Static Detection of Collusion Attacks in ARBAC-based Workflow Systems

Stefano Calzavara  Alvise Rabitti  Enrico Steffinlongo  Michele Bugliesi
Università Ca’ Foscari Venezia

Abstract—Authorization in workflow systems is usually built on top of role-based access control (RBAC); security policies on workflows are then expressed as constraints on the users performing a set of tasks and the roles assigned to them. Unfortunately, when role administration is distributed and potentially untrusted users contribute to the role assignment process, like in the case of Administrative RBAC (ARBAC), collusions may take place to circumvent the intended workflow security policies. In a collusion attack, a set of users of a workflow system collaborates by changing the user-to-role assignment, so as to sidestep the security policies and run up to completion a workflow they could not complete otherwise.

In this paper, we study the problem of collusion attacks in a formal model of workflows based on stable event structures and we define a precise notion of security against collusion. We then propose a static analysis technique based on a reduction to a role reachability problem for ARBAC, which can be used to prove or disprove security for a large class of workflow systems. We also discuss how to aggressively optimise the obtained role reachability problem to ensure its tractability. Finally, we implement our analysis in a tool, WARBAC, and we experimentally show its effectiveness on a set of publicly available examples, including a realistic case study.

I. INTRODUCTION

A workflow is a temporally organised collection of tasks, representing a business process specification. Workflow systems are software supporting a specific set of business processes through the execution of computerized task definitions. These software not only ensure that the execution of the tasks in a workflow respects the expected temporal order, but also that these tasks are performed by authorized users.

Authorization in workflow systems is usually built on top of role-based access control (RBAC). RBAC is a very natural choice for workflow systems, since roles provide a convenient abstraction to represent a (possibly large) set of users entitled to perform a given task [4], [7]. When role-based security policies on task execution are not expressible enough, security policies on workflows can also include constraints on the identity of the users performing a set of tasks, like binding-of-duty and separation-of-duty constraints [8], [19]. Binding-of-duty (BoD) constraints enforce two different tasks to be performed by the same user, e.g., to prevent an undesired disclosure of sensitive information or to ensure a single user takes full responsibility for a set of related tasks. Separation-of-duty (SoD) constraints, instead, play the dual role of ensuring that two different tasks are performed by two different users, e.g., to prevent frauds or conflicts of interest.

An important observation for security is that, though both role-based and identity-based security policies like BoD and SoD constraints are static and declarative in nature, the set of roles assigned to the users of a workflow system is typically not. For instance, the Administrative RBAC (ARBAC) standard [12] allows system administrators to specify which roles are entitled to assign other roles to users, based on the sets of roles assigned (or not assigned) to them; similarly, roles may be granted the ability of revoking other roles from system users. Role administration is thus highly distributed in ARBAC systems, which is a very desirable feature for normal system functionality; however, it is also well-known that such a feature poses an important security challenge, since the sets of roles which may be dynamically assigned to users is very hard to predict without automated tool support [17], [20].

In the case of workflow systems based on the ARBAC model, the fact that potentially untrusted users contribute to the role assignment process enables hard-to-spot collusions aimed at circumventing the intended workflow security policies. Specifically, in a collusion attack a set of users of a workflow system collaborates by changing the user-to-role assignment, so as to sidestep the security policies put in place by the system administrators and run up to completion a workflow they could not complete otherwise.

A. Motivating Example

We graphically represent workflows as directed graphs, including one node per task, a start node \( ✓ \) and an end node \( ✓ \). We use directed arrows to represent temporal dependencies, dashed lines labelled with \# to visualize exclusive choices and dashed lines labelled with either \( = \) and \( \neq \) to represent BoD and SoD constraints respectively. We annotate each task with a subscript including the set of roles entitled to perform it.

Consider now two users \( u_1 \) and \( u_2 \) who are assigned roles \( R_1, R_2 \) respectively. These users cannot complete the workflow just with their roles, since they lack the privileges needed to perform \( a, b \) or \( a, c \) without violating the BoD constraints. However, assume that any user who is assigned role \( R_2 \) is also allowed to assign role \( R_3 \) to any user owning role \( R_1 \): this kind of policies is common in access control systems supporting role administration, including the standard ARBAC model [12]. Under this policy, users \( u_1 \) and \( u_2 \) can collude to...
complete the workflow as follows: user \( u_2 \) assigns role \( R_3 \) to user \( u_1 \), who thus becomes able to perform tasks \( a \) and \( c \) on her own. Though such a possibility may be easy to spot in this simple example, detecting collusions is hard in general, since they may be enabled by arbitrarily long sequences of actions and the underlying ARBAC policy may include hundreds of rules for role administration like the one we discussed.

**B. Contributions**

In this paper, we make the following contributions:

1) we propose a formal model of workflows based on stable event structures \([24]\), which generalizes previous proposals based on partial orders \([21]\), \([22]\). We then integrate this model with ARBAC, by defining a small-step operational semantics for workflow systems where workflow actions (execution of a task) and administrative actions (assignment or revocation of roles) are arbitrarily interleaved;

2) we define a precise notion of security against collusion for workflow systems, ensuring that administrative actions cannot be abused to sidestep the workflow security policy. The definition is adapted from previous work on the security of delegation in access control systems \([23]\);

3) we propose a static analysis based on a reduction to a role reachability problem for ARBAC, which can be used to prove or disprove security against collusion for restricted yet useful classes of workflow systems. By reducing security to role reachability, it is possible to reuse available tools for role reachability analysis \([13]\), \([9]\), \([16]\) to effectively check it. We aggressively optimise the role reachability problem to ensure its tractability;

4) we implement our static analysis in a tool, WARBAC, and we experimentally show its effectiveness on a set of publicly available examples, including a realistic case study describing a first-aid procedure.

We make WARBAC, all the experimental data and an extended version of the present paper (with proofs) available online \([10]\).

**Structure of the Paper:** Section II presents the operational model. Section III introduces the formal definition of security and gives an example. Section IV details the reduction to role reachability and its optimization. Section V shows a few example reductions. Section VI presents WARBAC and reports on the experimental results. Section VII discusses related work. Section VIII concludes and hints at future work.
Besides being an expressive model on their own, stable event structures have also been proposed to define the semantics of several formalisms commonly used to model workflows, including CCS, CSP and Petri nets (see [24] for CCS/CSP and [5] for Petri nets). By working with stable event structures, we make our theory general enough to be directly applicable to any workflow specification language whose semantics can be defined in terms of these structures.

There are several slightly different definitions of stable event structure in the literature, the one we use here is taken from [18]: it is more compact than the original formulation in [24], but semantically equivalent to it. We just omit the labelling function from the definition, since it does not play any role in the present paper.

**Definition 3 (Stable Event Structure).** A stable event structure is a triple $\mathcal{E} = \langle E, \#, \vdash \rangle$, where:

1) $E$ is a denumerable set of events;
2) $\# \subseteq E \times E$ is a symmetric, irreflexive conflict relation;
3) $\vdash \subseteq 2^E \times E$ is an enabling relation;
4) for all events $e \in E$ and all sets of events $X, Y \subseteq E$ such that $X \neq Y$, the following stability axiom is satisfied:

$$X \vdash e \land Y \vdash e \Rightarrow \exists e_1, e_2 \in X \cup Y : (e_1, e_2) \in \#.$$ 

If $X \vdash e$ for some set of events $X \subseteq E$, then the occurrence of all the events in $X$ enables the occurrence of the event $e$; if $(e_1, e_2) \in \#$, instead, then the occurrence of event $e_1$ rules out the occurrence of event $e_2$ and vice-versa. The stability axiom requires that, if there are different enablings for the same event, they are conflicting, which ensures that each event is enabled in an essentially unique way.

The semantics of stable event structures is defined in terms of a set of configurations, defined as follows [24].

**Definition 4 (Configuration).** Given a stable event structure $\mathcal{E} = \langle E, \#, \vdash \rangle$, a configuration of $\mathcal{E}$ is a finite set of events $X \subseteq E$ such that:

1) $\forall e, e' \in X : (e, e') \not\in \#$;
2) $\forall e \in X. \exists e_1, \ldots, e_n \in X : e_n = e \land \forall i \leq n. \exists Y \subseteq \{e_1, \ldots, e_{i-1}\} : Y \vdash e_i$.

We let $\mathcal{F}(\mathcal{E})$ be the set of all the possible configurations of $\mathcal{E}$.

The first condition ensures that a configuration does not contain conflicting events, while the second condition says that for each event $e$ in a configuration there exists a sequence of events $e_1, \ldots, e_n = e$ again in the configuration such that each $e_i$ is enabled by a subset of $\{e_1, \ldots, e_{i-1}\}$. Intuitively, it is thus possible to build a chain of enablings that enables $e$ starting from the empty set and each configuration of a stable event structure can be understood as a computation history up to a certain state.

In our model, we represent tasks in a workflow as events of a stable event structure and we use the terms “event” and “task” interchangeably in the paper, picking the most natural choice based on the context of the discussion. A workflow is a stable event structure including a special event $\emptyset$, representing completion, and extended with a set of constraints $C$ and a task-to-role assignment function $\rho$. The constraints $C$ allow one to specify that two tasks must be performed by the same user (BoD) or by two different users (SoD), while the function $\rho$ assigns to each task a role which is needed to perform it$^1$.

**Definition 5 (Workflow).** A workflow is a triple $\mathcal{W} = \langle \mathcal{E}, C, \rho \rangle$, where:

1) $\mathcal{E} = \langle E \cup \{\emptyset\}, \#, \vdash \rangle$ is a stable event structure including an event $\emptyset \not\in E$ such that, whenever $X \vdash e$ for some $X$ and $e$, we have $\emptyset \not\in X$;
2) $C \subseteq E \times E \times \{=, \neq\}$ is a relation defining a set of BoD and SoD constraints such that:
   a) $\forall e \in E : (e, e, \neq) \not\in C$;
   b) $\forall e_1, e_2 \in E : (e_1, e_2, =) \in C \Rightarrow (e_2, e_1, =) \in C$;
   c) $\forall e_1, e_2, e_3 \in E : (e_1, e_2, \neq) \in C \Rightarrow (e_2, e_1, \neq) \in C$;
   d) $\forall e_1, e_2, e_3 \in E : (e_1, e_2, =) \in C \land (e_2, e_3, =) \in C \Rightarrow (e_1, e_3, =) \in C$;
   e) $\forall e_1, e_2, e_3 \in E : (e_1, e_2, =) \in C \land (e_2, e_3, =) \not\in C \Rightarrow (e_1, e_3, \neq) \not\in C$;
3) $\rho : E \to R$ is a function from events to roles.

To improve readability, in our examples we do not explicitly close the set of constraints with respect to the rules above. Given a workflow $\mathcal{W} = \langle \mathcal{E}, \#, \vdash \rangle, C, \rho \rangle$, we use a subscript notation to extract its different components, e.g., we let $\vdash_{\mathcal{W}}$ stand for $\vdash$ and $C_{\mathcal{W}}$ stand for $C$.

**C. ARBAC + Workflows**

Having introduced the ARBAC model and a formal definition of workflow, we now study their interplay by giving a reduction semantics to ARBAC workflow systems.

**Definition 6 (ARBAC Workflow System).** An ARBAC workflow system is a pair $\mathcal{A} = \langle S, \mathcal{W} \rangle$ including an ARBAC system $S$ and a workflow $\mathcal{W}$.

Since workflows in our model allow the specification of BoD and SoD constraints, the reduction semantics of ARBAC workflow systems needs to keep track of the author of the individual tasks. This is formalized by introducing the following notion of history.

**Definition 7 (History).** Given a workflow $\mathcal{W}$, a history $H : E_{\mathcal{W}} \to U$ is a partial function from tasks to users such that dom$(H)$ is a configuration of $\mathcal{E}_{\mathcal{W}}$. We let $\bot$ stand for the empty history.

Given a history $H$, we can readily check whether the workflow constraints are satisfied or not. Clearly, a BoD/SoD constraint predicating on the authors of two different tasks can only be checked when the second one is attempted, which leads to the following definition.

$^1$This is expressive enough to represent tasks which require multiple roles to be performed or any role in a given set. For instance, if both $r_1$ and $r_2$ are needed for a task $e$, one can introduce in the ARBAC policy a fresh role $r$ which is only granted to users who are assigned both $r_1$ and $r_2$. Then, it is enough to let $\rho(e) = r$. 
Definition 8 (Satisfiability). Let \( \sim \in \{=, \neq\} \), we say that \( H \) satisfies the constraint \((e_1, e_2, \sim)\) whenever \( H \models (e_1, e_2, \sim) \) can be proved by the following rules:

\[
\begin{align*}
&\frac{e_1 \notin \text{dom}(H)}{H \models (e_1, e_2, \sim)} & e_2 \notin \text{dom}(H) & H(e_1) \sim H(e_2) & H \models (e_1, e_2, \sim)
\end{align*}
\]

Let \( H \models C \) whenever \( \forall(e_1, e_2, \sim) \in C : H \models (e_1, e_2, \sim) \).

The operational semantics of an ARBAC workflow system \( \mathcal{A} = (\mathcal{S}, \mathcal{W}) \) is defined by means of a labelled reduction relation on states \( \sigma = (UR, H) \), including a user-to-role assignment \( UR \) and a history \( H \) (see Table II). Labels are drawn from the set of events \( \mathcal{E}_W \) in the workflow, extended with a distinguished event \( o \) representing the occurrence of an administrative action (assignment or revocation of roles). To update the history upon reduction, we use the following notation: for a partial function \( f \) with \( x \notin \text{dom}(f) \), let \( f[x \mapsto y] \) be the partial function \( g \) such that \( \text{dom}(g) = \text{dom}(f) \cup \{x\} \), \( g(x) = y \) and \( \forall z \in \text{dom}(f) : g(z) = f(z) \).

### Table II Reduction semantics \((\mathcal{P}, \mathcal{W} \vdash \sigma \xrightarrow{c} \sigma')\)

<table>
<thead>
<tr>
<th>Rule (R-ADMIN)</th>
<th>( \mathcal{P} \vdash \mathcal{W} \rightsquigarrow (UR, H) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{P} \vdash \mathcal{W} \rightsquigarrow (UR, H) )</td>
<td>( \mathcal{P} \vdash \mathcal{W} \rightsquigarrow (UR', H) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule (R-TASK)</th>
<th>( \mathcal{P} \vdash \mathcal{W} \rightsquigarrow (UR, H) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists X \subseteq \text{dom}(H) : \mathcal{X} \mathcal{W} \exists e' \in \text{dom}(H) : (e', e) \notin # \mathcal{W} )</td>
<td>( \mathcal{P} \vdash \mathcal{W} \rightsquigarrow (UR, H) )</td>
</tr>
</tbody>
</table>

Rule (R-ADMIN) is straightforward: it allows one to change the user-to-role assignment in accordance with the underlying ARBAC policy. Rule (R-TASK), instead, models the execution of a task. In words, it is possible to execute a task if: (1) there exists a user who is granted the required role for the task; (2) the execution of the task by this user does not violate the BoD/SoD constraints; (3) the task is enabled by the already performed tasks; and (4) the task does not conflict with any of the already performed tasks.

Observe that the reduction relation in Table II is well-defined, since the premises of rule (R-TASK) ensure that only valid histories are introduced upon reduction when starting from the empty history \( \bot \).

### III. Security Against Collusion

#### A. Formal Definition of Security

Let \( U_c \subseteq U \) be a set of colluding users. Intuitively, an ARBAC workflow system is secure against collusion by \( U_c \) whenever no sequence of administrative actions performed by the users in \( U_c \) can allow them to complete a workflow which they could not complete just with their original roles. We now formalise this intuition, though we need a number of auxiliary definitions first.

Given a user-to-role assignment \( UR \) and a set of colluding users \( U_c \), we let \( UR \downarrow U_c = \{ (ur, r) \in UR \mid u \in U_c \} \) stand for the subset of \( UR \) including only users in \( U_c \). We extend the notation to ARBAC workflow systems in the expected way, by having \( \langle \langle \mathcal{P}, UR \rangle, \mathcal{W} \rangle \downarrow U_c = \langle \langle \mathcal{P}, UR \downarrow U_c \rangle, \mathcal{W} \rangle \).

Given an ARBAC workflow system \( \mathcal{A} = (\langle \mathcal{P}, UR \rangle, \mathcal{W}) \), a trace of \( \mathcal{A} \) is a sequence of events \( t = e_1, \ldots, e_n \) such that:

\[
\exists \sigma_0, \ldots, \sigma_n : \sigma_0 = (UR, \bot) \land \forall 0 < i \leq n : \mathcal{P}, \mathcal{W} \triangleright \sigma_{i-1} \xrightarrow{e_i} \sigma_i.
\]

A trace \( t \) is successful iff \( \triangleright \) occurs in \( t \). We let \( T^\triangleright (\mathcal{A}) \) denote the set of the successful traces of \( \mathcal{A} \). A trace \( t \) is pure iff it does not contain any administrative action, i.e., iff \( o \) does not occur in \( t \). We let \( P T^\triangleright (\mathcal{A}) \) stand for the set of the pure, successful traces of \( \mathcal{A} \).

Definition 9 (Security Against Collusion). An ARBAC workflow system \( \mathcal{A} \) is secure against collusion by \( U_c \) iff:

\[
T^\triangleright (\mathcal{A} \downarrow U_c) \neq \emptyset \Rightarrow PT^\triangleright (\mathcal{A} \downarrow U_c) \neq \emptyset.
\]

#### B. Example

We now encode in our formalism the motivating example in Section I-A (Figure 1) and we show that it is not secure against collusion by \( \{u_1, u_2\} \).

First, we define \( \mathcal{W} = \langle \{(a, b, c, \triangleright), \{a\}, C, \rho\rangle \), where:

1) the enabling relation is:

\[
t = \{ (\emptyset, a), (\{a\}, b), (\{a\}, c), (\{b\}, \triangleright), (\{c\}, \triangleright) \}
\]

2) the conflict relation is \# = \{ (b, c), (c, b) \};

3) \( C = \{ (a, b, =), (a, c, =) \}\) enforces BoD between tasks \( a, b \) and between tasks \( a, c \);

4) \( \rho(a) = R_1, \rho(b) = R_2, \rho(c) = R_3 \) requires role \( R_1 \) to perform task \( a \), role \( R_2 \) to perform task \( b \) and role \( R_3 \) to perform task \( c \).

We then build on top of this workflow the ARBAC workflow system \( \mathcal{A} = (\langle \mathcal{P}, UR \rangle, \mathcal{W}) \), where:

1) \( \mathcal{P} = \{ (R_2, \{R_1\}, \emptyset, R_3) \} \) is the ARBAC policy which allows users with role \( R_2 \) to assign role \( R_3 \) to any user who is granted role \( R_1 \);

2) \( UR = \{ (u_1, R_1), (u_2, R_2) \} \) contains two users \( u_1, u_2 \) with roles \( R_1, R_2 \) respectively.

We have that \( \mathcal{A} \) is not secure against collusion by \( \{u_1, u_2\} \) according to Definition 9, since it is possible to put the event \( \triangleright \) in the history by first assigning role \( R_3 \) to \( u_1 \) and then letting her execute both \( a \) and \( c \); however, without administrative actions (the role assignment by \( u_2 \)) it is not possible for the two users to complete the workflow.

#### C. Checking Security Against Collusion

By definition, given an ARBAC workflow system \( \mathcal{A} \), its security against collusion by a set of users \( U_c \subseteq U \) can be checked as follows:

1) check if there exists a successful trace \( t \in T^\triangleright (\mathcal{A} \downarrow U_c) \): if this is not the case, \( \mathcal{A} \) is secure;

2) otherwise, \( \mathcal{A} \) is secure if (and only if) there exists also a pure successful trace \( t' \in PT^\triangleright (\mathcal{A} \downarrow U_c) \).
Formally, point (2) amounts to checking the satisfiability (or consistency) of a workflow, a problem which has received considerable attention in the past [21], [22]. In particular, Wang and Li proved that the workflow satisfiability problem is NP-hard in presence of SoD constraints [22]. Based on this result, it is clear that also point (1) is at least NP-hard in the general case, since points (1) and (2) coincide on the ARBAC workflow system implementing an empty ARBAC policy.

To the best of our knowledge, no algorithm has been proposed so far to deal with point (1). Building all the possible user-to-role assignments for the colluding users based on the ARBAC policy and then checking workflow satisfiability with respect to them is not feasible in practice, given the exponential blow-up of the possible role combinations and the inherent complexity of workflow satisfiability itself. In the next section, we propose the first feasible static analysis technique to solve point (1).

IV. Static Analysis

A. Overview

We propose to check security against collusion in ARBAC workflow systems through a reduction to the well-known role reachability problem for ARBAC [17], [20].

Given an ARBAC system \( S = (\mathcal{P}, \mathcal{UR}) \), a role \( r \) is said reachable in \( S \) if and only if it can be assigned to some user of the system at some point in time. Formally, this means that there exist a user \( u \) and a sequence of user-to-role assignments \( UR_0, \ldots, UR_n \) such that \( UR_0 = UR \) and:

\[
(\forall i \leq n : \mathcal{P} \triangleright UR_{i-1} \rightsquigarrow UR_i) \land r \in UR_n(u).
\]

We propose to encode the workflow as a set of can-assign rules, which extend the original ARBAC policy \( \mathcal{P} \) so that a specific role (introduced by the encoding) is reachable if and only if the workflow can be completed by the colluding users, possibly by making use of administrative actions; this characterization is proved correct for a restricted, yet useful, class of workflow systems discussed below. Notably, however, even for workflow systems which do not belong to this class, we prove that the unreachability of the role above ensures that the workflow cannot be completed by the colluding users, which may be enough to prove security against collusion in many practical cases.

By internalizing the workflow into the underlying ARBAC policy, we can reuse efficient tools for role reachability analysis [13], [9], [16] to prove or disprove security against collusion in workflow systems. Moreover, having a unified representation (in terms of ARBAC) of both the ARBAC policy and the workflow to analyse makes it possible to devise aggressive optimizations which exploit as much information as possible to simplify the security problem and make it tractable.

B. Reduction to Role Reachability

Let \( \mathcal{A} = (\mathcal{S}, \mathcal{W}) \), for each task \( e \in E_W \) we introduce three fresh roles: Allowed\( [e] \), Author\( [e] \) and Done\( [e] \). Moreover, for each pair of tasks \( e_1, e_2 \) such that \( (e_1, e_2, =) \in C_W \), we introduce a fresh role Eq\( [e_1, e_2] \). Finally, we introduce a fresh role Super, which will always be assigned to a dummy user introduced by the encoding. Notice that we can always ensure that the set of roles in \( \mathcal{A} \) does not clash with the set of roles introduced by the encoding just by performing a preliminary renaming of the elements of the former.

The core of the encoding is a translation from a workflow \( W \) into a set of can-assign rules \( [W] \). The translation requires the generation of the set of the configurations of \( E_W \) and it assumes the following definition of precedence, which can be used to identify the set of the predecessors of an event in a stable event structure [18].

**Definition 10** (Precedence). For a stable event structure \( E = (E, \triangleright, \#) \) and a configuration \( X \in \mathbb{F}(E) \), we define the precedence relation \( \prec_X \subseteq X \times X \) by having \( e \prec_X e' \) if and only if \( \exists Y \subseteq X : e \in Y \land Y \triangleright e' \). We then let \( \prec_X \) stand for the transitive closure of \( \prec_X \).

We let \( [W] \) be the smallest set of can-assign rules derived by the inference rules in Table III. The core intuitions underlying the translation can be summarized as follows:

- we assume the existence of a dummy user with the Super role, which we call the super user. We use the super user to trigger several can-assign rules generated by the translation and to keep track in the encoding of which tasks can be executed or have been performed so far. This is done by assigning to the super user a specific set of roles (again introduced by the translation) and by ensuring that they satisfy the invariants explained below;
- rule (T-ALLOWED): to assign role Allowed\( [e] \), we must ensure that: (1) all the predecessors of \( e \) have already been performed; (2) none of the predecessors of \( e \) violates a BoD constraint; and (3) no event which is conflicting with \( e \) has been previously allowed. Notice that Allowed\( [e] \) can only be assigned to the super user: this ensures that all the information about the allowed events is centralized on the super user, i.e., in the encoding we can always be aware of all the tasks which have been allowed so far;
- rule (T-AUTHOR): once Allowed\( [e] \) has been assigned to the super user, it is possible to attempt the assignment of role Author\( [e] \). This role can only be assigned to a user who has the required role to perform \( e \) according to the task-to-role assignment function \( \rho \); moreover, the user must satisfy all the BoD constraints between \( e \) and its predecessors, as well as all the SoD constraints between \( e \) and the other tasks of the workflow. Observe that the super user can never be assigned Author\( [e] \), reflecting the intuition that he is just a dummy user introduced by the encoding and not a real user of the system;
- rule (T-DONE): once Author\( [e] \) has been assigned to some user, it is possible to assign role Done\( [e] \) to the super user. This role assignment tracks that it was indeed possible to perform task \( e \). Role Done\( [e] \) may be required to enable the assignment of role Allowed\( [e'] \) for some event \( e' \) which is a successor of \( e \);
- rule (T-EQ): if a user is assigned both Author\( [e_1] \) and Author\( [e_2] \), she can also be assigned role Eq\( [e_1, e_2] \), thus
Table III Translation of a workflow \( \mathcal{W} \) into a set of can-assign rules \( \llbracket \mathcal{W} \rrbracket \)

| (T-L) \( X \in F(\mathcal{E}_\mathcal{W}) \) \( e \in X \) \( \text{Done} = \{ \text{Done}[e'] | e' <_X e \} \) | (T-EQ) \( (e_1, e_2, =) \in C_{\mathcal{W}} \) \( \llbracket \text{Super} \rrbracket \cup \text{Eqs} \cup \{ \text{Super} \}, \{ \text{Allowed}[e'] | (e, e') \in \#_W \}, \{ \text{Allowed}[e] \} \in \llbracket \mathcal{W} \rrbracket \) |
| --- | --- |
| (T-A) \( X \in F(\mathcal{E}_\mathcal{W}) \) \( e \in X \) \( \text{Neqs} = \{ \text{Author}[e'] | (e', e, \neq) \in C_{\mathcal{W}} \} \) | (T-AP) \( (e_1, e_2, =) \in C_{\mathcal{W}} \) \( \{ \text{Eq}[e_1, e_2,], \{ \text{Super} \}, \{ \text{Eq}[e_1, e_2] \} \in \llbracket \mathcal{W} \rrbracket \) |
| (T-D) \( \text{Done} [e] \in E_\mathcal{W} \) | (T-D) \( (e_1, e_2, =) \in C_{\mathcal{W}} \) \( \{ \text{Author}[e], \{ \text{Super} \}, \emptyset, \{ \text{Done}[e] \} \in \llbracket \mathcal{W} \rrbracket \) |

proving that a BoD constraint between \( e_1 \) and \( e_2 \) has been satisfied. Afterwards, by rule (T-AP), role \( \text{Eq}[e_1, e_2] \) can be further assigned to the super user: this may be needed to enable the assignment of role \( \text{Allowed}[e] \) for some event \( e \) following both \( e_1 \) and \( e_2 \).

Finally, the translation is extended so as to map an ARBAC workflow system \( \mathcal{A} = \langle \langle \text{CA}, \text{CR}, \text{UR} \rangle, \mathcal{W} \rangle \) into a corresponding ARBAC system \( \llbracket \mathcal{A} \rrbracket \) as follows:

\[
\llbracket \mathcal{A} \rrbracket = \langle \langle \text{CA}^-, \llbracket \mathcal{W} \rrbracket, \text{CR} \rangle, \text{UR} \rangle \cup \{ (u_0, \text{Super}) \},
\]

where \( u_0 \) is a fresh user extending the set of users \( U \) and \( \text{CA}^- \) is the set of the can-assign rules obtained by including \( \text{Super} \) in the negative preconditions of all the rules in \( \text{CA} \).

C. Formal Results

The first result we prove for the encoding we detailed is the following soundness theorem.

**Theorem 1** (Soundness). For any \( \mathcal{A} \) such that \( \text{T}^\vee (\mathcal{A}) \neq \emptyset \), the role \( \text{Done}[\sqrt{\cdot}] \) is reachable in \( \llbracket \mathcal{A} \rrbracket \).

**Proof.** See the long version [10].

The soundness theorem ensures that, if role \( \text{Done}[\sqrt{\cdot}] \) is not reachable in \( \llbracket \mathcal{A} \rrbracket \), then the ARBAC workflow system \( \mathcal{A} \) will never reach a state where the workflow has been completed by the system users, even by making use of administrative actions. In terms of security this means that, given a set of colluding users \( U_c \), the unreachability of role \( \text{Done}[\sqrt{\cdot}] \) in \( \llbracket \mathcal{A} \downarrow U_c \llbracket \) ensures that the users in \( U_c \) cannot complete the workflow, not even by changing roles, hence we get a proof of security for \( \mathcal{A} \) against collusion by \( U_c \) (by Definition 9).

Interestingly, we are also able to establish a completeness theorem for the restricted case of ARBAC workflow systems without BoD constraints.

**Theorem 2** (Completeness). For any \( \mathcal{A} \) not including BoD constraints, if the role \( \text{Done}[\sqrt{\cdot}] \) is reachable in \( \llbracket \mathcal{A} \rrbracket \), we have that \( \text{T}^\vee (\mathcal{A}) \neq \emptyset \).

**Proof.** See the long version [10].

The completeness theorem is useful to confirm the presence of a collusion attack. Let \( \mathcal{A} \) be an ARBAC workflow system and \( U_c \) be a set of colluding users, and assume we proved, for instance by using the techniques in [21], [22], that \( \mathcal{A} \downarrow U_c \) does not allow the users in \( U_c \) to complete the workflow just with their original roles. By Theorem 2, if we show that role \( \text{Done}[\sqrt{\cdot}] \) is reachable in \( \llbracket \mathcal{A} \downarrow U_c \llbracket \) and the workflow does not include any BoD constraint, we can confirm that the users in \( U_c \) can complete the workflow by making use of administrative actions, hence \( \mathcal{A} \) is not secure against collusion by \( U_c \) (again by Definition 9).

We conjecture that the completeness theorem can be actually extended to arbitrary ARBAC workflow systems, but this stronger property looks significantly harder to prove than Theorem 2. We provide an intuition on why excluding BoD constraints helps in the proof of Theorem 2. One of the subtlest differences between ARBAC systems and ARBAC workflow systems is that in the former any role, including the roles of the form \( \text{Author}[e] \) for some event \( e \), can be assigned to all users satisfying the preconditions of the can-assign rules granting that role; conversely, a task can be performed only once in an ARBAC workflow system, so the author of each task is uniquely determined there. Thus, in the encoding into ARBAC, a role like \( \text{Author}[e] \) can be potentially assigned to multiple users, though only one of these users actually performs \( e \) in the ARBAC workflow system. To prove completeness, one needs to show that assigning \( \text{Author}[e] \) to multiple users does not introduce “false attacks”. Notably, if no BoD constraint is put in place in the workflow system, roles like \( \text{Author}[e] \) do not occur in any positive precondition of the can-assign rules generated by the encoding into ARBAC: this means that it is enough to have just one user who is assigned the role, to trigger those can-assign rules where \( \text{Author}[e] \) is in administrative position (i.e., it appears as the first element of the rule). Without loss of generality, it is thus possible in the proof of Theorem 2 to only focus on well-formed traces, where each role like \( \text{Author}[e] \) is assigned at most once.
can also be used to solve the classic workflow satisfiability problem [21], [22] in the case of ARBAC workflow systems without BoD constraints. Given \( \mathcal{A} = (\mathcal{P}, UR, W) \), let:

\[
\llbracket \mathcal{A} \rrbracket^* = (\llbracket W \rrbracket, \emptyset), \quad UR \cup \{ (u_0, \text{Super}) \},
\]

where \( u_0 \) is a fresh user extending the set of users \( U \).

**Lemma 1 (Workflow Satisfiability).** For any \( \mathcal{A} \), \( \mathbb{P}^\mathcal{A} (\mathcal{A}) \neq \emptyset \) implies that the role Done[\( \check{\varphi} \)] is reachable in \( \llbracket \mathcal{A} \rrbracket^* \). Moreover, if \( \mathcal{A} \) does not include BoD constraints, then also the converse holds true.

**Proof.** It is enough to observe that \( \mathbb{P}^\mathcal{A} (\llbracket \mathcal{P}_\perp, UR, W \rrbracket) = \mathbb{P}^\mathcal{A} (\llbracket \mathcal{P}_\perp \perp, UR, W \rrbracket) \), where \( \mathcal{P}_\perp = (\emptyset, \emptyset) \) is the empty ARBAC policy. The result then is an immediate consequence of the theorems above.

In terms of computational complexity, one can observe that the workflow satisfiability problem is NP-hard in presence of SoD constraints [22], which we consider in this work. The encoding \( \llbracket \cdot \rrbracket^* \) generates an ARBAC system without can-revoke rules, so the corresponding role reachability problem is NP-complete [17]. Hence, in the general case, both problems are equally hard, which in principle makes our approach a viable solution also for the classic workflow satisfiability problem.

We conclude this section by summarizing in Table IV how Theorem 1 and Theorem 2 can be used to check the security of an ARBAC workflow system \( \mathcal{A} \) against collusion by \( U_c \).

Notice that, if Done[\( \check{\varphi} \)] is not reachable in \( \llbracket \mathcal{A} \downarrow U_c \rrbracket \), we can immediately prove security and ignore \( \llbracket \mathcal{A} \downarrow U_c \rrbracket^* \). One case is missing from the table, since it is not possible that Done[\( \check{\varphi} \)] is reachable in \( \llbracket \mathcal{A} \downarrow U_c \rrbracket^* \), but not in \( \llbracket \mathcal{A} \downarrow U_c \rrbracket \).

**Table IV Checking security of \( \mathcal{A} \) against collusion by \( U_c \)**

<table>
<thead>
<tr>
<th>BoD</th>
<th>Reach in ( \llbracket \mathcal{A} \downarrow U_c \rrbracket )</th>
<th>Reach in ( \llbracket \mathcal{A} \downarrow U_c \rrbracket^* )</th>
<th>Secure</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>?</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>?</td>
</tr>
</tbody>
</table>

**D. Optimizations**

The encoding of ARBAC workflow systems into ARBAC we presented enjoys important formal properties, but it may lead to the generation of large and complex ARBAC systems, which do not admit an efficient role reachability analysis. Luckily, there is a well-established approach to make role reachability tractable for ARBAC, that is the use of pruning techniques, which perform syntactic transformations shrinking the size of the analysed ARBAC system, without affecting the reachability of a given role of interest [17], [15], [13].

We present here an effective pruning technique we devised to simplify the role reachability problems generated by our encoding: though the pruning has been designed to work at its best in this specific context, we expect several ideas to be general enough to be useful for the simplification of arbitrary ARBAC systems.

1) **Definitions:** A role \( r \) is administrative if and only if there exists at least one can-assign rule of the form \( (r, R_p, R_n, r_t) \) for some \( R_p, R_n, r_t \) or one can-revoke rule of the form \( (r, r') \) for some \( r' \). A role is positive (resp. negative) if and only if it occurs in the positive (resp. negative) preconditions of some can-assign rule [13]. We say that a role is purely administrative if it is administrative, non-positive and non-negative. Purely administrative roles only need to be assigned to grant the right of performing some administrative actions. As such, it is not important to know who is granted a purely administrative role, as long as there is one user having the role when the administrative actions it enables are intended to be performed.

A role is positively stable if and only if it is irrevocable or non-negative. A role is negatively stable if and only if it is not assignable or it is both non-positive and non-administrative. If a user is assigned a positively stable role \( r \), we can assume that \( r \) will be assigned to her forever, either because it cannot be removed (if \( r \) is irrevocable) or because removing it does not enable new administrative actions (if \( r \) is non-negative). Dually, if a user is not assigned a negatively stable role \( r' \), we can assume that \( r' \) will never be assigned to her.

2) **Preprocessing:** Before pruning, each rule of the form \( (\text{Super}, R_p, R_n, \text{Allowed}[e]) \) is replaced by the set of rules:

\[
\{ (\text{Author}[e'], R_p, R_n, \text{Allowed}[e]) \mid \text{Done}[e'] \in R_p \}.
\]

It can be shown that this preprocessing step does not affect the reachability of any role, since each rule of the previous format is enabled if and only if all the rules in the set generated from it are enabled. The intuition behind this observation is that, for each event \( e' \), role \( \text{Done}[e'] \) can be assigned if and only if role \( \text{Author}[e'] \) is first assigned.

Moreover, as part of the preprocessing step, all roles of the form \( \text{Allowed}[e] \) are removed from the negative preconditions of the can-assign rules generated by the encoding. Though this may look surprising, it can be shown that it does not affect the reachability of role \( \text{Done}[\check{\varphi}] \), since the negative preconditions of the previous form are just an artefact to simplify the proof of the main formal results. To understand why the reachability of \( \text{Done}[\check{\varphi}] \) is not affected by this change, observe that a role like \( \text{Allowed}[e] \) is only included in the negative preconditions of a can-assign rule generated by the encoding in Table III if \( e \) is conflicting with some other event. Assume then there are two conflicting events \( e_1, e_2 \) according to the encoding, only one role between \( \text{Author}[e_1] \) and \( \text{Author}[e_2] \) can be assigned, since only one between \( \text{Allowed}[e_1] \) and \( \text{Allowed}[e_2] \) can be given to the super user. However, assume that both \( \text{Author}[e_1] \) and \( \text{Author}[e_2] \) were assigned to some user, since we removed the negative preconditions above. This may lead to two potentially troublesome scenarios, which would never happen if only one of the two roles was assigned:

1) \( \text{Done}[e_1] \) and \( \text{Done}[e_2] \) get both assigned;

2) \( \text{Eq}[e_1, e_2] \) gets assigned (if \( \text{Author}[e_1] \) and \( \text{Author}[e_2] \) are given to the same user).
Both these cases do not affect the reachability of \( \text{Done}[\sqrt{\cdot}] \), since \( e_1 \) and \( e_2 \) are conflicting and thus never included in the same configuration of the stable event structure underlying the workflow. By the definition of the encoding, this implies that \( \text{Done}[e_1] \) and \( \text{Done}[e_2] \) never occur together in the positive preconditions of a can-assign rule, and similarly \( \text{Eq}[e_1, e_2] \) is not included in any positive precondition, so the cases above do not enable more administrative actions.

Though the preprocessing we discussed increases the number of can-assign rules generated by the encoding, the newly introduced rules have a format which is more amenable for pruning and, pragmatically, eventually leads to the generation of ARBAC systems which are smaller and easier to analyse. Alternatively, one could skip the preprocessing and fine-tune the pruning rules to improve their effectiveness, but this would make the rules harder to present.

3) Pruning Rules: The pruning rules make use of a binary relation on roles \( \preceq \). Intuitively, \( r \preceq r' \) ensures that, whenever \( r' \) is assigned to some user of an ARBAC system, then also \( r \) is assigned to some user (not necessarily the same one). Formally, \( \preceq \) is defined as the least pre-order satisfying the following two clauses:

1) \( r \preceq r' \)  for all \( r' \) if \( r \) is initially assigned and positively stable;
2) \( r \preceq r' \)  if \( r \) is positively stable, \( r' \) is initially unassigned, and for all can-assign rules of the form \((r_a, R_p, R_n, r')\) we have \( r \in R_p \cup \{r_a\} \).

We are finally ready to present the pruning rules, assuming an initial user-to-role assignment \( UR \):

1) Let \( r_t \) be a non-negative role. If there exist a rule \( ca = (r, R_p, R_n, r_t) \) and a rule \( ca' = (r', R'_p \cup \{r\}, R'_n, r'_t) \) with \( R_p \subseteq R'_p \) and \( R_n \subseteq R'_n \) and there exists \( r'' \in R'_p \cup \{r'\} \) such that \( r \preceq r'' \) then replace \( ca' \) with \( (r', R'_p, R_n, r'_t) \);
2) Let \( r_t \) be a role. If there exist a rule \( ca = (r, R_p, R_n, r_t) \) and a rule \( ca' = (r', R'_p, R'_n, r_t) \) with \( R_p \subseteq R'_p \) and \( R_n \subseteq R'_n \) and there exists \( r'' \in R'_p \) such that \( r \preceq r'' \) then replace \( ca' \);
3) Let \( r_t \) be a purely administrative role and let \( r \preceq r_t \). If there exist a rule \( ca = (r, R_p, R_n, r_t) \) and a user \( u \) such that \( R_p \subseteq UR(u) \) and \( R_n \cap UR(u) = \emptyset \), the roles in \( R_p \) are positively stable and the roles in \( R_n \) are negatively stable, then remove \( ca \) and replace all the occurrences of \( r_t \) with \( r \) in the can-assign/can-revoke rules.

Rule 1 says that, if there exist a rule \( ca \) assigning a non-negative role \( r_t \) and a rule \( ca' \) including \( r_t \) in the positive preconditions, we can drop \( r_t \) from the positive preconditions of \( ca' \), as long as we are guaranteed that \( ca \) is always enabled when \( ca' \) is enabled up to the absence of \( r_t \). Rule 2 says that, if we have two rules \( ca \) and \( ca' \) assigning the same role \( r_t \) and \( ca \) is always enabled when \( ca' \) is enabled, we can remove rule \( ca' \). These rules are reminiscent of two pruning rules originally presented in [15], noted there as rules \( R_2 \) and \( R_5 \) respectively, but they make use of the \( \preceq \) relation to be more general and more effective on our cases.

Rule 3 deals with the assignment of purely administrative roles and it is trickier: it allows one to delegate the administrative rights of a purely administrative role \( r_t \) to any \( r \preceq r_t \). Notice that this change always preserves role reachability, since whenever \( r_t \) is assigned to some user, also \( r \) must be assigned to someone by definition of the \( \preceq \) relation. However, this change could alter the semantics of the ARBAC system if \( r \) is assigned, but \( r_t \) is not immediately assignable. To ensure that this does not happen, we have to check a few additional conditions. In particular, rule 3 checks the existence of a can-assign rule \( ca = (r, R_p, R_n, r_t) \) whose preconditions are satisfied by some user \( u \) in the initial user-to-role assignment: if the roles in \( R_p \) are positively stable and the roles in \( R_n \) are negatively stable, we can assume that the preconditions of \( ca \) will always be satisfied by \( u \), hence ensuring that \( r_t \) is always assignable when \( r \) is assigned to someone.

The pruning algorithm just amounts to continuously applying rules 1-3 to the ARBAC system, until no more rules can be applied. Termination is ensured by the observation that all rules reduce the size of the ARBAC system, either in terms of the number of can-assign/can-revoke rules or in terms of the size of the positive preconditions of the can-assign rules.

The pruning algorithm enjoys the following property:

**Theorem.** Role \( \text{Done}[\sqrt{\cdot}] \) is reachable in \([\mathcal{A}]\) if and only if it is reachable after pruning \([\mathcal{A}]\).

**Proof.** See the long version [10].

\[ \square \]

V. EXAMPLES

We show the static analysis at work on some simple, but representative examples. To improve readability, we only present the most interesting subset of the can-assign rules generated by the encoding into ARBAC. We do not apply the pruning algorithm to these simple examples, but we just present the result of the direct application of the encoding.

A. Exclusive Choice

Consider the workflow described in the motivating example in Section I-A (Figure 1), its translation into ARBAC is given in Table V.

Notice that roles Allowed[\( b \)] and Allowed[\( c \)] are mutually exclusive, since events \( b \) and \( c \) are conflicting: this implies that only one role between Author[\( b \)] and Author[\( c \)] can be assigned. Correspondingly, there are two ways to introduce role Allowed[\( \sqrt{\cdot} \)]: indeed, recall that role Done[\( b \)] / Done[\( c \)] can only be assigned by a user with role Author[\( b \)] / Author[\( c \)]. Moreover, notice that both Author[\( b \)] and Author[\( c \)] can only be assigned to a user who is assigned Author[\( a \)], thus ensuring that the BoD constraints in the workflow are satisfied. Finally, observe that the roles required to perform \( a, b, c \) according to the task-to-role assignment function of the workflow are included in the positive preconditions of the rules assigning the corresponding author role.
Consider the workflow in Figure 2, including three sequential tasks $a, b, c$ such that $a$ and $c$ must be performed by different authors. Its translation into ARBAC is given in Table VI.

Observe that role $Author[c]$ can only be assigned to a user who is not assigned $Author[a]$ and vice-versa, thus ensuring that the SoD constraint between $a$ and $c$ is satisfied.

Consider the workflow in Figure 3, including two parallel tasks $a, b$, which must both be performed before completing the workflow, graphically noted by joining the two edges entering $✓$ (formally, this can be represented by the enabling relation $∅ ⊢ a, ∅ ⊢ b$ and $\{a, b\} ⊢ ✓$). Moreover, assume there exists a BoD constraint between $a$ and $b$.

The translation of the workflow into ARBAC is shown in Table VII. Since there is no temporal dependence between $a$ and $b$, it is not possible to predict which of the two tasks is executed before; hence, both $Allowed[a]$ and $Allowed[b]$, as well as $Author[a]$ and $Author[b]$, can be liberally assigned without checking the BoD constraint between $a$ and $b$. However, $Allowed[✓]$ can only be introduced whenever $Eq[a, b]$ is assigned to the super user, which is only possible when there exists a user who is assigned both $Author[a]$ and $Author[b]$, i.e., when the BoD constraint between $a$ and $b$ is satisfied.

VI. IMPLEMENTATION

We developed WARBAC, a tool for checking the security against collusion of ARBAC workflow systems. Given an input file including the specification of an ARBAC workflow system $\mathcal{A}$ and a set of colluding users $U_c$, it runs a security verification by: (1) removing from $\mathcal{A}$ all the users not included in $U_c$; (2) encoding the security problem for the resulting system in terms of role reachability for ARBAC, based on the presented theory; and (3) simplifying the role reachability problem by running the pruning algorithm in Section IV-D. The generated role reachability queries for role $Done[✓]$ are then discharged by an existing state-of-the-art tool, VAC [13].

A. Implementing the Analysis

WARBAC reduces security against collusion to role reachability by implementing the checks summarised in Table IV (in Section IV-C). Since role reachability may be costly to check,
WARBA exploits different analysis backends supported by VAC to make the security analysis more efficient:

1) INTERPROC [2]: efficient, sound, but over-approximated analysis. A negative answer by INTERPROC proves role unreachability, but a positive answer may be incorrectly returned as the result of an over-approximation [14];

2) CBMC [1]: efficient bounded model-checking. A positive answer by CBMC proves role reachability, but a negative answer may be incorrectly returned as the result of a bound on the search space [13];

3) NuSMV [3]: computationally expensive sound and complete analysis. A positive/negative answer by NuSMV proves role reachability/unreachability [13].

When testing role reachability, WARBA first tries to prove unreachability by using INTERPROC; if this fails, it attempts to prove reachability by using CBMC with a depth search empirically set to 14; if this also fails, it resorts to running NuSMV to get a final answer.

Moreover, WARBA tries to simplify as much as possible the generated role reachability problems before attempting to solve them. In particular, it applies the following procedure: 1) run the pruning algorithm described in Section IV-D, and 2) run the pruning algorithm internally implemented in VAC, until no further simplification is possible.

B. Experiments

We created a set of examples to test WARBA, which we make available online [10]. All the examples refer to a medical setting: specifically, we extended an existing ARBAC system (the Hospital case study in [13]) with the specification of a number of different workflows. Most of the workflows we developed were larger, more complicated variants of the examples shown in Section V, implementing different patterns: sequential execution, parallel execution and exclusive choice. We also developed more complex workflows, representing realistic first-aid procedures. All the experiments were performed on a 64-bit Intel Xeon running at 2.4 GHz.

1) Synthetic Examples: Table VIII reports on the experimental results. The table shows for each example the following information:

1) the main pattern underlying the workflow, e.g., sequential;
2) the number of tasks in the workflow;
3) the type of enforced constraints (BoD or SoD);
4) the number of colluding users originally in input, after the pruning implemented in VAC is enabled, and after the full pruning is enabled;
5) the number of can-assign/can-revoke rules originally in input, after the pruning implemented in VAC is enabled, and after the full pruning is enabled;
6) the aggregate analysis time when only the pruning implemented in VAC is enabled and when the full pruning is enabled (the analysis never terminates within one hour if no form of pruning is enabled, so we do not report this information);
7) the expected analysis result (safe or unsafe) and the answer reported by WARBA.

The first obvious observation from the table is that enabling the full pruning algorithm is very important for the scalability of the analysis: in 13 out of 21 examples the improvement in performances is dramatic, with 7 cases failing to terminate within one hour if only the internal pruning of VAC is enabled, but analysed in a few minutes if the full pruning is used. There are 8 cases where activating the full pruning turns out to be overshooting, since the pruning performed by VAC is already very effective and the additional overhead of running the full pruning is not justified. Still, all these cases can be solved in seconds in both scenarios.

Most of the safe cases required WARBA to only run INTERPROC. As expected, the over-approximated analysis implemented in INTERPROC is very fast, since it only takes a few seconds in all the test cases, even the most complicated ones. Though over-approximated, the analysis performed by INTERPROC is useful in many practical cases, e.g., when it finds that a role required to complete the workflow is neither assigned initially, nor assignable to any of the colluding users. When INTERPROC is not able to prove security, WARBA runs CBMC and possibly NuSMV to get a more precise answer. Though the analysis implemented in NuSMV is potentially costly, our optimization techniques proved very effective to provide good analysis times, even for large settings.

We find it promising that realistic cases like the first-aid procedures described in the next section are analysed in minutes, though in some cases WARBA is unable to prove security, since our analysis is not proved complete for workflows using BoD constraints.

2) Case Study: The main case study we considered in our experiments is a workflow modelling a procedure to assist a patient in need for a first aid treatment. The workflow includes 10 different tasks:

a) a patient comes at the hospital and gets a ticket;

<table>
<thead>
<tr>
<th>Table VII Translation of the workflow in Figure 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Super, {Super}, φ, Allowed[a])</td>
</tr>
<tr>
<td>(Super, {Super}, φ, Allowed[b])</td>
</tr>
<tr>
<td>(Super, {Done[a]}, Done[b], Eq[a, b], Super}, φ, Allowed[✓])</td>
</tr>
<tr>
<td>(Super, {Author[a]}, Author[b], φ, Eq[a, b])</td>
</tr>
<tr>
<td>(Eq[a, b], {Super}, φ, Eq[a, b])</td>
</tr>
<tr>
<td>(Allowed[a], φ, {Super, Author[a]})</td>
</tr>
<tr>
<td>(Allowed[b], φ, {Super, Author[b]})</td>
</tr>
<tr>
<td>(Allowed[✓], {Super}, φ, Author[✓])</td>
</tr>
</tbody>
</table>
Table VIII Experimental results

<table>
<thead>
<tr>
<th>Type</th>
<th>Tasks</th>
<th>Cons</th>
<th>Orig</th>
<th>VAC</th>
<th>FullPrune</th>
<th>Orig</th>
<th>VAC</th>
<th>FullPrune</th>
<th>Aggregate Time</th>
<th>FullPrune</th>
<th>Expected</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>74</td>
<td>47</td>
<td>55</td>
<td>37</td>
<td>23</td>
<td>&gt;60m</td>
<td>2m41s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>sequential</td>
<td>9</td>
<td>BoD</td>
<td>121</td>
<td>76</td>
<td>46</td>
<td>57</td>
<td>39</td>
<td>22</td>
<td>&gt;60m</td>
<td>2m31s</td>
<td>U</td>
<td>?</td>
</tr>
<tr>
<td>sequential</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>5</td>
<td>3</td>
<td>55</td>
<td>2</td>
<td>1</td>
<td>6s</td>
<td>20s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>sequential</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>86</td>
<td>53</td>
<td>64</td>
<td>46</td>
<td>26</td>
<td>&gt;60m</td>
<td>4m36s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>sequential</td>
<td>12</td>
<td>BoD</td>
<td>121</td>
<td>88</td>
<td>25</td>
<td>66</td>
<td>48</td>
<td>17</td>
<td>&gt;60m</td>
<td>1m45s</td>
<td>U</td>
<td>?</td>
</tr>
<tr>
<td>sequential</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>5</td>
<td>4</td>
<td>64</td>
<td>2</td>
<td>1</td>
<td>5s</td>
<td>56s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>parallel</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>74</td>
<td>47</td>
<td>55</td>
<td>37</td>
<td>23</td>
<td>&gt;60m</td>
<td>2m39s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>parallel</td>
<td>9</td>
<td>BoD</td>
<td>121</td>
<td>6</td>
<td>3</td>
<td>57</td>
<td>3</td>
<td>1</td>
<td>4s</td>
<td>22s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>parallel</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>5</td>
<td>3</td>
<td>55</td>
<td>2</td>
<td>1</td>
<td>4s</td>
<td>22s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>parallel</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>88</td>
<td>28</td>
<td>64</td>
<td>46</td>
<td>17</td>
<td>&gt;60m</td>
<td>4m05s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>parallel</td>
<td>12</td>
<td>BoD</td>
<td>121</td>
<td>88</td>
<td>25</td>
<td>66</td>
<td>48</td>
<td>17</td>
<td>53m52s</td>
<td>1m46s</td>
<td>U</td>
<td>?</td>
</tr>
<tr>
<td>parallel</td>
<td>12</td>
<td>BoD</td>
<td>121</td>
<td>6</td>
<td>3</td>
<td>66</td>
<td>3</td>
<td>1</td>
<td>5s</td>
<td>51s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>choice</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>28</td>
<td>21</td>
<td>57</td>
<td>41</td>
<td>28</td>
<td>20m56s</td>
<td>5m55s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>choice</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>64</td>
<td>46</td>
<td>57</td>
<td>32</td>
<td>19</td>
<td>5m1s</td>
<td>1m12s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>choice</td>
<td>9</td>
<td>SoD</td>
<td>121</td>
<td>4</td>
<td>3</td>
<td>57</td>
<td>2</td>
<td>1</td>
<td>4s</td>
<td>20s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>choice</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>42</td>
<td>33</td>
<td>66</td>
<td>52</td>
<td>36</td>
<td>&gt;60m</td>
<td>17m15s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>choice</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>89</td>
<td>50</td>
<td>66</td>
<td>48</td>
<td>25</td>
<td>42m02s</td>
<td>2m53s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>choice</td>
<td>12</td>
<td>SoD</td>
<td>121</td>
<td>4</td>
<td>3</td>
<td>66</td>
<td>2</td>
<td>1</td>
<td>5s</td>
<td>40s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>first-aid</td>
<td>10</td>
<td>SoD</td>
<td>121</td>
<td>30</td>
<td>20</td>
<td>65</td>
<td>49</td>
<td>37</td>
<td>35m17s</td>
<td>12m09s</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>first-aid</td>
<td>10</td>
<td>SoD</td>
<td>121</td>
<td>50</td>
<td>34</td>
<td>65</td>
<td>49</td>
<td>37</td>
<td>43m15s</td>
<td>15m32s</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>first-aid</td>
<td>10</td>
<td>SoD</td>
<td>121</td>
<td>5</td>
<td>3</td>
<td>65</td>
<td>2</td>
<td>1</td>
<td>5s</td>
<td>29s</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

b) a doctor makes a preliminary evaluation and sends the patient in for a visit (f), while she provides the relative documentation to a receptionist (c);

c) a receptionist makes the paperwork summarizing the conditions of the patient, possibly while her visits are still ongoing;

d) when the papers are ready (c done) and the patient has been dismissed (i done), a receptionist closes the patient case suggesting additional treatment (j) or not (c);

e) the patient is fine: she shows the paperwork at the exit and leaves the hospital (√);

f) after the preliminary evaluation, a nurse marks the case as urgent (g) or not (h);

g) urgent case: a doctor treats the patient;

h) non-urgent case: a nurse treats the patient;

i) treatment done: a doctor dismisses the patient;

j) the patient needs additional treatment: she takes an appointment for a specialist examination and leaves (√).

The workflow enforces a SoD constraint between b and i: the doctor who first evaluates the case must be different from the doctor who dismisses the patient, so as to ensure a more thorough examination of the patient’s conditions. For readability, the workflow is graphically represented in Figure 4, where we use the letters P, D, N to represent roles Patient, Doctor, Nurse respectively.

VII. Related Work

The paper by Wang et al. on the security of delegation in access control systems [23] was one of the main sources of inspiration for the present work. The paper studies how delegation can be abused by colluding users to bypass the intended security policies in a workflow system. Though the security property we consider in the present paper is an adaptation of the security property in [23] to the case of ARBAC administrative actions, there are several notable differences between this work and [23]. First, static analysis is only briefly mentioned in [23] as a possible way to check security, but no static analysis is actually proposed by the authors: rather, the solution they develop requires an extension of the workflow system with additional runtime checks, which is inconvenient or even impossible in many practical scenarios. Second, we experimentally validate the applicability of our theory: developing a sound - yet precise - static analysis for workflow systems built on top of realistic ARBAC policies is hard, since intractability lurks around the corner [20]; indeed, we observed the need to aggressively optimise our encoding into ARBAC to obtain an efficient static analysis. Finally, the formal model in this paper is quite different and more general than the one in [23]: in particular, we focus on the ARBAC standard rather than on an ad-hoc extension of RBAC with delegation and we consider a more expressive model of workflows based on stable event structures rather than on a partially ordered set of tasks.

The satisfiability (or consistency) of workflows is a classic problem in computer security [21], [22]. Roughly, a workflow is satisfiable with respect to a given user-to-role assignment UR if and only if the users included in UR are able to complete it. Checking security against collusion (Definition 9) may require one to check the satisfiability of a workflow. However, as we discussed, checking security against collusion requires one to generalize algorithms for workflow satisfiability to deal with the presence of administrative actions changing the initial user-to-role assignment UR. Building all the possible user-to-role assignments and checking satisfiability with respect to
them is not feasible in practice, given the exponential blow-up of the possible role combinations assigned to the users of the system and the fact that workflow satisfiability is NP-hard in most practical cases [22].

Crampton and Khambhammettu proposed algorithms to check the satisfiability of workflows supporting delegation operations [11]. These algorithms ensure that permitting a delegation request does not prevent the completion of a workflow. Their goal is then essentially dual to the static analysis in this paper, which ensures that administrative actions cannot be abused to complete a workflow which could not be completed under the original user-to-role assignment. Indeed, one should observe that there is often a trade-off between security and business continuity: if collisions are the main concern for system administrators, the present paper proposes a viable solution; if instead it is better to ensure workflow termination at the price of permitting collision, the approach in [11] should be considered. We argue that these considerations strongly depend on the application scenario, the workflow semantics and the considered set of users.

Basin et al. conducted a formal study on the tension between security policies and business objectives in workflow systems represented as CSP processes [6]. They formalize a notion of obstruction, generalizing the notion of deadlock for systems where access control policies are enforced. Roughly, an obstruction happens when the enforcement of an access control policy prevents a possible execution path in a workflow. The paper presents the design and the implementation of an obstruction-free authorization enforcement mechanism for workflow systems.

VIII. CONCLUSION

We studied the problem of collusion attacks in ARBAC-based workflow systems, where malicious users may change the user-to-role assignment in the attempt of sidestepping the intended security policies. We formulated a formal definition of security against collusion and we proposed a novel static analysis technique which can be used to prove or disprove security for a large class of ARBAC workflow systems. We discussed how to aggressively optimise the static analysis to ensure its efficiency in practice and we showed the feasibility of our approach by implementing a tool, WARBAC, and by performing an experimental evaluation on a set of publicly available examples.

There are many avenues for future work. We would like to extend our theory to the case of workflows including loops, which are quite popular in practice, but were left out from the present paper for the sake of simplicity, most notably because the interaction between loops and BoD/SoD constraints is quite subtle [6]. Moreover, we plan to design and implement a translator from high-level workflow description languages like BPMN into event structures, thus making WARBAC easier to use. Again on the practical side, we plan to extend WARBAC with a module which, given a role reachability trace returned by VAC, verifies whether this trace actually corresponds to a successful trace of the workflow system: this would be very useful to improve the practicality of WARBAC in absence of a stronger completeness result for our static analysis.

Acknowledgements: We would like to thank the anonymous reviewers for their useful suggestions. This work was partially supported by the MIUR project ADAPT.

REFERENCES


