A Rule-based Approach to the Decidability of Safety of ABAC$_\alpha$

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

Temur Kutsia
RISC, Johannes Kepler University
Linz, Austria
kutsia@risc.jku.at

Besik Dundua
VIAM, Tbilisi State University and
International Black Sea University
Tbilisi, Georgia
bdundua@gmail.com

ABSTRACT

ABAC$_\alpha$ is a foundational model for attribute-based access control with a minimal set of capabilities to configure many access control models of interest, including the dominant traditional ones: discretionary (DAC), mandatory (MAC), and role-based (RBAC). A fundamental security problem in the design of ABAC is to ensure safety, that is, to guarantee that a certain subject can never gain certain permissions to access certain object(s).

We propose a rule-based specification of ABAC$_\alpha$ and of its configurations, and the semantic framework of ρLog to turn this specification into executable code for the operational model of ABAC$_\alpha$. Next, we identify some important properties of the operational model which allow us to define a rule-based algorithm for the safety problem, and to execute it with ρLog. The outcome is a practical tool to check safety of ABAC$_\alpha$ configurations.

ρLog is a system for rule-based programming with strategies and built-in support for constraint logic programming (CLP). We argue that ρLog is an adequate framework for the specification and verification of safety of ABAC$_\alpha$ configurations. In particular, the authorization policies of ABAC$_\alpha$ can be interpreted properly by the CLP component of ρLog, and the operations of its functional specification can be described by five strategies defined by conditional rewrite rules.

CCS CONCEPTS

• Security and privacy → Access control; **• Theory of computation → Rewrite systems**; **Logic and verification**; **Constraint and logic programming**.

KEYWORDS

attribute based access control (ABAC), rule-based programming, safety

ACM Reference Format:
Mircea Marin, Temur Kutsia, and Besik Dundua. 2019. A Rule-based Approach to the Decidability of Safety of ABAC$_\alpha$. In *The 24th ACM Symposium on Access Control Models and Technologies (SACMAT '19), June 3–6, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3322431.3325416

1 INTRODUCTION

Access control is a fundamental security requirement for computing environments: It controls the ability of a subject to use an object in some specific manner. A subject represents a user and any system process or entity that acts on behalf of a user. Users represent individuals who interact directly or indirectly with a system and have been authenticated and established their identities. Objects are the protected entities, and can represent either system abstractions (e.g., processes, files, or ports) or system resources (e.g., printers).

Attribute-based access control (ABAC) is a logical access control with great flexibility to specify access control policies as rules which get evaluated against the attributes of participating entities (user/subject or subject/object), operations, and the environment relevant to a request. Considerable work has been done and a number of formal models have been proposed recently for ABAC [3, 4, 11]. Among them, ABAC$_\alpha$ became popular because of its minimal set of features that make it powerful enough to configure the dominant traditional access control models DAC, MAC and RBAC.

A fundamental security problem in the design of ABAC models is safety. According to [2], the safety problem for protection systems is to determine in a given situation whether a subject can acquire a particular right to an object. Recently, it was shown that safety of ABAC$_\alpha$ is decidable [1]. The proof was based on state-matching reduction of safety of ABAC$_\alpha$ to safety of the preauthorization model UCON$^{\text{finite}}_{\text{preA}}$, which is known to be decidable [12].

In this paper we describe a rule-based algorithm to decide safety of ABAC$_\alpha$. We use the rule-based programming system ρLog [9, 10], which has adequate support (1) to define ABAC$_\alpha$ configurations in its constraint logic programming component, (2) to specify the operations of ABAC$_\alpha$ with rules, and (3) to define strategies which control the application of these rules and enable a rule-based implementation of a decision algorithm for the safety of ABAC$_\alpha$.

Section 2 contains a brief description of ρLog. In Sect. 3 we describe the ABAC$_\alpha$ model and our representation of ABAC$_\alpha$ configurations. In Sect. 4 we present our rule-based specification for the operations of ABAC$_\alpha$ and for the configuration-specific conditions that constrain their execution. Section 5 contains our main contributions: (1) a rule-based specification of an algorithm which can decide the safety problem for configurations of ABAC$_\alpha$, and can be executed efficiently in ρLog; and (2) theoretical results which guarantee the correctness of our algorithm. Section 6 concludes.

2 THE ρLOG SYSTEM

ρLog [8–10] is a system for rule-based programming developed on top of the rewriting and constraint solving capabilities of Mathematica [13]. It provides (1) a logical framework to reason in theories whose deduction rules can be specified by conditional rewrite rules of a very general kind, and (2) a semantic framework where computations are sequences of state transitions modelled as rewrite steps controlled by strategies. In this section, we focus on the use of ρLog as semantic framework to express possible evolutions of state transition systems.

A program consists of **rules** $s \rightarrow_{stg} t /; cond_1 \wedge \ldots \wedge cond_n$ with the intended reading "s reduces to t with strategy stg (notation $s \rightarrow_{stg} t$) whenever $cond_1$ and ... and $cond_n$ hold." Such a rule is a partial definition for strategy stg. To illustrate how reduction works, suppose states are lists of numbers, and we wish to define transitions that swap list elements which are not in increasing order. In ρLog syntax, the labeled rule for such transitions is

`{a___,x_,b___,y_,c___}` $\rightarrow_{\text{"sw"}}$ `{a,y,b,x,c}/;(x>y).`

where $\{\ldots\}$ is the list constructor, and "sw" is the strategy for this kind of reduction. Note the following peculiarities of ρLog:

(1) `a`, `x`, `b`, `y`, `c` are variables. They are identified by suffixing their first occurrences[1] in the rule with `_` or with `___`,

(2) We used two kinds of variables: those suffixed by `_` are placeholders for one element, and are called **ordinary variables**; those suffixed by `___` are placeholders for a sequence of elements, and are called **sequence variables**. Sequence variables are a novel capability of some programming languages, which increases the expressive power and conciseness of rule-based specifications. ρLog allows the use of anonymous variables: `_` is a nameless placeholder for an element, and `___` is a nameless placeholder for a sequence of elements.

(3) `(x>y)` is a boolean condition that is properly interpreted by the constraint logic programming component (CLP) of ρLog.

Thus, rewriting with this rule swaps list elements `x` and `y` if `x>y` and `x` occurs before `y` in the list. Repeated applications of such rewrite steps will eventually yield the sorted version of any list of numbers.

Sequence variables introduce nondeterminism in the rewriting process: For example, there are three ways to reduce $\{3,2,1\}$ with the labeled rule for "sw": $\{3,2,1\} \rightarrow_{\text{"sw"}} \{2,3,1\}$ with matcher $\{\mathtt{a} \mapsto \ulcorner\urcorner, x \mapsto 3, b \mapsto \ulcorner\urcorner, y \mapsto 2, c \mapsto \ulcorner 1 \urcorner\}$; $\{3,2,1\} \rightarrow_{\text{"sw"}} \{1,2,3\}$ with matcher $\{\mathtt{a} \mapsto \ulcorner\urcorner, \mathtt{x} \mapsto 3, \mathtt{b} \mapsto \ulcorner 2 \urcorner, \mathtt{y} \mapsto 1, \mathtt{c} \mapsto \ulcorner\urcorner\}$; $\{3,2,1\} \rightarrow_{\text{"sw"}} \{3,1,2\}$ with matcher $\{\mathtt{a} \mapsto \ulcorner 3 \urcorner, \mathtt{x} \mapsto 2, \mathtt{b} \mapsto \ulcorner\urcorner, \mathtt{y} \mapsto 1, \mathtt{c} \mapsto \ulcorner\urcorner\}$, where $\ulcorner e_1 \ \ldots \ e_n \urcorner$ represents the sequence of elements $e_1, \ldots, e_n$, in this order. This nondeterminism is due to the fact that matching with sequence variables is finitary [5, 6]. Algorithms which enumerate all finitely many matchers with terms containing such variables are described in [7].

In general, the conditional part of a rule is a conjunction of constraints of three kinds: (1) reducibility formulas $s \rightarrow_{stg} t$, (2) irreducibility formulas $s \nrightarrow_{stg} t$, (3) any boolean formulas expressed in the host language of Mathematica.

ρLog is designed to work with **strategies** of three kinds:

(1) Atomic strategies, designated by a string identifier sId, and defined by one or more labeled rules of the form

$$s \rightarrow_{\mathrm{sId}} t /; cond_1 \wedge \ldots \wedge cond_n.$$

The following atomic strategies are predefined:

"Id": $s \rightarrow_{\text{"Id"}} t$, abbreviated $s \equiv t$, which holds if $s = t$.

"elem": $l \rightarrow_{\text{"elem"}} e$ holds if e is an element of list l.

"subset": $l \rightarrow_{\text{"subset"}} s$ holds if s is subset of set l.

(2) Composite strategies, built from other strategies with combinators. The following strategy combinators are predefined:

1) $s \rightarrow_{stg_1 \circ stg_2} t$ holds if $s \rightarrow_{stg_1} u \rightarrow_{stg_2} t$ for some u.

2) $s \rightarrow_{stg_1 | stg_2} t$ holds if either $s \rightarrow_{stg_1} t$ or $s \rightarrow_{stg_2} t$.

3) $s \rightarrow_{stg^*} t$ holds if either $s \rightarrow_{\text{"Id"}} t$ or there exist $u_1, \ldots, u_n$ such that $s \rightarrow_{stg} u_1 \rightarrow_{stg} \ldots \rightarrow_{stg} u_n \rightarrow_{stg} t$.

4) $s \rightarrow_{\mathrm{Fst}[stg_1,\ldots,stg_n]} t$ holds if there exists $1 \le i \le n$ such that $s \rightarrow_{stg_i} t$ and $s \nrightarrow_{stg_j} t$ hold for $1 \le j < i$.

5) $s \rightarrow_{\mathrm{NF}[stg]} t$ holds if both $s \rightarrow_{stg^*} t$ and $t \nrightarrow_{stg}$ _ hold.

(3) Parametric strategies, defined by rules of the form

$$s \rightarrow_{\mathrm{sId}[s_1,\ldots,s_m]} t /; cond_1 \wedge \ldots \wedge cond_n$$

where sId is the strategy identifier (a string) and $s_1, \ldots, s_m$ are its parameters. The parameters provide syntactic material to be used in the conditional part of the rule. A useful predefined strategy is `"fmap"`: $f[s_1, \ldots, s_n] \rightarrow_{\text{"fmap"}[stg]} f[t_1, \ldots, t_n]$ holds if $s_i \rightarrow_{stg} t_i$ for $1 \le i \le n$.

The command to add a rule *rule* to the current program of a ρLog session is `DeclareRule[`*rule*`]`.

Queries are requests of the form

`Request[`$cond_1 \wedge \ldots \wedge cond_n$`]` or `RequestAll[`$cond_1 \wedge \ldots \wedge cond_n$`]`

They instruct ρLog to compute one (resp. all) substitution(s) for the variables in the formula $cond_1 \wedge \ldots \wedge cond_n$ for which it holds with respect to the current program. For example, if the current program contains the previous definition of `"sw"`, and we want to compute the substitution(s) for which $\{3,2,1\} \rightarrow_{\text{"sw"}} \mathtt{x}$ holds, we can call

`Request[`$\{3,2,1\} \rightarrow_{\text{"sw"}}$ `x_]` or `RequestAll[`$\{3,2,1\} \rightarrow_{\text{"sw"}}$ `x_]`

The answer to the first query will be $\{\mathtt{x} \rightarrow \{2,3,1\}\}$, and that to the second one is $\{\{\mathtt{x} \rightarrow \{2,3,1\}\}, \{\mathtt{x} \rightarrow \{1,2,3\}\}, \{\mathtt{x} \rightarrow \{3,1,2\}\}\}$.

Another use of ρLog is to compute a reduct of a term with respect to a strategy. The request `ApplyRule[`stg, s`]` instructs ρLog to compute one (if any) reduct of s with respect to strategy stg, that is, a term t such that formula $s \rightarrow_{stg} t$ holds. ρLog reports `"no solution found."` if there is no reduct of s with stg. ρLog can also be instructed to find all reducts of a term with respect to a strategy, with `ApplyRuleList[`stg, s`]`.

More information about ρLog can be found at

`http://staff.fmi.uvt.ro/~mircea.marin/rholog/`

3 ABAC$_\alpha$

ABAC$_\alpha$ [3] is a formal model of ABAC with a minimal set of features to configure the traditional models DAC, MAC, and RBAC. Its core components are entities of three kinds: users, subjects, and objects. 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.

Every kind of entity has a fixed set of attributes. Every attribute has a type, scope, and range of possible values. The sets of attributes specific to each kind of entity, together with their 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 <u>atomic</u> or <u>set</u>. The scope of each attribute at is a finite set SCOPE(at) of values. If at

[1] By 'first occurrences' of a variable in a rule we mean all its occurrences in the expression from the sequence (stg, s), $cond_1, \ldots, cond_n, t$ where it occurs first.

is of atomic type, then it 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 at is SCOPE(at) if at is of atomic type, and $2^{\text{SCOPE}(at)}$ otherwise.

User creation, attribute value assignment of user at creation time, user deletion and modification of a user's attribute values are operations done by the security administrator and are outside the scope of ABAC$_\alpha$. Therefore, in a running configuration of ABAC$_\alpha$, the set U of existing users is fixed, but the sets S of existing subjects and O of existing objects may change. The identity of every user is specified by the value of a special attribute "id" of atomic type.

In our system, configuration types are declared as follows:

```
DeclareCfgType[typeId,{"UA"→ {uAt1,...,uAtm},
     "SA"→ {sAt1,...,sAtn},"OA"→ {oAt1,...,oAtp},
     "Scope"→ {at1 → •[sId1,τ1],...,atr → •[sIdr,τr]}}]
```

Such a declaration specifies a configuration type with unique identifier `typeId` (a string), where: $\{uAt_1, \ldots, uAt_m\}$, $\{sAt_1, \ldots, sAt_n\}$, and $\{oAt_1, \ldots, oAt_p\}$ are the sets of attributes for users, subjects, and objects, respectively; $\{at_1, \ldots, at_r\}$ is their union; the scope of every attribute at_i (a string) is the set bound to identifier sId_i in a particular configuration (see below), and its type is $\tau_i \in$ {"elem", "subset"}, where "elem" stands for atomic and "subset" for set.

For example, the configuration type of the discretionary access model (DAC) is given by

```
DeclareCfgType["DAC",{"UA"→{"id"},"SA"→{"id"},
  "OA"→{"id","r","w"},"Scope"→{"id"→•["UId","elem"],
     "r"→ •["UId","subset"],"w"→ •["UId","subset"]}}]
```

We represent the system entities as first-order terms of the form

$\bullet$U[$uAt_1[v_1], \ldots, uAt_m[v_m]$]	(* users *)
$\bullet$S[$sAt_1[v_1], \ldots, sAt_n[v_n]$]	(* subjects *)
$\bullet$O[$oAt_1[v_1], \ldots, oAt_p[v_p]$]	(* objects *)

where v_i are the corresponding attribute values.

Every user has a unique ID, which is the string value of its "id" attribute. Subjects keep track of the ID of their creator in the value of their "id" attribute. We will assume without loss of generality that $uAt_1 = sAt_1 =$ "id", and that there is a function UId[e] which returns the value of the attribute "id" of $e \in U \cup S$.

A **configuration** of ABAC$_\alpha$ 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,
 {"CfgType"→typeId,
  "Users"→{uId1→u1,...,uIdm→um},
  "Range"→{"UId"→{uId1,...,uIdm},
               sId2 → SCOPE[at2],...,sIdr → SCOPE[atr]},
  "Subjects"→{s1,...,sn},"Objects"→{o1,...,oq}}]
```

Its side effect is to instantiate some globally visible entries: `CfgType[cId]` with `typeId`, `Users[cId]` with the set of users $\{u_1, \ldots, u_m\}$, every `User[cId,`uId_i`]` with user u_i, `Subjects[cId]` with the set of subjects $\{s_1, \ldots, s_n\}$, `Objects[cId]` with the set of objects $\{o_1, \ldots, o_q\}$. For example, the following is a declaration of a DAC configuration:

```
DeclareConfiguration["Cfg-01",{"CfgType"→"DAC",
 "Users"→{"u1"→ •U["id"["u1"]],"u2"→ •U["id"["u2"]],
         "u3"→ •U["id"["u3"]]},
 "Range"→{"UId"→{"u1","u2","u3"}},
 "Subjects"→{•S["id"["u1"]],•S["id"["u3"]]},
 "Objects"→{
   •O["id"["u1"],"r"[{"u1","u3"}],"w"[{"u1","u2"}]],
   •O["id"["u1"],"r"[{"u1","u3"}],"w"[{"u2","u3"}]]}}]
```

This configuration specifies an initial system state with three users identified by strings "u1", "u2", "u3"; two subjects; and two objects. The first object grants read access to subjects created by users "u1", "u3" and write access to subjects created by users "u1", "u2". The second object grants read access to subjects created by users "u1", "u3" and write access to subjects created by users "u2", "u3".

4 A STATE TRANSITION SYSTEM FOR ABAC$_\alpha$

A system with an initial configuration `cId` is a state transition system: states are triples $\{U, S, O\}$ consisting of the users (U), subjects (S), and objects (O) that exist at that moment in the system and are compatible with the specifications of `cId` and its configuration type (stored in `CfgType[cId]`); and transitions correspond to the operations from the functional specification of ABAC$_\alpha$. The state components U, S, O are assumed to be multisets because the entities of ABAC are identity-less: their behavior is uniquely determined by the values of their attributes. For this reason, different entities may assume the same term representation in our framework.

In this paper we take for granted the functional specification of ABAC$_\alpha$ given in [1]. It consists of six operations:

subject creation: user $u \in U$ can create a new subject s if formula ConstrS[u, s] holds. This formula belongs to the instance of the common policy language (CPL) that makes use of the attribute values of u and s.

subject deletion: $u \in U$ is free to delete any subject $s \in S$ it created before, that is, a subject s for which UId[u] = UId[s].

object creation: $s \in S$ can create a new object o if boolean formula ConstrO[s, o] holds. This formula belongs to the instance of CPL that uses the attribute values of entities s and o.

subject modification: $u \in U$ can change a subject $s \in S$ into s' if UId[u] = UId[s] = UId[s'] and ConstrModS[u, s, s'] holds. This formula belongs to the instance of CPL that uses the attribute values of u, s, and s'.

object modification: $s \in S$ can change $o \in O$ into o' if formula ConstrModO[s, o, o'] holds. This formula belongs to the instance of CPL that uses the attribute values of s, o, and o'.

authorized access: $s \in S$ can exercise a permission $p \in P$ on $o \in O$ if Auth[p, s, o] holds. P is a finite set of permission IDs (strings) and, for every $p \in P$, formula Auth[p, s, o] is from the instance of CPL that uses the attribute values of s and o.

Configuration-specific rules. CPL [3] is the common policy language part for the languages used to express the boolean formulas that constrain the operations of the functional specification of ABAC$_\alpha$. CPL describes a fragment of first-order logic where quantified formulas must be of the form $\exists x \in set.\varphi$ or $\forall x \in set.\varphi$ with set a finite set of values. The host language of ρLog is Mathematica, which is rich enough to express and decide the quantifier-free constraints of CPL. Moreover, we can eliminate all variable occurrences by repeated applications of the following reductions:

$$\exists x \in set.\varphi = \bigvee_{v \in set} \varphi[x \mapsto v] \qquad \forall x \in set.\varphi = \bigwedge_{v \in set} \varphi[x \mapsto v]$$

Every configuration type typeId has its own formulas that constrain the functionality of ABAC$_\alpha$. These formulas are expressed in instances of CPL. Therefore, we can write rule-based definitions of these conditions, which are specific to every typeId:

```
ConstrS[•U[...],•S[...]] →_{typeId} True/;φ_1.
ConstrO[•S[...],•O[...]] →_{typeId} True/;φ_2.
ConstrModS[•U[...],•S[...],•S[...]] →_{typeId} True/;φ_3.
ConstrModO[•S[...],•O[...],•O[...]] →_{typeId} True/;φ_4.
Auth[p_1,•S[...],•O[...]] →_{typeId} True/;φ_{5,1}.
...
Auth[p_r,•S[...],•O[...]] →_{typeId} True/;φ_{5,r}.
```

where $\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_{5,1}, \ldots, \varphi_{5,r}$ are formulas expressed in CPL.

Auxiliary functions and strategies. Subject and object creation are nondeterministic operations: they can be implemented by guessing the entity, and then creating it if the condition described by the corresponding CPL-formula holds. This process can be implemented in two steps. First, we define the functions sSeed[cId] which yields

$\bullet$S[sAt_1[SCOPE(sAt_1), τ_1], ..., sAt_n[SCOPE(sAt_n), τ_n]]

and oSeed[cId] which yields the term

$\bullet$O[oAt_1[SCOPE(oAt_1), τ_1], ..., oAt_n[SCOPE(oAt_p), τ_p]]

For example, sSeed["Cfg-01"] yields

•S["id"[{"u1","u2","u3"},"elem"]]

and oSeed["Cfg-01"] yields the term

•O["id"[S,"elem"],"r"[S,"subset"],"w"[S,"subset"]]

where S is {"u1","u2","u3"}.

Next, we use these terms as seeds from which we can produce any subject (resp. object) that may be created. In rule-based thinking, an entity $E[at_1[v_1], \ldots, at_k[v_k]]$ can be produced from the seed term $E[at_1[scope_1, \tau_1], \ldots, at_k[scope_k, \tau_k]]$ if and only if the reducibility formulas $scope_i \rightarrow_{\tau_i} v_i$ hold. Incidentally, the attribute type identifiers $\tau_i \in$ {"elem", "subset"} are also built-in atomic strategies of ρLog which guarantee the correctness of this rule-based thinking. Its implementation in ρLog is straightforward: If we define the strategy "setAt" with

```
att_[scope_,type_]→_{"setAt"}att[v]/;(scope→_{type}v_).
```

then the set of entities that can be generated from a seed term st is the set of all e for which reducibility formula $st \rightarrow_{\text{"fmap"["setAt"]}} e$ holds. Continuing this line of reasoning, we can argue that:

- user u can create new subject s iff u $\rightarrow_{\text{"createS"[cId]}}$ s holds, where this parametric strategy is defined by the rule

  ```
  u_→_{"createS"[cId]}s/;(sSeed[cId]→_{"fmap"["setAt"]}s_)∧
          (UId[u]≡UId[s])∧(ConstrS[u,s]→_{CfgType[cId]}True).
  ```

- subject s can create object o iff s $\rightarrow_{\text{"createO"[cId]}}$ o holds, where this parametric strategy is defined by the rule

  ```
  s_→_{"createO"[cId]}o/;(oSeed[cId]→_{"fmap"["setAt"]}o_)∧
                              (ConstrO[s,o]→_{CfgType[cId]}True).
  ```

- subject s can modify its attribute values and become sN iff its user creator u is in set U of existing users and is capable to change s to sN. We express this as the validity of the reducibility formula {U, s} $\rightarrow_{\text{"modSA"[cId]}}$ sN, where this strategy is defined by the rule

  ```
  {U_,s_}→_{"modSA"[cId]}sN/;
    (User[cId,UId[s]]≡u_)∧MemberQ[U,u]∧
    (sSeed[cId]→_{"fmap"["setAt"]}sN_)∧(UId[s]≡UId[sN])∧
    (ConstrModS[u,s,sN]→_{CfgType[cId]}True).
  ```

  If the creator of s is an existing user, then the admissible changes of s into sN are defined by the rule

  ```
  s_→_{"modSA"[cId]}sN/;(sSeed[cId]→_{"fmap"["setAt"]}sN_)∧
                (UId[s]≡UId[sN])∧(User[cId,UId[s]]≡u_)∧
                (ConstrModS[u,s,sN]→_{CfgType[cId]}True).
  ```

- subject s can change the attributes of object o to become object oN iff {s, o} $\rightarrow_{\text{"modOA"[cId]}}$ oN holds, where

  ```
  {s_,o_}→_{"modOA"[cId]}oN/;(oSeed[cId]→_{"fmap"["setAt"]}oN_)
                   ∧(ConstrModO[s,o,oN]→_{CfgType[cId]}True).
  ```

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

```
{{a___,u_,b___},S_,O_}→_{"createSubj"[cId]}
       {{a,u,b},Prepend[S,s],O}/;(u→_{"createS"[cId]}s_)∧
                                         Not[MemberQ[S,s]].
{{a___,u_,b___},{x___,s_,y___},O_}→_{"deleteSubj"[cId]}
                    {{a,u,b},{x,y},O}/;(UId[u]≡UId[s]).
{U_,{a___,s_,b___},O_}→_{"createObj"[cId]}
        {U,{a,s,b},Prepend[O,o]}/;(s→_{"createO"[cId]} o_)∧
                                          Not[MemberQ[O,o]].
{U_,{a___,s_,b___},O_}→_{"modifySubj"[cId]}
                  {U,{a,sN,b},O}/;({U,s}→_{"modSA"[cId]}sN_).
{U_,{a___,s_,b___},{x___,o_,y___}}→_{"modifyObj"[cId]}
           {U,{a,s,b},{x,oN,y}}/;({s,o}→_{"modOA"[cId]}oN).
```

In the state transitions defined by these rules, the entity matched by s_ is the selected subject, and the one matched by o_ is the selected object. Subjects may be created, and selected subjects are deleted or modified. Objects may be created, and selected objects are modified. To keep track of the attribute values of a subject or object in a particular state, we define its descendant with respect to a sequence Π of state transitions.

Suppose $\{U, S, O\}$ is a state for a configuration identified by cId, and $s \in S, o \in O$. Also, let $\pi : \{U, S, O\} \rightarrow_{stgId[\text{cId}]} \{U, S', O'\}$ be a state transition step. The descendants $desc_\pi(s)$ and $desc_\pi(o)$ of s and o with respect to π are defined as follows:

- $desc_\pi(s) := \bot$ if $\pi : \{U, S, O\} \rightarrow_{\text{"deleteSubj"[cId]}} \{U, S', O\}$ and the selected subject from S is s
- $desc_\pi(s) := s'$ if $\pi : \{U, S, O\} \rightarrow_{\text{"modSA"[cId]}} \{U, (S - \{s\}) \cup \{s'\}, O\}$
- $desc_\pi(o) := o'$ if $\pi : \{U, S, O\} \rightarrow_{\text{"modOA"[cId]}} \{U, S, (O - \{o\}) \cup \{o'\}\}$
- In the unspecified situations we have $desc_\pi(e) := e$.

Let $\Pi : \{U, S, O\} \rightarrow^* \{U, S', O'\}$ be a sequence of $n \geq 0$ transition steps and $s \in S$, $o \in O$. If $n = 0$ then $desc_\Pi(s) := s$ and $desc_\Pi(o) := o$. Otherwise, let π be the first transition step of Π, and Π′ the sequence of remaining transition steps. In this case $desc_\Pi(o) := desc_{\Pi'}(desc_\pi(o))$ and

$$desc_\Pi(s) := \begin{cases} \bot & \text{if } desc_\pi(s) = \bot, \\ desc_{\Pi'}(desc_\pi(s)) & \text{otherwise.} \end{cases}$$

More generally, we define the set of descendants of $e \in S \cup O$ from a state $\Sigma := \{U, S, O\}$ as follows:

$$Desc^{\Sigma}(e) := \{e' \mid e' \neq \bot \text{ and } e' = desc_{\Pi}(e) \text{ for some sequence } \Pi \text{ of state transitions starting with } \Sigma\}.$$

5 SAFETY OF ABAC$_\alpha$

The safety problem for configurations of ABAC$_\alpha$ is:

Given a state $\{U, S, O\}$ for an ABAC$_\alpha$ configuration identified by `cId`, entities $s \in S$, $o \in O$, and a permission $p \in P$,

Decide if there exists a sequence of state transitions $\Pi : \{U, S, O\} \rightarrow_{stg_1} \{U, S_1, O_1\} \ldots \rightarrow_{stg_n} \{U, S_n, O_n\}$ abbreviated $\Pi{:}\{U, S, O\} \rightarrow^* \{U, S_n, O_n\}$, such that $desc_{\Pi}(s) \neq \bot$ and $\texttt{Auth}[p, desc_{\Pi}(s), desc_{\Pi}(o)] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds.

We start by pointing out a few properties of ABAC$_\alpha$ that allow us to make some simplifying assumptions.

The five kinds of transitions are influenced only by the selected entities, at most two, and affect only one entity; the other entities of the system state are not affected. In particular:

(1) Objects can only participate at changing their own attributes. Since the presence of objects from $O - \{o\}$ plays no role in deciding whether $\texttt{Auth}[p, desc_{\Pi}(s), desc_{\Pi}(o)]$ holds for some Π, it is harmless to assume that the initial state is $\{U, S, \{o\}\})$ and Π has no object creation steps.

(2) If $\{U, S, O\} \rightarrow_{stg} \{U, S', O'\}$ then $\{U, S \cup S'', O'\} \rightarrow_{stg} \{U, S \cup S'', O'\}$ holds too, because we can choose the same participating entities to perform the transition. Therefore, it is harmless to assume that Π has no subject deletion steps.

Thus, it is harmless to assume that (1) Π has no transition steps of the strategies `"deleteSubj"[cId]` and `"createObj"[cId]`, and (2) the given state is of the form $\{U, S, \{o\}\}$.

From now on we assume that $\Sigma := \{U, S, \{o\}\}$ is a state for an ABAC$_\alpha$ configuration identified by `cId`, and that $s \in S$ and $p \in P$.

Lemma 1. *Let $s' \in Desc^{\Sigma}(s)$ and $o' \in Desc^{\Sigma}(o)$. There exists a derivation $\Pi : \Sigma \rightarrow^* \{U, S', \{o'\}\}$ such that $s' = desc_{\Pi}(s)$.*

Proof. Let u be the creator of s. $o' \in Desc^{\Sigma}(o)$ implies the existence of $\Pi_1 : \Sigma \rightarrow^* \{U, S'', \{o'\}\}$. If $s' \in S''$ then the lemma holds for $\Pi = \Pi_1$. Note that, if $u \notin U$, then s can not change its attribute values, thus $Desc^{\Sigma}(s) = \{s\} \subseteq S''$ and the lemma holds for $\Pi = \Pi_1$.

The only case left to analyse is when $u \in U$ and $s' \notin S''$. In this case, there exists a derivation $u \rightarrow_{\texttt{"createS"[cId]}} s_0 \rightarrow^*_{\texttt{"modSA"[cId]}} s$. $s' \in Desc^{\Sigma}(s)$ implies the existence of a derivation $s \rightarrow^*_{\texttt{"modSA"[cId]}} s'$. By concatenating these two derivations we obtain

$$u \rightarrow_{\texttt{"createS"[cId]}} s_0 \rightarrow^*_{\texttt{"modSA"[cId]}} s'.$$

Therefore, if $s_0 \notin S''$ then Π_1 can be extended as follows:

$$\Pi : \Sigma \rightarrow^* \{U, S'', \{o'\}\} \rightarrow_{\texttt{"createSubj"[cId]}} \{U, S'' \uplus \{s_0\}, \{o'\}\} \rightarrow^*_{\texttt{"modifySubj"[cId]}} \{U, S'' \uplus \{s'\}, O'\}.$$

Otherwise, $S'' = \{s_0\} \uplus S'''$ and Π_1 can be extended as follows:

$$\Pi : \Sigma \rightarrow^* \{U, \{s_0\} \uplus S''', \{o'\}\} \rightarrow^*_{\texttt{"modifySubj"[cId]}} \{U, \{s'\} \uplus S''', \{o'\}\}.$$

□

Theorem 1. *There is a derivation $\Pi : \Sigma \rightarrow^* \{U, S', \{o'\}\}$ such that $desc_{\Pi}(s) \neq \bot$ and $\texttt{Auth}[p, desc_{\Pi}(s), o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds, if and only if $\texttt{Auth}[p, s', o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds for some $s' \in Desc^{\Sigma}(s)$ and $o' \in Desc^{\Sigma}(o)$.*

Proof. If Π exists then $o' \in Desc^{\Sigma}(o)$ and we can choose $s' := desc_{\Pi}(s) \in Desc^{\Sigma}(s)$ such that $\texttt{Auth}[p, s', o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds.

Conversely, if $\texttt{Auth}[p, s', o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds for some $s' \in Desc^{\Sigma}(s)$ and $o' \in Desc^{\Sigma}(o)$ then, according to Lemma 1, there exists a derivation $\Pi : \Sigma \rightarrow^* \{U, S', \{o'\}\}$ such that $s' = descs_{\Pi}(s)$. Therefore, $\texttt{Auth}[p, desc_{\Pi}(s), o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds too. □

According to Theorem 1, we can proceed in two steps:

(1) look at all possible descendants s' of s; decide `UNSAFE` if any of them gets permission p on o, otherwise compute $Desc^{\Sigma}(s)$,

(2) look at all possible descendants o' of o; decide `UNSAFE` if any $s' \in Desc^{\Sigma}(s)$ can exercise permission p on some o', otherwise decide `SAFE`.

It remains to figure out how to compute $Desc^{\Sigma}(s)$ and $Desc^{\Sigma}(o)$, and how to check if $\texttt{Auth}[p, s', o'] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds for some $s' \in Desc^{\Sigma}(s)$ and $o' \in Desc^{\Sigma}(o)$. For this purpose, we define three strategies:

```
{{___,s_,___},{___,o_,___}}→"auth?"[p_,ct_] True/;
                                 (Auth[p,s,o] →ct True).
{___,s_,___}→"newModS"[c_,U_,S_] sN/;
              ({U,s} →"modSA"[c] sN)∧Not[MemberQ[S,sN]].
{{___,s_,___},{___,o_,___}}→"newModO"[c_,O_] oN/;
              ({s,o} →"modOA"[c] oN_)∧Not[MemberQ[O,oN]].
```

If S', S'' are sets of subjects and O', O'' are sets of objects, then:
1) $\texttt{ApplyRule["auth?"[}p\texttt{,CfgType[cId]],}\{S', O'\}\texttt{]}$ returns `True` iff some $s' \in S'$ can exercise permission p on some $o' \in O'$.
2) $\texttt{ApplyRuleList["newModS"[cId,}U, S''\texttt{],}S'\texttt{]}$ returns all descendants of elements from S' which are not in S''.
3) $\texttt{ApplyRuleList["newModO"[cId,}O''\texttt{],}\{S', O'\}\texttt{]}$ returns all descendants of elements from O' produced by some subject from S', which are not in O''.

Step 1. In this step we accumulate the value of $Desc^{\Sigma}(s)$ in a variable `sDESC` while checking if some $s' \in \texttt{sDESC}$ has the authority to exercise permission p on o. If the creator of s is not a user in the given configuration then the only descendant of s is s, and we only have to check if $\texttt{Auth}[p, s, o] \rightarrow_{\texttt{CfgType[cId]}} \texttt{True}$ holds or not. If yes, we decide `UNSAFE`, otherwise we set $\texttt{sDESC} := \{s\}$.

Otherwise, the creator of s is in U and $Desc^{\Sigma}(s) = \bigcup_{k=0}^{\infty} S_k$ where $S_1 := \{s\}$ and

$$\begin{aligned} S_{k+1} &:= \{s'' \notin \textstyle\bigcup_{i=1}^{k} S_i \mid \exists s' \in S_k . s' \rightarrow_{\texttt{"modSA"[cId]}} s''\} \\ &= \texttt{ApplyRuleList["newModS"[cId,}U, \textstyle\bigcup_{i=1}^{k} S_i\texttt{]}, S_k\texttt{]} \end{aligned}$$

Based on this observation, we can interleave the incremental accumulation in `sDESC` of the elements of S_k with the detection of unsafety when $\texttt{Auth}[p, s', o]$ holds for some $s' \in S_k$:

```
ct := CfgType[cId];
if ApplyRule["auth?"[$p$,ct],{{$s$},{$o$}}]==True
  return UNSAFE;
sDESC := {$s$}; sN := $\emptyset$;
if User[cId,UId[$s$]]$\in U$
 sN := ApplyRuleList["newModS"[cId,$U$,sDESC],{$s$}];
while sN $\neq \emptyset$ do
   (* test if Auth[$p, s', o$] holds for some $s' \in$ sN *)
   if ApplyRule["auth?"[$p$,ct],{sN,{$o$}}]==True
     return UNSAFE;
   sDESC := sDESC $\cup$ sN;
```

```
    sN := ApplyRuleList["newModS"[cId,$U$,sDESC],sN];
```

Note that, before entering the k-th loop of **while**, the values of sDESC and sN are $\bigcup_{i=1}^{k} S_i$ and S_{k+1}. The **while** loop will terminate because sDESC is finite, therefore $S_n = \emptyset$ for some $n \in \mathbb{N}$.

Step 2. In this step we accumulate the descendants of o in a variable oDESC and report UNSAFE as soon as we detect that formula $\{\mathsf{sDESC},\mathsf{oDESC}\} \to_{\text{"auth?"}[p,\mathsf{CfgType[cId]}]} \mathsf{True}$ holds.

First, we must figure out how to compute oDESC. The only subjects which may change the attribute values of (descendants) of o from Σ are those from $\left\{s'' \mid \exists s' \in S \cup \mathsf{sNew}.s' \to^{*}_{\text{"modSA"[cId]}} s''\right\}$ where sNew is the set of subjects that may be created by users from U. sALL can be computed incrementally as follows:

```
sNew := $\bigcup_{u \in U}$ ApplyRuleList[$u$, "createS"[cId]];
sALL := $\emptyset$; sN := $S \cup$ sNew;
while sN $\neq \emptyset$ do
    sALL := sALL $\cup$ sN;
    sN := ApplyRuleList["newModS"[cId,$U$,sALL],sN];
```

In every loop of **while** we compute the elements of a nonempty subset $\mathsf{sN} \subseteq \bigcup_{s \in S \cup \mathsf{sNew}} Desc^{\Sigma}(s)$ which are not yet accumulated in sALL, and accumulate them in sALL. The loop will terminate because the set $\bigcup_{s \in S \cup \mathsf{sNew}} Desc^{\Sigma}(s)$ is finite.

It is easy to see that $Desc^{\Sigma}(o)$ coincides with $\bigcup_{k=1}^{\infty} O_k$ where $O_0 := \{o\}$ and

$$\begin{aligned} O_{k+1} &:= \{o'' \notin \textstyle\bigcup_{i=1}^{k} O_i \mid \exists s' \in \mathsf{sALL}.\exists o' \in O_k. \\ &\qquad\qquad \{s', o'\} \to_{\text{"modOA"[cId]}} o''\} \\ &= \mathsf{ApplyRuleList}[\text{"newModO"}[\mathsf{cId}, \textstyle\bigcup_{i=1}^{k} O_i], \{\mathsf{sALL}, \mathsf{O_k}\}] \end{aligned}$$

Based on this observation, we can iterate the computation of O_k together with the test if $\mathsf{Auth}[p, s', o']$ holds for some $s' \in \mathsf{sDESC}$ and $o' \in O_k$. This iterative process stops when we reach a k with $O_k = \emptyset$, and this will eventually happen because $Desc^{\Sigma}(o)$ is finite.

```
oDesc := $\emptyset$; oN := {$o$};
while oN $\neq \emptyset$ do
    if ApplyRule["auth?"[$p$,ct],{sDESC,oN}]==True
        return UNSAFE;
    oDESC := oDESC $\cup$ oN;
    oN := ApplyRuleList["newModO"[cId,oDESC],{sALL,oN}];
```

The final algorithm. By putting together the rule-based algorithms for steps 1 and 2, we obtain the algorithm illustrated in Figure 1.

```
ct := CfgType[cId];
$U$ := Users[cId]; $S$ := Subjects[cId];
if ApplyRule["auth?"[$p$,ct],{{$s$},{$o$}}]==True
    return UNSAFE;
(* start accumulating the elements of $Desc^{\Sigma}(s)$ in sDESC *)
sDESC := {$s$}; sN := $\emptyset$;
if User[cId,UId[$s$]]$\in U$
  sN:=ApplyRuleList["newModS"[cId,$U$,sDESC],{$s$}];
while sN $\neq \emptyset$ do
    if ApplyRule["auth?"[$p$,ct],{sN,{$o$}}]==True
        return UNSAFE;
    sDESC := sDESC $\cup$ sN;
    sN := ApplyRuleList["newModS"[cId,$U$,sDESC],sN];
sNew := $\bigcup_{u \in U}$ ApplyRuleList[$u$, "createS"[cId]];
sALL := $\emptyset$;
(* accumulate in sALL the elements of
  {$s'' \mid \exists s' \in S \cup$ sNew.$s' \to^{*}_{\text{"modSA"[cId]}} s''$} *)
sN := $S \cup$ sNew;
while sN $\neq \emptyset$ do
    sALL := sALL $\cup$ sN;
    sN := ApplyRuleList["newModS"[cId,$U$,sALL],sN];
oDESC := $\emptyset$; oN := {$o$};
while oN $\neq \emptyset$ do
    if ApplyRule["auth?"[$p$,ct],{sDESC,oN}]==True
        return UNSAFE;
    oDESC := oDESC $\cup$ oN;
    oN := ApplyRuleList["newModO"[cId,oDESC],{sALL,oN}];
return SAFE;
```

Figure 1: The safety check algorithm for ABAC$_\alpha$.

6 CONCLUSIONS

We proved that the safety problem of ABAC$_\alpha$ can be reduced to checking if $\mathsf{Auth}[p, s', o']$ holds for some $s' \in Desc^{\Sigma}(s)$ and $o' \in Desc^{\Sigma}(o)$, and solved it by identifying a rule-based algorithm that interleaves the detection of unsafety with the incremental computation of $Desc^{\Sigma}(s)$ and $Desc^{\Sigma}(o)$.

Our algorithm is parametric with respect to the configurations of ABAC$_\alpha$. Therefore, whenever we want to check that, for a given configuration, a subject s never gets authorization to exercise permission p on an object o, it is enough to do the following: (1) specify the configuration and its type, and (2) call the method

```
CheckSafety[cId,$s$,$o$,$p$]
```

which runs our safety check algorithm. It returns SAFE if s never gets authorization to exercise permission p on o, and UNSAFE otherwise.

7 ACKNOWLEDGMENTS

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

REFERENCES

[1] T. Ahmed and R. Sandhu. 2017. Safety of ABAC$_\alpha$ is Decidable. In *Network and System Security*, Z. Yan, R. Molva, W. Mazurczyk, and R. Kantola (Eds.). Springer International Publishing, 257–272.

[2] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. 1976. Protection in operating systems. *Commun. ACM* 19, 8 (Aug. 1976), 461–471.

[3] X. Jin. 2014. *Attribute-Based Access Control Models and Implementation in Cloud Infrastructure as a Service.* Ph.D. Dissertation. University of Texas at San Antonio.

[4] X. Jin, R. Krishnan, and R. Sandhu. 2012. A unified attribute-based access control model covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI*, N. Cuppens-Boulahia, F. Cuppens, and J. Garcia-Alfaro (Eds.). LNCS, Vol. 7371. Springer, Berlin, Heidelberg, 41–55.

[5] T. Kutsia. 2004. Solving equations involving sequence variables and sequence functions. In *Proceedings of AISC 2004*, B. Buchberger and J. A. Campbell (Eds.). LNCS, Vol. 3249. Springer, 157–170.

[6] T. Kutsia. 2007. Solving equations with sequence variables and sequence functions. *JSC* 42, 3 (2007), 352–388.

[7] T. Kutsia and M. Marin. 2005. Can context sequence matching be used for querying XML?. In *Proceedings of UNIF'05*, L. Vigneron (Ed.). IEEE Computer Society, Nara, Japan, 77–92.

[8] M. Marin and T. Ida. 2005. Rule-Based Programming with ρLog. In *Proceedings of SYNASC'05*, D. Zaharie, D. Petcu, V. Negru, T. Jebelean, G. Ciobanu, A. Cicortas, A. Abraham, and M. Paprzycki (Eds.). IEEE Computer Society, 31–38.

[9] M. Marin and T. Kutsia. 2006. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics* 16, 1-2 (2006), 151–168.

[10] M. Marin and F. Piroi. 2004. Deduction and presentation in ρLog. *ENTCS* 93 (2004), 161–182.

[11] J. Park and R. Sandhu. 2004. The UCON$_{\text{ABC}}$ Usage Control Model. *ACM Transactions on Information and System Security (TISSEC)* 7, 1 (2004), 161–182.

[12] P. V. Rajkumar and R. Sandhu. 2016. Safety decidability for pre-authorization usage control with finite attribute domains. *IEEE Transactions on Dependable and Secure Computing* 13, 5 (2016), 582–590.

[13] S. Wolfram. 2003. *The Mathematica Book* (5th ed.). Wolfram Media.