned}\texttt{createS}(x_{cId}) :: x_u \Longrightarrow x_s \leftarrow\ & (\texttt{fmap(setAt)} :: \text{sSeed}(x_{cId}) \Longrightarrow x_s)\ ,\\ & (\texttt{id} :: \text{UId}(x_u) \Longrightarrow \text{UId}(x_s))\ ,\\ & (\texttt{ConstrS}\ (\text{CfgType}(x_{cId})) :: \{x_u,x_s\} \Longrightarrow \texttt{true})\ .\end{aligned}$$

2) a subject s can create an object o if $\texttt{createO(}\texttt{cId}\texttt{)} :: s \Longrightarrow o$ holds, where the defining rule of the parametric strategy `createO` is

$$\begin{aligned}\texttt{createO}(x_{cId}) :: x_s \Longrightarrow x_o \leftarrow\ & (\texttt{fmap(setAt)} :: \text{oSeed}(x_{cId}) \Longrightarrow x_o)\ ,\\ & (\texttt{id} :: \text{UId}(x_u) \Longrightarrow \text{UId}(x_s))\ ,\\ & (\texttt{ConstrO}\ (\text{CfgType}(x_{cId})) :: \{x_s,x_o\} \Longrightarrow \texttt{true})\ .\end{aligned}$$

3.2.2 Modification of entity attributes

Users can try to modify the attributes of subjects created by them, and subjects can try to modify the attributes of objects. A simple way to model these operations for an ABAC$_\alpha$ configuration $\texttt{cId}$ of type $\texttt{typeId}$ is as follows:

3) Modification of the attribute values of a subject s by a user u can be viewed as generating a subject s' for which $\texttt{ConstrModS(}\texttt{typeId}\texttt{)} :: \{u,s,s'\} \Longrightarrow \texttt{true}$ holds. The outcome of changing the attribute values of s is s'. We define

$$\begin{aligned}\texttt{modSA}\,(x_{cId}) :: \{x_u,x_s\} \Longrightarrow x'_s \leftarrow\ & (\texttt{fmap}\,(\texttt{setAt} :: \text{sSeed}(x_{cId}) \Longrightarrow x'_s),\\ & (\texttt{id} :: \text{UId}(x_u) \Longrightarrow \text{UId}(x_s))\ ,\ (\texttt{id} :: \text{UId}(x_s) \Longrightarrow \text{UId}(x'_s))\ ,\\ & (\texttt{ConstrModS}\,(\text{CfgType}(x_{cId}) :: \{x_u,x_s,x'_s\} \Longrightarrow \texttt{true})\ .\end{aligned}$$

and note that $\texttt{modSA(}\texttt{cId}\texttt{)} :: s \Longrightarrow s'$ holds if and only if the user u who created subject s is allowed to modify the attribute values of s to become s'.

4) Modification of the attribute values of an object o by a subject s can be viewed as generating an object o' for which $\texttt{ConstrModO(}\texttt{typeId}\texttt{)} :: \{s,o,o'\} \Longrightarrow \texttt{true}$ holds. The outcome of changing the attribute values of o is o'. We define

$$\begin{aligned}\texttt{modOA}\,(x_{cId}) :: \{x_s,x_o\} \Longrightarrow x'_o \leftarrow\ & (\texttt{fmap}\,(\texttt{setAt} :: \text{oSeed}(x_{cId}) \Longrightarrow x'_o),\\ & (\texttt{ConstrModO}\,(\text{CfgType}(x_{cId}) :: \{x_s,x_o,x'_o\} \Longrightarrow \texttt{true})\ .\end{aligned}$$

3.2.3 State transitions

A system with an ABAC$_\alpha$ access control model can be viewed as a state transition system whose states are triples $\{U,S,O\}$ consisting of the existing users (U), subjects (S), and objects (O), and whose transitions correspond to the six operations controlled by the policies of ABAC$_\alpha$.

Except for authorized access, the other five operations from the functional specification of ABAC$_\alpha$ determine state transitions. Their rule-based specifications are:

$\texttt{createSubj}(x_{cId}) :: \{\{\overline{x},x_u,\overline{y}\},x_S,x_O\} \Longrightarrow$
$\quad \{\{\overline{x},x_u,\overline{y}\},x_S \cup \{x_s\},x_O\} \leftarrow (\texttt{createS}(x_{cId}) :: x_u \Longrightarrow x_s), x_s \notin x_S.$
$\texttt{deleteSubj}(\_) :: \{\{\overline{x}_1,x_u,\overline{x}_2\},\{\overline{y}_1,x_s,\overline{y}_2\},x_O\} \Longrightarrow$
$\quad \{\{\overline{x}_1,x_u,\overline{x}_2\},\{\overline{y}_1,\overline{y}_2\},x_O\} \leftarrow (\texttt{id} :: \mathrm{UId}(x_u) \Longrightarrow \mathrm{UId}(x_s)).$
$\texttt{createObj}(x_{cId}) :: \{x_U,\{\overline{x},x_s,\overline{y}\},x_O\} \Longrightarrow$
$\quad \{x_U,\{\overline{x},x_s,\overline{y}\},x_O \cup \{x_o\}\} \leftarrow (\texttt{createO}(x_{cId}) :: x_s \Longrightarrow x_o), x_o \notin x_O.$
$\texttt{modifySubj}(x_{cId}) :: \{x_U,\{\overline{x},x_s,\overline{y}\},x_O\} \Longrightarrow$
$\quad \{x_U,\{\overline{x},x'_s,\overline{y}\},x_O\} \leftarrow (\texttt{modSA}(x_{cId}) :: \{x_U,x_s\} \Longrightarrow x'_s).$
$\texttt{modifyObj}(x_{cId}) :: \{x_U,\{\overline{x}_1,x_s,\overline{x}_2\},\{\overline{y}_1,x_o,\overline{y}_2\}\} \Longrightarrow$
$\quad \{x_U,\{\overline{x}_1,x_s,\overline{x}_2\},\{\overline{y}_1,x'_o,\overline{y}_2\}\} \leftarrow (\texttt{modOA}(x_{cId}) :: \{x_s,x_o\} \Longrightarrow x'_o).$

In the state transitions defined by these rules, the entities matched by x_u,x_s,x_o are those who interact during the rule application.

3.3 Safety analysis

Safety is a fundamental problem for any protection system. The safety problem for ABAC$_\alpha$ asks whether a subject s can obtain permission p for an object o. Recently, it has been shown that this problem is decidable [1], by identifying a state-matching reduction from ABAC$_\alpha$ to the pre-authorization usage control model with finite attribute domains (UCON$^{\text{finite}}_{\text{preA}}$). The result follows from the facts that (1) the safety problem of UCON$^{\text{finite}}_{\text{preA}}$ is decidable [12], and (2) state-matching reductions, like the one defined in [1], preserve security properties including safety. It provides an indirect way to implement an algorithm to decide the safety problem of ABAC$_\alpha$. In [10] we noticed that this indirection can be avoided: a direct analysis of the operational model of ABAC$_\alpha$ revealed the main reasons when a configuration is unsafe. In this section we recall the theoretical results reported in [10], and illustrate how to use ρLog to turn our theoretical findings into rule-based specifications that can be directly executed. We claim that our approach is a natural and effective way to solve the safety problem for any configuration of ABAC$_\alpha$.

3.3.1 Properties of ABAC$_\alpha$ derivations

We start from the state transition view of the operational model described in Sect. 3.2.3. If $e \in S \cup O$ then a derivation $\Pi : St = \{U,S,O\} \Longrightarrow \ldots \Longrightarrow \{U,S',O'\}$ whose transition steps do not delete e may modify the attributes values of e. To analyze the possible changes of the attribute values of e in ABAC$_\alpha$, we introduce the auxiliary notion of *descendant* of e in Π: $desc_\Pi(e)$ is the entity $e' \in S' \cup O'$ which represents e after performing the operations $op_1, \ldots, op_n$ in this order. Another useful auxiliary notion is $Desc^{St}(e) = \{desc_\Pi(e) \mid \Pi : St \Longrightarrow^* \{U,S',O'\}\}$.

With these preparations, the safety problem for ABAC$_\alpha$ is

Given an ABAC$_\alpha$ configuration `cId` with initial state $St = \{U,S,O\}$, a subject $s \in S$, an object $o \in O$, and a permission $p \in P$,

Decide if there is a derivation $\Pi : St \Longrightarrow \ldots \Longrightarrow \{U,S',O'\}$ whose transitions steps do not delete the descendants of s, such that subject $desc_\Pi(s)$ can be authorized to obtain permission p on object $desc_\Pi(o)$. Formally, this means that the formula $\texttt{Auth}(\texttt{typeId}, p) :: \{desc_\Pi(s), desc_\Pi(o)\} \Longrightarrow \texttt{true}$ holds, where `typeId` is the configuration type of `cId`.

In this state transition system, objects can only participate at changing their own attributes. Therefore, objects from $O - \{o\}$ do not affect the truth value of the formula $\texttt{Auth}(\texttt{typeId}, p) :: \{desc_\Pi(s), desc_\Pi(o)\} \Longrightarrow \texttt{true}$. Hence it is harmless to assume that the initial state is $\{U,S,\{o\}\})$ and Π has no object creation steps. Also, if $\{U,S,O\} \Longrightarrow \{U,S',O'\}$ then $\{U,S \cup S'',O'\} \Longrightarrow \{U,S \cup S'',O'\}$ holds too, because we can choose the same participating entities to perform the transition. Therefore, we can assume that Π has no subject deletion steps.

Thus, we can assume without loss of generality that the safety problem is

Given an ABAC$_\alpha$ configuration `cId` with initial state $St_0 = \{U,S,\{o\}\}$ with $s \in S$, and a permission $p \in P$,

Decide UNSAFE if there is a derivation $\Pi : St \rightarrow^* (U,S',\{o'\})$ without subject deletion and object creation steps, such that the reducibility formula

$$\texttt{Auth(CfgType(cId), } p\texttt{)} :: \{desc_\Pi(s), o'\} \Longrightarrow \texttt{true}$$

holds, and SAFE otherwise.

By [10, Theorem 1], the answer is UNSAFE if and only if there exist $s' \in Desc^{St}(s)$ and $o' \in Desc^{St}(o)$ such that $\texttt{Auth(typeId, } p\texttt{)} :: \{s',o'\} \Longrightarrow \texttt{true}$ holds. In ABAC$_\alpha$, all attributes assume values from finite sets specified for `cId`, therefore $Desc^{St}(s)$ and $Desc^{St}(o)$ are finite sets that can be computed. Based on this observation, we designed a safety decision algorithm that computes incrementally the finite sets $Desc^{St}(s)$ and $Desc^{St}(o)$, and interleaves their computation with testing if $\texttt{Auth(typeId, } p\texttt{)} :: \{s',o'\} \Longrightarrow \texttt{true}$ holds for some $s' \in Desc^{St}(s)$ and $o' \in Desc^{St}(o)$.

3.3.2 A rule-based safety decision algorithm

Suppose u is the creator of s. If $u \notin U$ then $Desc^{St}(s) = \{s\}$, otherwise $Desc^{St}(s) = \bigcup_{k=1}^{\infty} S_k$ where $S_1 = \{s\}$ and

$$S_{n+1} = \left\{ s'' \notin \bigcup_{k=1}^{n} S_k \,\middle|\, \exists s' \in \bigcup_{k=1}^{n} S_k.(\texttt{ModSA}\,(\,cId\,) :: \{u, s'\} \Longrightarrow s'') \right\} \quad \text{if } n \geq 1.$$

Because $Desc^{St}(s)$ is finite, $Desc^{St}(s) = \bigcup_{k=1}^{n_0} S_k$ where $n_0 = \min\{n \in \mathbb{N} \mid S_n = \emptyset\}$. The partition $\{S_k \mid 1 \leq k \leq n_0\}$ of $Desc^{St}(s)$ can be computed iteratively: $S_1 = \{s\}$, and $S_{k+1} = \text{ApplyRuleList}(\texttt{nextS}(cId, \bigcup_{i=1}^{k} S_i), \{U, S_k\})$ where the parametric strategy `nextS` is defined by the rule

$$\texttt{nextS}(x_{cId}, x_S) :: \{\{\_\_, x_u, \_\_\}, \{\_\_, x_s, \_\_\}\} \Longrightarrow \\ x'_s \leftarrow (\texttt{modSA}(x_{cId}) :: \{x_u, x_s\} \Longrightarrow x'_s), x'_s \notin x_S.$$

We can speed up the safety decision algorithm by interleaving the computation of every S_k with testing if $\texttt{Auth}(\text{CfgType}(cId), p) :: \{s', o\} \Longrightarrow \texttt{true}$ holds for some $s' \in S_k$. We can do this test by checking if $\text{ApplyRule}(\texttt{auth?}(p, cId), \{S_k, \{o\}\})$ yields `true`, where the parametric strategy `auth?` is defined by the rule

$$\texttt{auth?}\,(x_p, x_{cId}) :: \{\{\_\_, x_s, \_\_\}, \{\_\_, x_o, \_\_\}\} \Longrightarrow \texttt{true} \leftarrow \\ \texttt{Auth}(\text{CfgType}(x_{cId}, x_p)) :: \{x_s, x_o\} \Longrightarrow \texttt{true})\ .$$

As soon as any of these tests yields `true`, the decision algorithm stops by returning UNSAFE. Otherwise, we end up computing the set $Desc^{St}(s)$ and will start computing $Desc^{St}(o)$. The computation of this set can proceed in two steps:

1. First, we compute the set S_{all} of all subjects that can show up in the system: $S_{all} = \bigcup_{k=1}^{\infty} S_k$ where $S_1 = S$, S_2 is the set of all subjects that can be created by users in U, and

$$S_{n+1} = \left\{ s'' \notin \bigcup_{k=1}^{n} S_k \,\middle|\, \exists u \in U. \exists s' \in \bigcup_{k=1}^{n} S_k.(\texttt{ModSA}\,(\,cId\,) :: \{u, s'\} \Longrightarrow s'') \right\}$$

   If $n \geq 2$. Because S_{all} is finite, $S_{all} = \bigcup_{k=1}^{n_1} S_k$ where $n_1 = \min\{n \geq 2 \mid \wedge S_n = \emptyset\}$. The partition $\{S_k \mid 1 \leq k \leq n_1\}$ of S_{all} can be computed incrementally:

$$S_2 = \bigcup_{u \in U} \text{ApplyRuleList}(\texttt{createS}\,(\,cId\,), u)$$

$$S_{n+1} = \text{ApplyRuleList}(\texttt{nextS}(cId, \bigcup_{k=1}^{n} S_k), \{U, S_k\}) \qquad \text{if } n \geq 2.$$

2. $Desc^{st}(o) = \bigcup_{k=1}^{\infty} O_k$ where $O_1 = \{o\}$ and

$$O_{n+1} = \left\{ o'' \notin \bigcup_{k=1}^{n} o_k \,\middle|\, \exists s' \in S_{all}. \exists o' \in \bigcup_{k=1}^{n} O_k.(\texttt{ModOA}\,(\,cId\,) :: \{s', o'\} \Longrightarrow o'') \right\}$$

if $n \geq 1$. Since $Desc^{st}(o)$ is finite, $Desc^{st}(o) = \bigcup_{k=1}^{n_2} O_k$ where $n_2 = \min\{n \in \mathbb{N} \mid O_n = \emptyset\}$.

With ρLog, it is easy to compute incrementally the partition $\{O_k \mid 1 \leq k \leq n_2\}$ of $Desc^{st}(o)$: for every $k \geq 1$ we have

$$O_{k+1} = \text{ApplyRuleList}(\texttt{nextO}(cId), \bigcup_{i=1}^{k} O_i), \{S_{all}, O_k\})$$

where the parametric strategy `nextO` is defined by the rule

$$\texttt{nextS}(x_{cId}, x_O) :: \{\{\_\_, x_s, \_\_\}, \{\_\_, x_o, \_\_\}\} \Longrightarrow \\ x'_o \leftarrow (\texttt{modOA}(x_{cId}) :: \{x_s, x_o\} \Longrightarrow x'_o), x'_o \notin x_O.$$

Here, again, we can speed up the safety decision algorithm by interleaving the computation of every O_k with testing if $\texttt{Auth}(\text{CfgType}(cId), p) :: \{s', o'\} \Longrightarrow \texttt{true}$ holds for some $s' \in S_{all}$ and $o' \in O_k$. We can do this test by checking if the request $\text{ApplyRule}(\texttt{auth?}(p, cId), S_{all}, O_k)$ yields `true`. As soon as this happens, the algorithm stops by returning UNSAFE. Otherwise, we stop and return SAFE.

4 Conclusion

State-matching reduction [15] is a powerful technique to prove security properties (including safety) of state transition systems. This indirect way to define an algorithm for the safety problem of ABAC_α configurations makes hard to observe some important properties that can be used to improve its performance. The direct rule-based analysis performed by us has the following advantages:

1. It provides a unified framework to specify policies for ABAC_α configurations, the operational model, execute them, and verify some security properties, including safety.
2. It allowed us to detect some useful properties of the transition model, that simplified significantly de design of our decision algorithm for safety. In particular, it allowed us to reduce the safety problem of to a simpler one: check if $\texttt{Auth}(\text{CfgType}(cId), p) :: \{s', o'\} \Longrightarrow \texttt{true}$ holds for some $s' \in Desc^{St}(s)$ and $o' \in Desc^{St}(o)$. We solved it by identifying rule-based algorithms that interleave detection of unsafety with the incremental computation of $Desc^{St}(s)$ and $Desc^{St}(o)$.
3. With ρLog, we turned such a rule-based specification into executable code and obtained a practical tool to check the safety of any configuration of interest. The rule-based specification is parametric with respect to the configuration types of ABAC_α. Therefore, whenever we want to check that, for a given configuration, a subject s never gets permission p on an object o, it is enough to do the following:

   a. specify the configuration and its type, as indicated in Sect. 3.

b. call the method CheckSafety($\texttt{cfgId}, s, o, p$) which runs our safety-check algorithm. It returns SAFE if s never gets permission p on o, and UNSAFE otherwise.

There are many other rule-based systems with support for strategic programming, that can be used to formalize state transition systems and study their properties. But ρLog has some outstanding capabilities for this purpose:

1. It has four kinds of variables which give the user flexible control to select the components of the term which is transformed. The code is usually quite short and declaratively clear, as witnessed by the rule-based specification of ABAC$_\alpha$.
2. It inherits from the Wolfram language of Mathematica a rich variety of constraints that can be used in requests and the conditional parts of rules. In particular, the boolean formulas that constrain the operations of ABAC$_\alpha$ have direct translations as constraints in the CLP component of ρLog.
3. It can generate human-readable traces of the reductions that yield an answer. For the safety problems of ABAC$_\alpha$, this capability could be used to produce scenarios that indicate the sequence of transitions that yield a state where a subject s can exercise a permission p on an object o. This capability could become a useful tool to detect security holes of ABAC$_\alpha$ configurations, and to fix them. We leave the extension of our a tool with this capability as a direction of future work.

Acknowledgements This work was supported by Shota Rustaveli National Science Foundation of Georgia under the grant no. FR17_439 and by the Austrian Science Fund (FWF) under the project P 28789-N32

References

1. T. Ahmed and R. Sandhu. Safety of ABAC$_\alpha$ Is Decidable. In Z. Yan, R. Molva, W. Mazurczyk, and R. Kantola, editors, *Network and System Security*, pages 257–272. Springer International Publishing, 2017.
2. K. R. Apt and M. H. van Emden. Contributions to the Theory of Logic Programming. *JACM*, 29(3):841–862, 1982.
3. B. Buchberger and J. A. Campbell, editors. *Proceedings of Artificial Intelligence and Symbolic Computation (AISC 2004)*, volume 3249 of *LNCS*. Springer, 2004.
4. D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
5. V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to Attribute Based Access Control (ABAC) Definition and Considerations, 2014. NIST Special Publication 800-162.
6. X. Jin. *Attribute-Based Access Control Models and Implementation in Cloud Infrastructure as a Service*. PhD thesis, University of Texas at San Antonio, 2014.
7. X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro, editors, *Data and Applications Security and Privacy XXVI*, volume 7371 of *LNCS*, pages 41–55. Springer, Berlin, Heidelberg, 2012.

8. T. Kutsia and M. Marin. Can context sequence matching be used for querying XML? In L. Vigneron, editor, *Proceedings of the 19th International Workshop on Unification (UNIF'05)*, pages 77–92, Nara, Japan, 22 Apr. 2005.
9. M. Marin and T. Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.
10. M. Marin, T. Kutsia, and B. Dundua. A Rule-based Approach to the Decidability of $ABAC_\alpha$. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*, SACMAT 2019, pages 173–178, New York, NY, USA, 2019. Association for Computing Machinery.
11. M. Marin and F. Piroi. Rule-Based Programming with Mathematica. In *Proceedings of International Mathematica Symposium (IMS 2004)*, Banff, Canada, 2004.
12. P. V. Rajkumar and R. Sandhu. Safety decidability for pre-authorization usage control with identifier attribute domains. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.
13. R. S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.
14. R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.
15. M. V. Tripunitara and N. Li. A theory for comparing the expressive power of access control models. *J. Comput. Secur.*, 15(2):231–272, 2007.
16. S. Wolfram. *The Mathematica Book*. Wolfram Media, 5th edition, 2003.