CORSO DI DOTTORATO DI RICERCA
IN INFORMATICA

CICLO XXX

TESI DI RICERCA

Efficient security analysis of Administrative Access Control Policies

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01

COORDINATORE DEL DOTTORATO
Prof. Riccardo FOCARDI

SUPERVISORE
Prof. Michele BUGLIESI

DOTTORANDO
Enrico STEFFINLONGO
Matricola 826043

To Emilio (Milo)

Abstract

In recent years access control has been a crucial aspect of computer systems, since it is the component responsible for giving users specific permissions enforcing a administrator-defined policy. This lead to the formation of a wide literature proposing and implementing access control models reflecting different system perspectives. Moreover, many analysis techniques have been developed with special attention to scalability, since many security properties have been proved hard to verify. In this setting the presented work provides two main contributions.

In the first, we study the security of workflow systems built on top of an attribute-based access control in the case of collusion of multiples users. We define a formal model for an ARBAC based workflow system and we state a notion of security against collusion. Furthermore we propose a scalable static analysis technique for proving the security of a workflow. Finally we implement it in a prototype tool showing its effectiveness.

In the second contribution, we propose a new model of administrative attribute-based access control (AABAC) where administrative actions are enabled by boolean expressions predicating on user attributes values. Subsequently we introduce two static analysis techniques for the verification of reachability problem: one precise, but bounded, and one over-approximated. We also give a set of pruning rules in order to reduce the size of the problem increasing scalability of the analysis. Finally, we implement the analysis in a tool and we show its effectiveness on several realistic case studies.

Acknowledgements

Many people played a fundamental role in the development of this work. Firstly I would like to thank my wife Elisabetta, and all my family, especially my sister Anna and my mother Luisa for all their support and patience.
Many thanks also to my supervisor Michele Bugliesi who was always able to enlighten me with his experience and enthusiasm. Notwithstanding his major commitments, he has always been available to guide me and suggest me friendly and efficiently.
I would also like to thank Stefano Calzavara who co-supervised my Ph.D. for the constant precious time he dedicated to this work. Without his help and support this research would have been much harder to be carried out.
Also a major thank is due to Gennaro Parlato and Annalisa Ferrara who hosted me during my period abroad at Southampton University (UK) for almost one year. The cooperation with Gennaro has enriched me largely for his constant and passionate sharing of know-how and experience. Working at his side for this year has been priceless.
Finally I want to thank all my colleagues and friends with whom I shared good and difficult moments of this Ph.D.. Especially I am indebted with Alvise Spanò and Mikhail Ramalho for their patience, sharing of experience and insightful technical suggestions on several practical problems.
Another big thanks is also due to Gianluca Barbon for been a trustful confidant and for having a friendly shoulder throughout the many difficulties.
Last, but not least I want to thank Giuseppe Maggiore, Mohamed Abbadi and Francesco Di Giacomo for inspiring me and suggesting me this path.
The last acknowledgment is for my grandfather Emilio, for having been an example to me and for having taught me, since I was a baby, the importance of study and the beauty of science.

Contents

List of Tables

Preface

This thesis presents the result of my research work during my Ph.D. studies in Computer Science at Università di Ca' Foscari – Venezia from September 2014 to September 2017, under the supervision of Michele Bugliesi. Some of the Ph.D. research activities have been conducted at the University of Southampton (U.K.), as a joint work with Gennaro Parlato. Chapter 2 is the presentation of a joint work conducted with Michele Bugliesi Stefano Calzavara and Alvise Rabitti presented at the 29^{th} IEEE Computer Security Foundations Symposium in June 2016 [19]. Chapter 4 presents an ongoing joint project with Michele Bugliesi, Stefano Calzavara, Anna Lisa Ferrara, Truc L. Nguyen, and Gennaro Parlato. At the time of writing it has not been published, nor submitted because some challenging topics have been found, and we are working to complete it. It gives the start to further study to be conducted and hopefully we will conclude it and publish it soon. Appendix A reports a joint project with Michele Bugliesi, Stefano Calzavara, Silvia Crafa that has been presented at the 24^{th} European Symposium on Programming in April 2015 [17]. In that work I contributed by implementing the tool CHENand discussing with Stefano Calzavara about the work. During my Ph.D. studies I also completed a joint work with Gianluca Barbon, Agostino Cortesi and Pietro Ferrara that has been presented at the 12^{th} International Workshop on Security and Trust Management in September 2016 [7]. This work however is not part of this thesis.

Chapter 1

Introduction

Nowadays computer systems offer crucial services in almost all fields of our lives and influence our way of living in a pervasive way. All the services we normally access to are managed and controlled by sophisticated systems. A main concern derived by the usage of those complex systems is the necessity of restricting the access to sensitive resources. Access control became in the last decades a very important field of research, aiming at defining an effective way to specify correct security policies. The literature on this field is mainly operating around three main topics: firstly defining expressive models to express in an effective way which resources are sensitive and who should be entitled to access them (*policy specification*). Secondarily, there is a need for guaranteeing a correct enforcement mechanism of the above-defined policies (*policy enforcement*). Finally, we have to consider the problem of *policy verification*: namely, the crucial test which proves if the given policies satisfy certain security-relevant properties. The main contributions of this thesis are related to the first and especially the third topics.

Policy specification. Many ways of specifying policies have been proposed in the literature, from the precise Access Control Matrix (ACM) to more abstract ones such as Role-Based Access Control, Attribute-Based Access Control and Relationship-Based Access Control (ReBAC). In Access Control Matrix there is a punctual mapping from user to permissions; this model although extremely fine-grained is hard to maintain in large systems. In Role-Based Access Control (RBAC), users obtain specific permissions based on the roles they have. In Attribute-Based Access Control (ABAC) the permissions are given on the basis of formulas predicating on the attribute of users and in Relationship-Based Access Control (ReBAC) the permissions are granted if there is a particular relationship between the user and the resource.

To be able to work within these static systems, the problem of the policy administration is crucial. In order to avoid the need of recurring to a centralized administrator every time we require to change the permissions of a user, it is important to define a way to give users capability of managing permissions of themself. For this reason several administrative versions of access control models have been proposed. On one hand the self-administration capability dramatically enhanced the expressiveness and the maintenance of the policy. On the other hand, writing a correct administrative policy is a challenging problem, since privileges assigned to users change dynamically due to administrative actions, thus potentially leading to unintended security breaches.

Policy verification. The main idea of *policy verification* is that we want to verify if a certain access control policy satisfies specific security properties. This is essential, especially when we have a dynamic policy such in the administrative version of access control systems. Indeed, for an administrator is very hard to write a policy that respects some security properties he has in mind. For this reason, substantial researches have been carried out to address the verification problem of access control policies, in order to check a wide area of different security properties.

Let us briefly present ARBAC to give a more concrete example of the influence of the *policy specification* and *verification* in the real world.

Example: Role-Based Access Control. A sophisticated way of specifying access control policies, that is widely used, is the so called Role-Based Access Control (RBAC). It has been standardized by NIST [30], and it is used in many commercial applications e.g., OpenMRS, Oracle DBMS, grsecurity, Moodle. Its fame comes by the fact that it allows the deployment of access control policies in systems with a large number of users and permissions. The key idea is that sets of permissions, are abstracted into roles which are assigned to users, so that every user is granted permissions depending on the roles assigned to him. This way of specifying the access control policies is very effective and intuitive since the number of roles is static and much smaller than the number of users and permissions and each role intuitively represents the position of users in the organization.

Although RBAC become a very famous access control model, its lack of distributed administration severely limits its effectiveness. Indeed if we want to change the user role assignment, that is static, we need the administrator to change it, making the maintenance unfeasible in case there is a large number of users involved. To address this shortcoming, an administrative version of it, namely ARBAC, has been proposed and adopted by many computer systems. In this version, we allow users involved in the system to assign and revoke roles to other users on the basis of an administration policy, thus greatly enhancing the expressiveness of the system: now we do not have a centralized administrator, but users in the system are able to self-administrate themself. However writing correct ARBAC policies is hard, making the *verification* process an almost necessary step. The main *verification* problem for ARBAC is the role reachability one: given an ARBAC policy, a set of users with a certain initial configuration and a target role, we want to prove if it is possible to reach a new configuration such that a user have the target role. Many desirable security properties of ARBAC system have been encoded in terms of instances of role reachability. This problem is notoriously hard to solve due to the exponential explosion of the state space, but many scalable analysis techniques have been developed [71, 31, 18, 41].

Contributions

This thesis makes its main contributions in the field of *specification* and *verification* of access control systems.

In the first part of this thesis we conduct a study of workflow systems. Workflow systems are a typical example where the access control component is used to enforce a security policy in a wider computing system. In a workflow system we have a set of tasks composing a business process and a set of users that cooperate to complete them, and an access control system enforcing a security policy that is responsible to allow or disallow users to complete each workflow task.

In this work, we present several contributions on this topic: we propose a model of workflow systems based on stable event structures, where the authorization to complete tasks is managed by an ARBAC access control component with the addition of binding and separation of duty constraints on users. We then define a notion of security against collusion ensuring that administrative actions cannot be abused to sidestep the workflow security policy. Subsequently, we 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 against collusion for restricted yet useful classes of workflow systems. We also aggressively optimise the role reachability problem to ensure its tractability. Finally we implement our static analysis in the WARBAC tool, and we show its effectiveness in realistic case studies.

The second part of this work contains a study on Attribute-Based Access Control systems. Attribute-Based Access Control, ABAC, proved to be a very intuitive and effective access control system holding several advantages compared to other systems. The key idea of ABAC, similarly to RBAC, is that permissions are given using a declarative security policy that abstracts from the users in the system. However, instead of predicating on an intermediate layer such as roles in RBAC, here the policy predicates directly on attributes of users, enabling a finer granularity of the access control system. In this work we give a model for an administrative version of ABAC, AABAC, that is deeply inspired by ARBAC. It allows a distributed management of the system, where administrative actions are guarded by preconditions predicating on the attributes of the involved users. We then give two scalable approximated, but refinable, analysis techniques to verify the given model and a pruning technique aimed at reducing the size of the problem enhancing scalability of the analysis. Finally we discuss the results of the VACSAT tool, that implements our *verification* techniques, on a large set of AABAC policies.

In the appendix of this thesis we also present a work on security of Google Chrome browser extensions.

1.1 Structure of this thesis

This thesis is composed of three parts:

- in chapter 2 we present the study of collusion attacks in ARBAC-based workflow systems. Proofs of this work are reported in chapter 3;
- in chapter 4 we propose a new administrative model for attribute-based access control and we give various analysis techniques;
- in chapter 5 we conclude.

Chapter 2

Static Detection of Collusion Attacks in ARBAC-based Workflow Systems

2.1 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 [2, 10]. When role-based security policies on task execution are not expressive 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 [13, 52]. 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 [30] 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 [45, 69].

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.

2.1.1 Motivating Example

We graphically represent workflows as directed graphs, including one node per task, a start node $\cdot$ and an end node $\checkmark$. 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. For instance, Figure 2.1 represents a workflow with three tasks a, b, c, which can be performed by any user who is granted role $\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3$ respectively. After the execution of task a, the workflow offers an exclusive choice between tasks b and c and, no matter which task is chosen, the second task must be performed by the same user who performed a.

Consider now two users u_1 and u_2 who are assigned roles $\mathsf{R}_1, \mathsf{R}_2$ 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 [30]. Under this policy, users u_1 and u_2 can

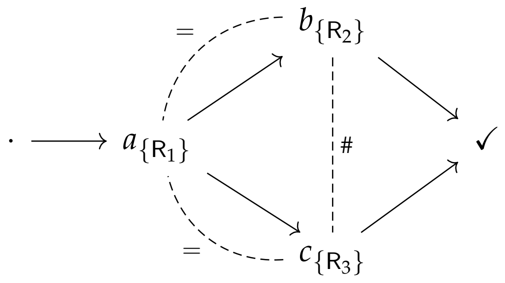


FIGURE 2.1: Example of workflow

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.

2.1.2 Contributions

In this work, we make the following contributions:

1. we propose a formal model of workflows based on *stable event structures* [76], which generalizes previous proposals based on partial orders [72, 74]. 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 [75];

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 [31, 18, 41] 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 work (with proofs) available online [19].

Structure of the Work Section 2.2 presents the operational model. Section 2.3 introduces the formal definition of security and gives an example. Section 2.4 details the reduction to role reachability and its optimization. Section 2.5 shows a few example reductions. Section 2.6 presents WARBAC and reports on the experimental results. Section 2.7 discusses related work. Section 2.8 concludes and hints at future work.

2.2 Operational Model

Our operational model is obtained by integrating a standard ARBAC model, as formalized, e.g., in [18, 31], with a workflow represented as a *stable event structure* [76], extended with a security policy assigning required roles to tasks and supporting both BoD and SoD constrains.

2.2.1 ARBAC

We presuppose the existence of finite sets of users U and roles R.

Definition 1 (ARBAC Policy). *An* ARBAC policy *is a pair* $\mathcal{P} = \langle CA, CR \rangle$, *where* $CA \subseteq R \times 2^R \times 2^R \times R$ *is a can-assign relation and* $CR \subseteq R \times R$ *is a can-revoke relation.*

A can-assign rule $(r_a, R_p, R_n, r_t) \in CA$ states that a user with role r_a can assign role r_t to any user who has all the roles in the set R_p (the *positive* preconditions) and none of the roles in the set R_n (the *negative* preconditions). A can-revoke rule $(r_a, r_t) \in CR$, instead, states that a user with role r_a can unconditionally revoke role r_t from any user.

Definition 2 (ARBAC System). *An* ARBAC system *is a pair* $\mathcal{S} = \langle \mathcal{P}, UR \rangle$, *where* $\mathcal{P}$ *is an ARBAC policy and* $UR \subseteq U \times R$ *is an initial user-to-role assignment.*

For any user u, let $UR(u) = \{r \mid (u, r) \in UR\}$. The operational semantics of an ARBAC system $\mathcal{S} = \langle \mathcal{P}, UR \rangle$ is defined by the changes which can be performed to the initial user-to-role assignment UR according to the policy $\mathcal{P}$. This is defined by the reduction relation $\mathcal{P} \rhd UR \rightsquigarrow UR'$ in Table 2.1, providing the formal counterpart of the intuitions above.

Table 2.1 Reduction semantics for ARBAC ($\mathcal{P} \rhd UR \rightsquigarrow UR'$)

(R-ASSIGN)
$$\frac{(u_a, r_a) \in UR \quad (r_a, R_p, R_n, r_t) \in CA \quad R_p \subseteq UR(u) \quad R_n \cap UR(u) = \emptyset}{\langle CA, CR \rangle \rhd UR \rightsquigarrow UR \cup \{(u, r_t)\}}$$

(R-REVOKE)
$$\frac{(u_a, r_a) \in UR \quad (r_a, r_t) \in CR}{\langle CA, CR \rangle \rhd UR \rightsquigarrow UR \setminus \{(u, r_t)\}}$$

2.2.2 Workflows

We propose a general, abstract model of workflows based on *stable event structures*, a true concurrency model originally proposed by Winskel [76]. Since stable event can naturally model sequential and parallel execution of tasks, as well as non-deterministic choices [51, 76], they are very appealing candidates to represent workflows. Moreover, stable event structures are more expressive than previous models of workflows based on a partially ordered set of tasks [72, 74], since the latter correspond to *elementary event structures* [77] and hence cannot represent non-determinism, which instead is a desirable feature for many workflows (including the one in Figure 2.1). 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 [76] 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 [51]: it is more compact than the original formulation in [76], but semantically equivalent to it. We just omit the labelling function from the definition, since it does not play any role in the present work.

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:*

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

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 [76].

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. $\forall e, e' \in X : (e, e') \notin \#$*;*

2. $\forall e \in X. \exists e_1, \ldots, e_n \in X : e_n = e \wedge \forall i \leq n. \exists Y \subseteq \{e_1, \ldots, e_{i-1}\} : Y \vdash e_i.$

We let $\mathbb{F}(\mathcal{E})$ *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 work, picking the most natural choice based on the context of the discussion. A workflow is a stable event structure including a special event $\checkmark$, representing completion, and extended with a set of constraints C and a task-to-role assignment function ρ. 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 ρ 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. *$\mathcal{E} = \langle E \cup \{\checkmark\}, \#, \vdash \rangle$ is a stable event structure including an event $\checkmark \notin E$ such that, whenever $X \vdash e$ for some X and e, we have $\checkmark \notin X$;*
2. *$C \subseteq E \times E \times \{=, \neq\}$ is a relation defining a set of BoD and SoD constraints such that:*
   (a) *$\forall e \in E : (e, e, \neq) \notin C$;*
   (b) *$\forall e_1, e_2 \in E : (e_1, e_2, =) \in C \Rightarrow (e_2, e_1, =) \in C$;*
   (c) *$\forall e_1, e_2 \in E : (e_1, e_2, \neq) \in C \Rightarrow (e_2, e_1, \neq) \in C$;*
   (d) *$\forall e_1, e_2, e_3 \in E : (e_1, e_2, =) \in C \wedge (e_2, e_3, =) \in C \Rightarrow (e_1, e_3, =) \in C$;*
   (e) *$\forall e_1, e_2, e_3 \in E : (e_1, e_2, =) \in C \wedge (e_2, e_3, \neq) \in C \Rightarrow (e_1, e_3, \neq) \in C$;*
3. *$\rho : E \rightarrow R$ 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 $\mathcal{W} = \langle \langle E, \#, \vdash \rangle, C, \rho \rangle$, 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.

2.2.3 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 $\mathcal{A} = \langle \mathcal{S}, \mathcal{W} \rangle$ including an ARBAC system $\mathcal{S}$ 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*.

[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 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 $\perp$ 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.

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:*

$$\frac{e_1 \notin dom(H)}{H \models (e_1, e_2, \sim)} \qquad \frac{e_2 \notin dom(H)}{H \models (e_1, e_2, \sim)} \qquad \frac{H(e_1) \sim H(e_2)}{H \models (e_1, e_2, \sim)}$$

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 $\mathcal{A} = \langle \mathcal{S}, \mathcal{W} \rangle$ is defined by means of a labelled reduction relation on states $\sigma = \langle UR, H \rangle$, including a user-to-role assignment UR and a history H (see Table 2.2). Labels are drawn from the set of events $E_{\mathcal{W}}$ in the workflow, extended with a distinguished event $\circ$ 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 $x \notin dom(f)$, let $f[x \mapsto y]$ be the partial function g such that $dom(g) = dom(f) \cup \{x\}$, $g(x) = y$ and $\forall z \in dom(f) : g(z) = f(z)$.

Table 2.2 Reduction semantics ($\mathcal{P}, \mathcal{W} \rhd \sigma \xrightarrow{e} \sigma'$)

(R-ADMIN)

$$\frac{\mathcal{P} \rhd UR \rightsquigarrow UR'}{\mathcal{P}, \mathcal{W} \rhd \langle UR, H \rangle \xrightarrow{\circ} \langle UR', H \rangle}$$

(R-TASK)

$$\frac{\begin{array}{cc} \rho_{\mathcal{W}}(e) \in UR(u) & H[e \mapsto u] \models C_{\mathcal{W}} \\ \exists X \subseteq dom(H) : X \vdash_{\mathcal{W}} e & \forall e' \in dom(H) : (e', e) \notin \#_{\mathcal{W}} \end{array}}{\mathcal{P}, \mathcal{W} \rhd \langle UR, H \rangle \xrightarrow{e} \langle UR, H[e \mapsto u] \rangle}$$

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 2.2 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 $\perp$.

2.3 Security Against Collusion

2.3.1 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 $UR \downarrow_{U_c} = \{(u, r) \in UR \mid u \in U_c\}$ 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 $\mathcal{A} = \langle\langle \mathcal{P}, UR\rangle, \mathcal{W}\rangle$, a *trace* of $\mathcal{A}$ is a sequence of events $t = e_1, \ldots, e_n$ such that:

$$\exists \sigma_0, \ldots, \sigma_n : \sigma_0 = \langle UR, \perp\rangle \wedge \forall i \leq n : \mathcal{P}, \mathcal{W} \triangleright \sigma_{i-1} \xrightarrow{e_i} \sigma_i.$$

A trace t is *successful* iff $\checkmark$ occurs in t. We let $\mathbb{T}^{\checkmark}(\mathcal{A})$ 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 $\circ$ does not occur in t. We let $\mathbb{PT}^{\checkmark}(\mathcal{A})$ 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:*

$$\mathbb{T}^{\checkmark}(\mathcal{A}\downarrow_{U_c}) \neq \emptyset \Rightarrow \mathbb{PT}^{\checkmark}(\mathcal{A}\downarrow_{U_c}) \neq \emptyset.$$

2.3.2 Example

We now encode in our formalism the motivating example in Section 2.1.1 (Figure 2.1) and we show that it is not secure against collusion by $\{u_1, u_2\}$.

First, we define $\mathcal{W} = \langle\langle\{a, b, c, \checkmark\}, \vdash, \#\rangle, C, \rho\rangle$, where:

1. the enabling relation is:

$$\vdash = \{(\emptyset, a), (\{a\}, b), (\{a\}, c), (\{b\}, \checkmark), (\{c\}, \checkmark)\}$$

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. $\rho(a) = \mathsf{R}_1, \rho(b) = \mathsf{R}_2, \rho(c) = \mathsf{R}_3$ 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 $\mathcal{A} = \langle\langle\mathcal{P}, UR\rangle, \mathcal{W}\rangle$, where:

1. $\mathcal{P} = \langle\{(\mathsf{R}_2, \{\mathsf{R}_1\}, \emptyset, \mathsf{R}_3)\}, \emptyset\rangle$ 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. $UR = \{(u_1, \mathsf{R}_1), (u_2, \mathsf{R}_2)\}$ contains two users u_1, u_2 with roles $\mathsf{R}_1, \mathsf{R}_2$ 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 $\checkmark$ 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.

2.3.3 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 $t \in \mathbb{T}^{\checkmark}(\mathcal{A}{\downarrow_{U_c}})$: 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 $t' \in \mathbb{PT}^{\checkmark}(\mathcal{A}{\downarrow_{U_c}})$.

Formally, point (2) amounts to checking the *satisfiability* (or *consistency*) of a workflow, a problem which has received considerable attention in the past [72, 74]. In particular, Wang and Li proved that the workflow satisfiability problem is NP-hard in presence of SoD constraints [74]. 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).

2.4 Static Analysis

2.4.1 Overview

We propose to check security against collusion in ARBAC workflow systems through a reduction to the well-known *role reachability problem* for ARBAC [45, 69].

Given an ARBAC system $\mathcal{S} = \langle \mathcal{P}, UR \rangle$, a role r is said *reachable* in $\mathcal{S}$ 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:

$$(\forall i \leq n : \mathcal{P} \rhd UR_{i-1} \rightsquigarrow UR_i) \wedge r \in UR_n(u).$$

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 worklow 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 [31, 18, 41] 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.

2.4.2 Reduction to Role Reachability

Let $\mathcal{A} = \langle \mathcal{S}, \mathcal{W} \rangle$, for each task $e \in E_{\mathcal{W}}$ we introduce three fresh roles: $\mathsf{Done}[e]$, $\mathsf{Allowed}[e]$ and $\mathsf{Author}[e]$. Moreover, for each pair of tasks e_1, e_2 such that $(e_1, e_2, =) \in C_{\mathcal{W}}$, we introduce a fresh role $\mathsf{Eq}[e_1, e_2]$. 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 $\mathcal{W}$ into a set of can-assign rules $[\![\mathcal{W}]\!]$. The translation requires the generation of the set of the configurations of $\mathcal{E}_{\mathcal{W}}$ 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 [51].

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

We let $[\![\mathcal{W}]\!]$ be the smallest set of can-assign rules derived by the inference rules in Table 2.3. 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;

Table 2.3 Translation of a workflow $\mathcal{W}$ into a set of can-assign rules $[\![\mathcal{W}]\!]$

(T-ALLOWED)

$$\frac{X \in \mathbb{F}(\mathcal{E}_{\mathcal{W}}) \qquad e \in X \qquad \mathit{Done} = \{\mathsf{Done}[e'] \mid e' <_X e\} \\ \mathit{Eqs} = \{\mathsf{Eq}[e_1, e_2] \mid e_1 <_X e \wedge e_2 <_X e \wedge (e_1, e_2, =) \in C_{\mathcal{W}}\}}{(\mathsf{Super}, \mathit{Done} \cup \mathit{Eqs} \cup \{\mathsf{Super}\}, \{\mathsf{Allowed}[e'] \mid (e, e') \in \#_{\mathcal{W}}\}, \mathsf{Allowed}[e]) \in [\![\mathcal{W}]\!]}$$

(T-EQ)

$$\frac{(e_1, e_2, =) \in C_{\mathcal{W}}}{(\mathsf{Super}, \{\mathsf{Author}[e_1], \mathsf{Author}[e_2]\}, \emptyset, \mathsf{Eq}[e_1, e_2]) \in [\![\mathcal{W}]\!]}$$

(T-PROPEQ)

$$\frac{(e_1, e_2, =) \in C_{\mathcal{W}}}{(\mathsf{Eq}[e_1, e_2], \{\mathsf{Super}\}, \emptyset, \mathsf{Eq}[e_1, e_2]) \in [\![\mathcal{W}]\!]}$$

(T-AUTHOR)

$$\frac{X \in \mathbb{F}(\mathcal{E}_{\mathcal{W}}) \qquad e \in X \qquad \mathit{Neqs} = \{\mathsf{Author}[e'] \mid (e', e, \neq) \in C_{\mathcal{W}}\} \\ \mathit{Eqs} = \{\mathsf{Author}[e'] \mid e' <_X e \wedge (e', e, =) \in C_{\mathcal{W}}\}}{(\mathsf{Allowed}[e], \{\rho_{\mathcal{W}}(e)\} \cup \mathit{Eqs}, \mathit{Neqs} \cup \{\mathsf{Super}\}, \mathsf{Author}[e]) \in [\![\mathcal{W}]\!]}$$

(T-DONE)

$$\frac{e \in E_{\mathcal{W}}}{(\mathsf{Author}[e], \{\mathsf{Super}\}, \emptyset, \mathsf{Done}[e]) \in [\![\mathcal{W}]\!]}$$

- rule (T-ALLOWED): to assign role $\mathsf{Allowed}[e]$, 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 $\mathsf{Allowed}[e]$ 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 $\mathsf{Allowed}[e]$ has been assigned to the super user, it is possible to attempt the assignment of role $\mathsf{Author}[e]$. 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 ρ; 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 $\mathsf{Author}[e]$, 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 $\mathsf{Author}[e]$ has been assigned to some user, it is possible to assign role $\mathsf{Done}[e]$ to the super user. This role assignment tracks that it was indeed possible to perform task e. Role $\mathsf{Done}[e]$ may be required to enable the assignment of role $\mathsf{Allowed}[e']$ for some event e' which is a successor of e;

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

Finally, the translation is extended so as to map an ARBAC workflow system $\mathcal{A} = \langle\langle\langle CA, CR\rangle, UR\rangle, \mathcal{W}\rangle$ into a corresponding ARBAC system $[\![\mathcal{A}]\!]$ as follows:

$$[\![\mathcal{A}]\!] = \langle\langle CA^- \cup [\![\mathcal{W}]\!], CR\rangle, UR \cup \{(u_0, \mathsf{Super})\}\rangle,$$

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

2.4.3 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 $\mathbb{T}^{\checkmark}(\mathcal{A}) \neq \emptyset$, the role* $\mathsf{Done}[\checkmark]$ *is reachable in $[\![\mathcal{A}]\!]$.*

Proof. See section 3.1. □

The soundness theorem ensures that, if role $\mathsf{Done}[\checkmark]$ is not reachable in $[\![\mathcal{A}]\!]$, 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 $\mathsf{Done}[\checkmark]$ in $[\![\mathcal{A}{\downarrow_{U_c}}]\!]$ 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* $\mathsf{Done}[\checkmark]$ *is reachable in* $[\![\mathcal{A}]\!]$*, we have that* $\mathbb{T}^{\checkmark}(\mathcal{A}) \neq \emptyset$*.*

Proof. See section 3.2. □

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 [72, 74], 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 $\mathsf{Done}[\checkmark]$ is reachable in $[\![\mathcal{A}{\downarrow_{U_c}}]\!]$ 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 $\mathsf{Author}[e]$ for some task 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 $\mathsf{Author}[e]$ 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 $\mathsf{Author}[e]$ to multiple users does not introduce "false attacks". Notably, if no BoD constraint is put in place in the workflow system, roles like $\mathsf{Author}[e]$ 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 $\mathsf{Author}[e]$ 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 $\mathsf{Author}[e]$ is assigned at most once.

It should not be surprising that the encoding we propose can also be used to solve the classic workflow satisfiability problem [72, 74] in the case of ARBAC workflow systems without BoD constraints. Given $\mathcal{A} = \langle\langle\mathcal{P}, UR\rangle, \mathcal{W}\rangle$, let:

$$[\![\mathcal{A}]\!]^* = \langle\langle[\![\mathcal{W}]\!], \emptyset\rangle, UR \cup \{(u_0, \mathsf{Super})\}\rangle,$$

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

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

Proof. Is enough to observe that $\mathbb{PT}^{\checkmark}(\langle\langle\mathcal{P}, UR\rangle, \mathcal{W}\rangle) = \mathbb{T}^{\checkmark}(\langle\langle\mathcal{P}_\perp, UR\rangle, \mathcal{W}\rangle)$, where $\mathcal{P}_\perp = \langle\emptyset, \emptyset\rangle$ 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 [74], which we consider in this work. The encoding $[\![\cdot]\!]^*$ generates an ARBAC system without can-revoke rules, so the corresponding role reachability problem is NP-complete [45]. 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 2.4 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 $\mathsf{Done}[\checkmark]$ is not reachable in $[\![\mathcal{A}{\downarrow}_{U_c}]\!]$, we can immediately prove security and ignore $[\![\mathcal{A}{\downarrow}_{U_c}]\!]^*$. One case is missing from the table, since it is not possible that $\mathsf{Done}[\checkmark]$ is reachable in $[\![\mathcal{A}{\downarrow}_{U_c}]\!]^*$, but not in $[\![\mathcal{A}{\downarrow}_{U_c}]\!]$.

Table 2.4 Checking security of $\mathcal{A}$ against collusion by U_c

BoD	Reach in $[\![\mathcal{A}{\downarrow}_{U_c}]\!]$	Reach in $[\![\mathcal{A}{\downarrow}_{U_c}]\!]^*$	Secure
no	no	no	yes
no	yes	no	no
no	yes	yes	yes
yes	no	no	yes
yes	yes	no	?
yes	yes	yes	?

2.4.4 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 [45, 33, 31].

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.

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 [31]. We say that a role is *purely administrative* iff 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.

Preprocessing

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

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

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 $\mathsf{Done}[e']$ can be assigned if and only if role $\mathsf{Author}[e']$ is first assigned.

Moreover, as part of the preprocessing step, all roles of the form $\mathsf{Allowed}[e]$ 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 $\mathsf{Done}[\checkmark]$, 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 $\mathsf{Done}[\checkmark]$ is not affected by this change, observe that a role like $\mathsf{Allowed}[e]$ is only included in the negative preconditions of a can-assign rule generated by the encoding in Table 2.3 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 $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ can be assigned, since only one between $\mathsf{Allowed}[e_1]$ and $\mathsf{Allowed}[e_2]$ can be given to the super user. However, assume that both $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ 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. $\mathsf{Done}[e_1]$ and $\mathsf{Done}[e_2]$ get both assigned;
2. $\mathsf{Eq}[e_1, e_2]$ gets assigned (if $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ are given to the same user).

Both these cases do not affect the reachability of $\mathsf{Done}[\checkmark]$, 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 $\mathsf{Done}[e_1]$ and $\mathsf{Done}[e_2]$ never occur together in the positive preconditions of a can-assign rule, and similarly $\mathsf{Eq}[e_1, e_2]$ 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.

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 $ca' = (r', R'_p \cup \{r_t\}, R'_n, r'_t)$ with $R_p \subseteq R'_p$, $R_n \subseteq R'_n$ 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 $R_p \subseteq R'_p$, $R_n \subseteq R'_n$ and there exists $r'' \in R'_p \cup \{r'\}$ such that $r \preceq r''$, then remove 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 [33], 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 3. *Role* $\mathsf{Done}[\checkmark]$ *is reachable in* $[\![\mathcal{A}]\!]$ *if and only if it is reachable after pruning* $[\![\mathcal{A}]\!]$.

Proof. See section 3.3. □

2.5 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.

2.5.1 Exclusive Choice

Consider the workflow described in the motivating example in Section 2.1.1 (Figure 2.1), its translation into ARBAC is given in Table 2.5.

Table 2.5 Translation of the workflow in Figure 2.1

$$(\mathsf{Super}, \{\mathsf{Super}\}, \emptyset, \mathsf{Allowed}[a])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Super}\}, \{\mathsf{Allowed}[c]\}, \mathsf{Allowed}[b])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Super}\}, \{\mathsf{Allowed}[b]\}, \mathsf{Allowed}[c])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Done}[b], \mathsf{Eq}[a,b], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[\checkmark])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Done}[c], \mathsf{Eq}[a,c], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[\checkmark])$$

$$(\mathsf{Super}, \{\mathsf{Author}[a], \mathsf{Author}[b]\}, \emptyset, \mathsf{Eq}[a,b])$$

$$(\mathsf{Super}, \{\mathsf{Author}[a], \mathsf{Author}[c]\}, \emptyset, \mathsf{Eq}[a,c]) \qquad (\mathsf{Eq}[a,b], \{\mathsf{Super}\}, \emptyset, \mathsf{Eq}[a,b])$$

$$(\mathsf{Eq}[a,c], \{\mathsf{Super}\}, \emptyset, \mathsf{Eq}[a,c]) \qquad (\mathsf{Allowed}[a], \{\mathsf{R}_1\}, \{\mathsf{Super}\}, \mathsf{Author}[a])$$

$$(\mathsf{Allowed}[b], \{\mathsf{R}_2, \mathsf{Author}[a]\}, \{\mathsf{Super}\}, \mathsf{Author}[b])$$

$$(\mathsf{Allowed}[c], \{\mathsf{R}_3, \mathsf{Author}[a]\}, \{\mathsf{Super}\}, \mathsf{Author}[c])$$

$$(\mathsf{Allowed}[\checkmark], \emptyset, \{\mathsf{Super}\}, \mathsf{Author}[\checkmark])$$

Notice that roles $\mathsf{Allowed}[b]$ and $\mathsf{Allowed}[c]$ are mutually exclusive, since events b and c are conflicting and this implies that only one role between $\mathsf{Author}[b]$ and $\mathsf{Author}[c]$ can be assigned. Correspondingly, there are two ways to introduce role $\mathsf{Allowed}[\checkmark]$: indeed, recall that role $\mathsf{Done}[b]$ / $\mathsf{Done}[c]$ can only be assigned by a user with role $\mathsf{Author}[b]$ / $\mathsf{Author}[c]$. Moreover, notice that both $\mathsf{Author}[b]$ and $\mathsf{Author}[c]$ can only be assigned to a user who is assigned $\mathsf{Author}[a]$, 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.

2.5.2 Sequential Execution

Consider the workflow in Figure 2.2, 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 2.6.

$$\begin{tikzcd}
\cdot \arrow[r] & a_{\{\mathsf{R}_1\}} \arrow[r] \arrow[rr, dash, dashed, bend left, "\neq"] & b_{\emptyset} \arrow[r] & c_{\{\mathsf{R}_2\}} \arrow[r] & \checkmark
\end{tikzcd}$$

FIGURE 2.2: Sequential execution with separation-of-duty

Table 2.6 Translation of the workflow in Figure 2.2

$$(\mathsf{Super}, \{\mathsf{Super}\}, \emptyset, \mathsf{Allowed}[a]) \qquad (\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[b])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Done}[b], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[c])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Done}[b], \mathsf{Done}[c], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[\checkmark])$$

$$(\mathsf{Allowed}[a], \{\mathsf{R}_1\}, \{\mathsf{Author}[c], \mathsf{Super}\}, \mathsf{Author}[a])$$

$$(\mathsf{Allowed}[b], \emptyset, \{\mathsf{Super}\}, \mathsf{Author}[b])$$

$$(\mathsf{Allowed}[c], \{\mathsf{R}_2\}, \{\mathsf{Author}[a], \mathsf{Super}\}, \mathsf{Author}[c])$$

$$(\mathsf{Allowed}[\checkmark], \emptyset, \{\mathsf{Super}\}, \mathsf{Author}[\checkmark])$$

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

2.5.3 Parallel Execution

Consider the workflow in Figure 2.3, 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 $\emptyset \vdash a$, $\emptyset \vdash b$ and $\{a, b\} \vdash \checkmark$). Moreover, assume there exists a BoD constraint between a and b.

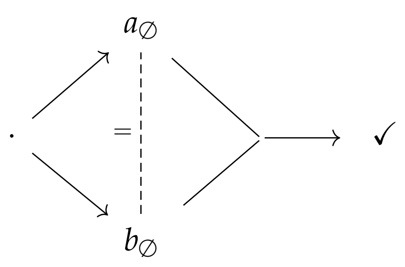


FIGURE 2.3: Parallel execution with binding-of-duty

Table 2.7 Translation of the workflow in Figure 2.3

$$(\mathsf{Super}, \{\mathsf{Super}\}, \emptyset, \mathsf{Allowed}[a]) \qquad (\mathsf{Super}, \{\mathsf{Super}\}, \emptyset, \mathsf{Allowed}[b])$$

$$(\mathsf{Super}, \{\mathsf{Done}[a], \mathsf{Done}[b], \mathsf{Eq}[a,b], \mathsf{Super}\}, \emptyset, \mathsf{Allowed}[\checkmark])$$

$$(\mathsf{Super}, \{\mathsf{Author}[a], \mathsf{Author}[b]\}, \emptyset, \mathsf{Eq}[a,b]) \qquad (\mathsf{Eq}[a,b], \{\mathsf{Super}\}, \emptyset, \mathsf{Eq}[a,b])$$

$$(\mathsf{Allowed}[a], \emptyset, \{\mathsf{Super}\}, \mathsf{Author}[a]) \qquad (\mathsf{Allowed}[b], \emptyset, \{\mathsf{Super}\}, \mathsf{Author}[b])$$

$$(\mathsf{Allowed}[\checkmark], \{\mathsf{Super}\}, \emptyset, \mathsf{Author}[\checkmark])$$

The translation of the workflow into ARBAC is shown in Table 2.7. 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 $\mathsf{Allowed}[a]$ and $\mathsf{Allowed}[b]$, as well as $\mathsf{Author}[a]$ and $\mathsf{Author}[b]$, can be liberally assigned without checking the BoD constraint between a and b. However, $\mathsf{Allowed}[\checkmark]$ can only be introduced whenever $\mathsf{Eq}[a,b]$ is assigned to the super user, which is only possible when there exists a user who is assigned both $\mathsf{Author}[a]$ and $\mathsf{Author}[b]$, i.e., when the BoD constraint between a and b is satisfied.

2.6 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 2.4.4.

The generated role reachability queries for role Done[✓] are then discharged by an existing state-of-the-art tool, VAC [31].

2.6.1 Implementing the Analysis

WARBAC reduces security against collusion to role reachability by implementing the checks summarised in Table 2.4 (in Section 2.4.3). Since role reachability may be costly to check, WARBAC exploits a set of analysis backends supported by VAC to make the security analysis more efficient:

1. INTERPROC [42]: 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 [32];

2. CBMC [50]: 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 [31];

3. NuSMV [21]: computationally expensive sound and complete analysis. A positive/negative answer by NuSMV proves role reachability/unreachability [31].

When testing role reachability, WARBAC 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, WARBAC 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 2.4.4, and 2) run the pruning algorithm internally implemented in VAC, until no further simplification is possible.

2.6.2 Experiments

We created a set of examples to test WARBAC, which we make available online [19]. All the examples refer to a medical setting: specifically, we extended an existing ARBAC system (the Hospital case study in [31]) with the specification of a number of different workflows. Most of the workflows we developed are larger, more complicated variants of the examples shown in

Section 2.5, 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.

Synthetic Examples

Table 2.8 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 WARBAC.

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 WARBAC to only run INTERPROC. As expected, the over-approximated analysis implemented in INTERPROC is very

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

TABLE 2.8: Experimental results

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, WARBAC 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 WARBAC is unable to prove security, since our analysis is not proved complete for workflows using BoD constraints.

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;

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

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*);

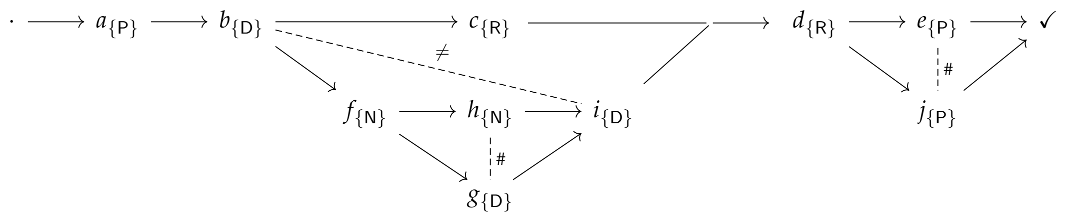
g) urgent case: a doctor treats the patient;

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 2.4, where we use the letters P, D, N to represent roles Patient, Doctor, Nurse respectively.



$$\begin{tikzcd}
\cdot \arrow[r] & a_{\{\mathsf{P}\}} \arrow[r] & b_{\{\mathsf{D}\}} \arrow[rr] \arrow[dr] \arrow[drrr, dashed, no head, "\neq"] & & c_{\{\mathsf{R}\}} \arrow[rr] & & d_{\{\mathsf{R}\}} \arrow[r] \arrow[dr] & e_{\{\mathsf{P}\}} \arrow[r] \arrow[d, dashed, no head, "\#"] & \checkmark \\
& & & f_{\{\mathsf{N}\}} \arrow[r] \arrow[dr] & h_{\{\mathsf{N}\}} \arrow[r] \arrow[d, dashed, no head, "\#"] & i_{\{\mathsf{D}\}} \arrow[ur] & & j_{\{\mathsf{P}\}} \arrow[ur] & \\
& & & & g_{\{\mathsf{D}\}} \arrow[ur] & & & &
\end{tikzcd}$$

FIGURE 2.4: Case study: a first aid procedure

2.7 Related Work

The paper by Wang *et al.* on the security of delegation in access control systems [75] 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 work is an adaptation of the security property in [75] to the case of ARBAC administrative actions, there are several notable differences between this work and [75]. First, static analysis is only briefly mentioned in [75] 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 [69]; indeed, we observed the need to aggressively optimise our encoding into ARBAC to obtain an efficient static analysis. Finally, the formal model in this work is quite different and more general than the one in [75]: 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.

Bertolisi *et al.* introduced an approach for the synthesis of run-time monitors to enforce separation and binding of duty security constraints to workflows [11]. Their work however focuses on runtime checks, so it could be seen as a complement of this work as we focus only on static ones. Moreover their work focuses mainly on enforcement of BoD and SoD constraints and does not provide any enforcement on security against collusion.

The satisfiability (or consistency) of workflows is a classic problem in computer security [72, 74, 78]. 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 [74].

Crampton and Khambhammettu proposed algorithms to check the satisfiability of workflows supporting delegation operations [24]. 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 work, 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 collusions are the main concern for system administrators, the present work proposes a viable solution; if instead it is better to ensure workflow termination at the price of permitting collusion, the approach in [24] 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 [9]. 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.

2.8 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 work for the sake of simplicity, most notably because the interaction between loops and BoD/SoD constraints is quite subtle [9]. 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 st wronger completeness result for our static analysis.

Chapter 3

Proofs of Chapter 2

Here we detail the formal proofs of theorems 1, 2 and 3 introduced in chapter 2.

The results in this chapter assume a disjointness condition between the set of roles occurring in the ARBAC policy underlying the workflow system to analyse and the set of roles introduced by our formal encoding. This disjointness condition can always be enforced by a suitable renaming of the roles in the policy and we assume that such a renaming has been done. We say that a role r is *regular* iff it is not introduced by the encoding and, when writing $\mathcal{P}$ or using the unfolded notation $\langle CA, CR \rangle$ in the next results, we tacitly assume that only regular roles occur in $\mathcal{P}, CA, CR$. Similarly, we assume that only regular roles occur in the initial user-to-role assignment of the ARBAC systems we consider.

We write $UR \approx_{reg} UR'$ if and only if, for all users u and regular roles r, $r \in UR(u)$ iff $r \in UR'(u)$. We write $UR \approx_{enc} UR'$ if and only if, for all users u and non-regular roles r, $r \in UR(u)$ iff $r \in UR'(u)$.

3.1 Proof of Theorem 1

Lemma 2. *If $UR_1 \approx_{reg} UR_2$ and $\mathcal{P} \rhd UR_1 \rightsquigarrow UR_1'$, then there exists UR_2' such that $\mathcal{P} \rhd UR_2 \rightsquigarrow UR_2'$ with $UR_1' \approx_{reg} UR_2'$ and $UR_2 \approx_{enc} UR_2'$.*

Proof. By a case analysis on the rule applied in the reduction step. Observe that $\mathcal{P} \rhd UR_1 \rightsquigarrow UR_1'$ can only assign/revoke a regular role r to/from a user u and the same can-assign/can-revoke rule can be used to derive $\mathcal{P} \rhd UR_2 \rightsquigarrow UR_2'$ by assigning/revoking r to/from u: this implies $UR_1' \approx_{reg} UR_2'$ and $UR_2 \approx_{enc} UR_2'$. □

Definition 11 (Correspondence)**.** *We write $\langle UR, H \rangle \simeq_C UR'$ if and only if all the following conditions hold:*

a. $H \models C$

b. $UR \approx_{reg} UR'$

c. *for each event e and user u,* $H(e) = u$ *if and only if* $\mathsf{Author}[e] \in UR'(u)$

d. *there exists a distinguished user* u_0*, the super user, such that* $\mathsf{Super} \in UR'(u_0)$*. Moreover,* $\{e \mid \mathsf{Allowed}[e] \in UR'(u_0)\} = dom(H)$*.*

Lemma 3 (Sound Simulation)**.** *For any workflow* $\mathcal{W} = \langle \mathcal{E}, C, \rho \rangle$*, we have that* $\langle UR_1, H \rangle \simeq_C UR_2$ *and* $\langle CA, CR \rangle, \mathcal{W} \rhd \langle UR_1, H \rangle \xrightarrow{e} \langle UR_1', H' \rangle$ *imply* $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \rhd UR_2 \rightsquigarrow^* UR_2'$ *with* $\langle UR_1', H' \rangle \simeq_C UR_2'$*.*

Proof. By a case analysis on the rule applied in the reduction step.

In case (R-ADMIN), we have $\langle CA, CR \rangle \rhd UR_1 \rightsquigarrow UR_1'$ and $H' = H$. By Lemma 2, $\langle CA, CR \rangle \rhd UR_2 \rightsquigarrow UR_2'$ with $UR_1' \approx_{reg} UR_2'$ and $UR_2 \approx_{enc} UR_2'$. Since reachability is preserved when more can-assign rules are added to an ARBAC policy, we also have $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \rhd UR_2 \rightsquigarrow UR_2'$. We then get $\langle UR_1', H \rangle \simeq_C UR_2'$ by $UR_1' \approx_{reg} UR_2'$ and the hypothesis $\langle UR_1, H \rangle \simeq_C UR_2$. Notice that the observation $UR_2 \approx_{enc} UR_2'$ is needed to conclude.

In case (R-TASK), we have $UR_1' = UR_1$ and $H' = H[e \mapsto u]$ and:

1. $\rho(e) \in UR_1(u)$

2. $H[e \mapsto u] \models C$

3. $\exists X \subseteq dom(H) : X \vdash e$

4. $\forall e' \in dom(H) : (e', e) \notin \#$

We first show that it is possible to assign role $\mathsf{Allowed}[e]$ to some user. Since $H[e \mapsto u]$ is a history, we know that $e \notin dom(H)$ and $Y = dom(H) \cup \{e\}$ is a configuration. By (T-ALLOWED), this implies that $[\![\mathcal{W}]\!]$ contains the following can-assign rule:

$$\frac{\begin{array}{c} Done = \{\mathsf{Done}[e'] \mid e' <_Y e\} \\ Eqs = \{\mathsf{Eq}[e_1, e_2] \mid e_1 <_Y e \wedge e_2 <_Y e \wedge (e_1, e_2, =) \in C\} \\ Confl = \{\mathsf{Allowed}[e'] \mid (e, e') \in \#_{\mathcal{W}}\} \end{array}}{(\mathsf{Super}, Done \cup Eqs \cup \{\mathsf{Super}\}, Confl, \mathsf{Allowed}[e]) \in [\![\mathcal{W}]\!]}$$

Since there exists a super user u_0 by definition of corresponding states, to trigger the rule we only need to show that the positive and the negative preconditions can be satisfied by him. We start by observing that, since also

$dom(H)$ is a configuration by definition of history, we know that:

$$\forall e' \in dom(H).\exists e_1, \ldots, e_n \in dom(H) : e_n = e' \wedge \forall i \leq n.\exists Y_i \subseteq \{e_1, \ldots, e_{i-1}\} : Y_i \vdash e_i.$$

By point (3) there exists $X \subseteq dom(H)$ such that $X \vdash e$, hence Y is a configuration such that, for any event $e' \in Y$, we have that $e' <_Y e$ implies $e' \neq e$. To prove it, assume by contradiction that $e' = e$, then $e' <_Y e$ implies the existence of $e'_1, \ldots, e'_m \in Y$ and $X_1, \ldots, X_{m+1} \subseteq Y$ such that:

- $X_1 \vdash e'_1, \ldots, X_m \vdash e'_m$ and $X_{m+1} \vdash e$
- $e \in X_1$ and $\forall j \leq m : e'_j \in X_{j+1}$

Let us focus now on the event $e'_1 \in Y$ such that $X_1 \vdash e'_1$ with $e \in X_1$, we have two cases. If $e'_1 = e$, then we also have $X \vdash e'_1$ with $X \subseteq dom(H)$; if $e'_1 \neq e$, then $e'_1 \in dom(H)$, hence we also have $Y_j \vdash e'_1$ for some $Y_j \subseteq dom(H)$. Notably, both X and Y_j are different than X_1, since they do not include e. Given that Y is conflict-free by definition of configuration and $X, X_1, Y_j \subseteq Y$, this contradicts the stability axiom in the definition of workflow. Hence, for any $e' <_Y e$, we have $e' \in dom(H)$, which is crucial for the continuation of the proof.

We can finally show that the positive and the negative preconditions of the rule above are satisfied by the super user u_0, possibly after the application of additional can-assign rules:

- *Done*: pick any $e' <_Y e$, we showed that $e' \in dom(H)$. By definition of corresponding states, there exists a user u' such that $\mathsf{Author}[e'] \in UR_2(u')$, hence we can apply rule (T-DONE):

$$(\mathsf{Author}[e'], \{\mathsf{Super}\}, \emptyset, \mathsf{Done}[e'])$$

  to assign role $\mathsf{Done}[e']$ to the super user u_0;

- *Eqs*: pick any e_1, e_2 such that $e_1 <_Y e$ and $e_2 <_Y e$, we showed that $e_1, e_2 \in dom(H)$. Assume that $(e_1, e_2, =) \in C$. Since $H[e \mapsto u] \models C$ implies $H \models C$, we know that there exists a user u' such that $H(e_1) = H(e_2) = u'$. By definition of corresponding states, we have $\{\mathsf{Author}[e_1], \mathsf{Author}[e_2]\} \subseteq UR_2(u')$ and we can apply rule (T-EQ):

$$(\mathsf{Super}, \{\mathsf{Author}[e_1], \mathsf{Author}[e_2]\}, \emptyset, \mathsf{Eq}[e_1, e_2])$$

to assign $\mathsf{Eq}[e_1, e_2]$ to u'. By rule (T-PropEq):

$$(\mathsf{Eq}[e_1, e_2], \{\mathsf{Super}\}, \emptyset, \mathsf{Eq}[e_1, e_2])$$

we can then assign role $\mathsf{Eq}[e_1, e_2]$ also to the super user u_0;

- *Confl*: since $Y = dom(H) \cup \{e\}$ is a configuration, we know that for all $e' \in dom(H)$ we have $(e, e') \notin \#_{\mathcal{W}}$. We then observe that by definition of corresponding states the super user u_0 satisfies the following property:

  $$\{e \mid \mathsf{Allowed}[e] \in UR_2(u_0)\} = dom(H).$$

  Since $[\![\mathcal{W}]\!]$ contains rule (T-Allowed):

  $$(\mathsf{Super}, \{\mathsf{Super}\}, \{\mathsf{Allowed}[e'] \mid (e, e') \in \#_{\mathcal{W}}\}, \mathsf{Allowed}[e])$$

  it is possible to assign role $\mathsf{Allowed}[e]$ to u_0.

We then prove that there exists a can-assign rule introducing role $\mathsf{Author}[e]$ which can be fired. Specifically, by (T-Author), we know that $[\![\mathcal{W}]\!]$ contains the following can-assign rule:

$$\frac{\begin{array}{c} Eqs = \{\mathsf{Author}[e'] \mid e' <_Y e \wedge (e', e, =) \in C\} \\ Neqs = \{\mathsf{Author}[e'] \mid (e', e, \neq) \in C\} \end{array}}{(\mathsf{Allowed}[e], \rho(e) \cup Eqs, Neqs \cup \{\mathsf{Super}\}, \mathsf{Author}[e])}$$

Since we already proved that $\mathsf{Allowed}[e]$ can be assigned to u_0, to trigger the rule we just need to show that the user u who performed the task e satisfies the positive and the negative preconditions of the rule, possibly after the application of additional can-assign rules:

- $\rho(e)$: since $\rho(e) \in UR_1(u)$ by point (1) and $UR_1 \approx_{reg} UR_2$ by definition of corresponding states, we have $\rho(e) \in UR_2(u)$;

- *Eqs*: pick any $e' <_Y e$, we showed that $e' \in dom(H)$. Since $H[e \mapsto u] \models C$ by point (2), we know that $(e', e, =) \in C$ implies $H(e') = u$. Hence, $\mathsf{Author}[e'] \in UR_2(u)$ by definition of corresponding states;

- *Neqs*: pick any e' such that $(e', e, \neq) \in C$, we have two sub-cases. If $e' \notin dom(H)$, we know that no user is assigned role $\mathsf{Author}[e']$ in UR_2 by definition of corresponding states and we are done. Otherwise, if

$H(e') = u'$ for some user u', we observe that $H[e \mapsto u] \models C$ by point (2), hence $u' \neq u$. Since $\mathsf{Author}[e']$ can be assigned only to one user by definition of corresponding states, we have that $\mathsf{Author}[e'] \notin UR_2(u)$;

- Super: by observing that $u \neq u_0$, since u_0 is a distinguished user obtained by extending the set of users U.

Combining all the observations above, we get $UR_2 \rightsquigarrow^* UR_2' \cup \{(u, \mathsf{Author}[e])\}$ for some UR_2'. Showing $\langle UR_1, H[e \mapsto u] \rangle \simeq_C UR_2' \cup \{(u, \mathsf{Author}[e])\}$ amounts to a simple syntactic check on the detailed construction. □

We are now ready to prove Theorem 1:

Proof. Let $\mathcal{A} = \langle \langle \mathcal{P}, UR \rangle, \mathcal{W} \rangle$ and assume that there exists a sequence of events $t = e_1, \ldots, e_n$ including ✓ such that:

$$\exists \sigma_0, \ldots, \sigma_n : \sigma_0 = \langle UR, \perp \rangle \wedge \forall i \leq n : \mathcal{P}, \mathcal{W} \triangleright \sigma_{i-1} \xrightarrow{e_i} \sigma_i.$$

Let $\mathcal{P} = \langle CA, CR \rangle$, we have:

$$[\![\mathcal{A}]\!] = \langle \langle CA \cup [\![\mathcal{W}]\!], CR \rangle, UR \cup \{(u_0, \mathsf{Super})\} \rangle,$$

where u_0 is a distinguished user such that $UR(u_0) = \emptyset$, obtained by extending the set of users U. Since $\langle UR, \perp \rangle \simeq_{C_{\mathcal{W}}} UR \cup \{(u_0, \mathsf{Super})\}$, by multiple applications of Lemma 3 there exists UR' such that:

$$\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \triangleright UR \cup \{(u_0, \mathsf{Super})\} \rightsquigarrow^* UR',$$

and $\sigma_n \simeq_{C_{\mathcal{W}}} UR'$. Let $\sigma_n = \langle UR_n, H \rangle$: since ✓ occurs in t, there exists u such that $H(✓) = u$, which implies that $\mathsf{Author}[✓] \in UR'(u)$ by definition of $\sigma_n \simeq_{C_{\mathcal{W}}} UR'$. The conclusion then follows by using rule (T-DONE) from $[\![\mathcal{W}]\!]$. □

3.2 Proof of Theorem 2

Given a user-to-role assignment UR, let:

- $\hat{H}(UR) = \{(e, u) \mid \mathsf{Author}[e] \in UR(u)\}$
- $Evt(UR) = \{e \mid \exists u : (e, u) \in \hat{H}(UR)\}$

We write $\mathcal{E} \vdash_{\diamond} UR$ if and only if both these conditions are satisfied:

- $\forall e, u_1, u_2 : (e, u_1) \in \hat{H}(UR) \wedge (e, u_2) \in \hat{H}(UR) \Rightarrow u_1 = u_2$
- $Evt(UR) \in \mathbb{F}(\mathcal{E})$

Observe that, if $\mathcal{W} = \langle \mathcal{E}, C, \rho \rangle$ and $\mathcal{E} \vdash_\diamond UR$, then $\hat{H}(UR)$ is a history for $\mathcal{W}$, hence we can reuse the notation we introduced for histories.

Definition 12 (Well-formedness)**.** *A user-to-role assignment UR is* well-formed *for $\mathcal{W}$ if and only if both these conditions are satisfied:*

1. $\mathcal{E}_\mathcal{W} \vdash_\diamond UR$
2. $\hat{H}(UR) \models C_\mathcal{W}$

Lemma 4. *Let UR be a user-to-role assignment such that* Super *is only assigned to a distinguished user u_0. For any workflow $\mathcal{W}$, if $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \rhd UR \rightsquigarrow^* UR'$ and $Evt(UR) \in \mathbb{F}(\mathcal{E}_\mathcal{W})$, then $Evt(UR') \in \mathbb{F}(\mathcal{E}_\mathcal{W})$.*

Proof. Let $\mathcal{E}_\mathcal{W} = \langle E, \#, \vdash \rangle$, we proceed by induction on the number of reduction steps. If no reduction takes place, the conclusion is immediate by the hypothesis $Evt(UR) \in \mathbb{F}(\mathcal{E}_\mathcal{W})$. Otherwise, assume there exists UR'' such that $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \rhd UR \rightsquigarrow^* UR''$ and $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \rhd UR'' \rightsquigarrow UR'$. By induction hypothesis we have $Evt(UR'') \in \mathbb{F}(\mathcal{E}_\mathcal{W})$, we show that $Evt(UR') \in \mathbb{F}(\mathcal{E}_\mathcal{W})$. Notice that also in UR'' there exists a unique user u_0 such that $\mathsf{Super} \in UR''(u_0)$, since no ARBAC rule assigns or revokes Super. Notably, this also means that this user never changes upon reduction.

The only case which can break the property when moving from UR'' to UR' is the application of a can-assign rule giving a role $\mathsf{Author}[e]$ to some user u and:

1. either there exists $e_i \in Evt(UR'')$ such that $(e, e_i) \in \#$
2. or there is no $X \subseteq Evt(UR'')$ such that $X \vdash e$

By rule (T-AUTHOR), the can-assign rule has the following format for some configuration $Y \in \mathbb{F}(\mathcal{E})$ and $e \in Y$:

(T-AUTHOR)
$$\frac{\begin{array}{c} Neqs = \{\mathsf{Author}[e'] \mid (e', e, \neq) \in C_\mathcal{W}\} \\ Eqs = \{\mathsf{Author}[e'] \mid e' <_Y e \wedge (e', e, =) \in C_\mathcal{W}\} \end{array}}{(\mathsf{Allowed}[e], \rho_\mathcal{W}(e) \cup Eqs, Neqs \cup \{\mathsf{Super}\}, \mathsf{Author}[e])}$$

We show that both the possibilities above cannot happen:

1. Since the can-assign rule is fired, there exists a user u' s.t. $\mathsf{Allowed}[e] \in UR''(u')$. By rule (T-ALLOWED), this user is unique and coincides with u_0, since he must be granted the Super role, which is only assigned to u_0. By definition of $Evt(UR'')$, for each event $e_i \in Evt(UR'')$ there exists a user u_i such that $\mathsf{Author}[e_i] \in UR''(u_i)$. By the definition of $[\![\mathcal{W}]\!]$, $\mathsf{Author}[e_i]$ can only be assigned if $\mathsf{Allowed}[e_i]$ is first assigned to the super user. Since no role of the form $\mathsf{Allowed}[e_i]$ is revocable, this implies that $\{\mathsf{Allowed}[e_i] \mid e_i \in Evt(UR'')\} \subseteq UR''(u_0)$. We now observe that the rule which grants $\mathsf{Allowed}[e]$ to u_0 requires that $\{\mathsf{Allowed}[e'] \mid (e, e') \in \#\} \cap UR''(u_0) = \emptyset$. Since we know that $\{\mathsf{Allowed}[e_i] \mid e_i \in Evt(UR'')\} \subseteq UR''(u_0)$, we conclude that for each $e_i \in Evt(UR'')$ we must have that $(e, e_i) \notin \#$;

2. Since the can-assign rule is fired, there exists a user u' s.t. $\mathsf{Allowed}[e] \in UR''(u')$. By rule (T-ALLOWED) and observing that no role of the form $\mathsf{Done}[e_i]$ is revocable, there exists a configuration $Z \in \mathbb{F}(\mathcal{E}_{\mathcal{W}})$ such that $e \in Z$ and $\{\mathsf{Done}[e_i] \mid e_i <_Z e\} \subseteq UR''(u')$. By the definition of $[\![\mathcal{W}]\!]$, $\mathsf{Done}[e_i]$ can only be assigned if $\mathsf{Author}[e_i]$ is first assigned to some user. Since no role of the form $\mathsf{Author}[e_i]$ is revocable, for each event $e_i <_Z e$ there must exist a user u_i such that $\mathsf{Author}[e_i] \in UR''(u_i)$. We then know that $\{e_i \mid e_i <_Z e\} \subseteq Evt(UR'')$ by definition of the latter. Since Z is a configuration and $e \in Z$, there exists $Z' \subseteq Z$ such that $Z' \vdash e$. Given that we have $Z' \subseteq \{e_i \mid e_i <_Z e\}$ by definition of $<_Z$ and $\{e_i \mid e_i <_Z e\} \subseteq Evt(UR'')$, we can conclude.

□

Lemma 5 (Well-formed Traces). *Let $\mathcal{A} = \langle\langle \mathcal{P}, UR\rangle, \mathcal{W}\rangle$ be an ARBAC workflow system without binding-of-duty constraints. If the role* $\mathsf{Done}[\checkmark]$ *is reachable in $[\![\mathcal{A}]\!]$ and UR is well-formed for $\mathcal{W}$, then* $\mathsf{Done}[\checkmark]$ *is reachable using only user-to-role assignments which are well-formed for $\mathcal{W}$.*

Proof. Let $\mathcal{P} = \langle CA, CR\rangle$ and $\mathcal{W} = \langle \mathcal{E}, C, \rho\rangle$. Since $\mathsf{Done}[\checkmark]$ is reachable in $[\![\mathcal{A}]\!]$, there exist a user u and a sequence of user-to-role assignments $UR_0, \ldots,$ UR_n such that:

- $UR_0 = UR \cup \{(u_0, \mathsf{Super})\}$, where u_0 is a distinguished user s.t. $UR(u_0)$ $= \emptyset$

- $\forall i \leq n : \langle CA \cup [\![\mathcal{W}]\!], CR\rangle \triangleright UR_{i-1} \rightsquigarrow UR_i \wedge \mathsf{Done}[\checkmark] \in UR_n(u)$

We proceed by induction on the number of ill-formed elements in $UR_0, \ldots, UR_n$. If there is none, we are done. Otherwise, assume there are $k > 0$ ill-formed elements and let UR_i with $i > 0$ be the first one, we now show how to build a trace with $k - 1$ ill-formed elements which eventually assigns $\mathsf{Done}[\checkmark]$. This is enough to conclude by induction hypothesis.

We start by observing that $\forall i \leq n : Evt(UR_i) \in \mathbb{F}(\mathcal{E})$ by Lemma 4. Hence, there are only two potential ways to violate well-formedness and we show how to deal with them:

1. assume that there exist two different users u_1, u_2 and an event e such that $\mathsf{Author}[e] \in UR_i(u_1)$ and $\mathsf{Author}[e] \in UR_i(u_2)$. Since C does not contain binding-of-duty constraints, $\mathsf{Author}[e]$ can only occur in the negative preconditions of a can-assign rule, hence assigning $\mathsf{Author}[e]$ to u_2 does not enable new role assignments for him. Moreover, since $\mathsf{Author}[e] \in UR_i(u_1)$, role $\mathsf{Done}[e]$ can already be assigned by u_1. We then deduce that the assignment of $\mathsf{Author}[e]$ to u_2 is useless and we can remove it;

2. assume that each role of the form $\mathsf{Author}[e]$ is assigned to at most one user in UR_i, but $\hat{H}(UR_i) \not\models C$. Since C does not contain binding-of-duty constraints, there must exist two events e_1, e_2 such that $(e_1, e_2, \neq) \in C$ and $\{\mathsf{Author}[e_1], \mathsf{Author}[e_2]\} \subseteq UR_i(u)$ for some user u. This is not possible, since the rule assigning $\mathsf{Author}[e_1]$ includes $\mathsf{Author}[e_2]$ in the negative preconditions and vice-versa.

□

Definition 13 (Weak Correspondence). *We write $\langle UR, H \rangle \simeq UR'$ if and only if both the following conditions hold:*

a. $UR \approx_{reg} UR'$

b. for each event e and user u, $H(e) = u$ if and only if $\mathsf{Author}[e] \in UR'(u)$

Lemma 6 (Complete Simulation). *For any workflow $\mathcal{W}$ without binding-of-duty constraints, we have that $\langle UR_1, H \rangle \simeq UR_2$ and $\langle CA \cup [\![\mathcal{W}]\!], CR \rangle \triangleright UR_2 \rightsquigarrow UR_2'$ with UR_2, UR_2' well-formed for $\mathcal{W}$ imply that either $\langle UR_1, H \rangle \simeq UR_2'$ or there exists $\langle UR_1', H' \rangle$ such that $\langle CA, CR \rangle, \mathcal{W} \triangleright \langle UR_1, H \rangle \xrightarrow{e} \langle UR_1', H' \rangle$ and $\langle UR_1', H' \rangle \simeq UR_2'$.*

Proof. By a case analysis on the rule applied in the reduction step.

If $\langle CA \cup [\![\mathcal{W}]\!], CR\rangle \triangleright UR_2 \rightsquigarrow UR'_2$ is derived by rule (R-REVOKE), then we have $UR'_2 = UR_2 \setminus \{(u,r)\}$ for some u, r and the same can-revoke rule can be used to prove $\langle CA, CR\rangle, \mathcal{W} \triangleright \langle UR_1, H\rangle \xrightarrow{\circ} \langle UR_1 \setminus \{(u,r)\}, H\rangle$ and to conclude.

Otherwise, $\langle CA \cup [\![\mathcal{W}]\!], CR\rangle \triangleright UR_2 \rightsquigarrow UR'_2$ is derived by rule (R-ASSIGN) and we have $UR'_2 = UR_2 \cup \{(u,r)\}$ for some u, r. If r is a regular role, then the applied can-assign rule must belong to CA, hence the same rule can be applied to prove $\langle CA, CR\rangle, \mathcal{W} \triangleright \langle UR_1, H\rangle \xrightarrow{\circ} \langle UR_1 \cup \{(u,r)\}, H\rangle$ and to conclude. If r is not a regular role, we distinguish two sub-cases: if r does not have the form $\mathsf{Author}[e]$ for some event e, then $\langle UR_1, H\rangle \simeq UR_2 \cup \{(u,r)\}$ and we are done.

Otherwise, let $r = \mathsf{Author}[e]$ for some event e, we show that $\langle CA, CR\rangle, \mathcal{W} \triangleright \langle UR_1, H\rangle \xrightarrow{e} \langle UR_1, H[e \mapsto u]\rangle$, which is enough to conclude, since $\langle UR_1, H\rangle \simeq UR_2$ implies $\langle UR_1, H[e \mapsto u]\rangle \simeq UR_2 \cup \{(u, \mathsf{Author}[e])\}$. We first observe that u cannot be the super user, since rule (T-AUTHOR) explicitly ensures that this is not the case: hence, user u is also included in UR_1 and can potentially perform the task. We then prove that all the premises of rule (R-TASK) are satisfied:

- $e \notin dom(H)$: we known that $\hat{H}(UR_2) = H$ by definition of weakly corresponding states. We then observe that no user is assigned role $\mathsf{Author}[e]$ in UR_2, otherwise $UR_2 \cup \{(u, \mathsf{Author}[e])\}$ would not be well-formed;

- $\rho_{\mathcal{W}}(e) \in UR_1(u)$: since $\mathsf{Author}[e]$ is assigned to u, we know that $\rho_{\mathcal{W}}(e) \in UR_2(u)$ by definition of rule (T-AUTHOR). Given that $UR_1 \approx_{reg} UR_2$ by definition of weakly corresponding states, we have the desired conclusion;

- $H[e \mapsto u] \models C_{\mathcal{W}}$: since $UR_2 \cup \{(u, \mathsf{Author}[e])\}$ is well-formed for $\mathcal{W}$ by hypothesis, we have $\hat{H}(UR_2 \cup \{(u, \mathsf{Author}[e])\}) \models C_{\mathcal{W}}$. To conclude then, it is enough to observe that $\hat{H}(UR_2) = H$ by definition of weakly corresponding states, hence $\hat{H}(UR_2 \cup \{(u, \mathsf{Author}[e])\}) = H[e \mapsto u]$;

- $\exists X \subseteq dom(H) : X \vdash_{\mathcal{W}} e$: since $UR_2 \cup \{(u, \mathsf{Author}[e])\}$ is well-formed for $\mathcal{W}$ by hypothesis, we have $Evt(UR_2 \cup \{(u, \mathsf{Author}[e])\}) \in \mathbb{F}(\mathcal{E}_{\mathcal{W}})$. To conclude then, it is enough to observe that $\hat{H}(UR_2) = H$ by definition of weakly corresponding states;

- $\forall e' \in dom(H) : (e', e) \notin \#_{\mathcal{W}}$: since $UR_2 \cup \{(u, \mathsf{Author}[e])\}$ is well-formed for $\mathcal{W}$ by hypothesis, we have $Evt(UR_2 \cup \{(u, \mathsf{Author}[e])\}) \in \mathbb{F}(\mathcal{E}_{\mathcal{W}})$. To conclude then, it is enough to observe that $\hat{H}(UR_2) = H$ by definition of weakly corresponding states.

□

We are now ready to prove Theorem 2:

Proof. Let $\mathcal{A} = \langle\langle\langle CA, CR\rangle, UR\rangle, \mathcal{W}\rangle$ and assume that:

$$\langle CA \cup [\![\mathcal{W}]\!], CR\rangle \rhd UR \cup \{(u_0, \mathsf{Super})\} \rightsquigarrow^* UR',$$

for some UR' such that $\mathsf{Done}[\checkmark] \in UR'(u)$ for some user u. Since no role of the form $\mathsf{Author}[e]$ is assigned in the initial user-to-role assignment UR, we have that UR is well-formed for $\mathcal{W}$. By Lemma 5, we then know that there exists a sequence of user-to-role assignments $UR_1, \ldots, UR_n$ which are well-formed for $\mathcal{W}$ and:

$$\langle CA \cup [\![\mathcal{W}]\!], CR\rangle \rhd UR \cup \{(u_0, \mathsf{Super})\} \rightsquigarrow UR_1 \rightsquigarrow \ldots \rightsquigarrow UR_n,$$

and $\mathsf{Done}[\checkmark] \in UR_n(u')$ for user u'. Since $\langle UR, \bot\rangle \simeq UR \cup \{(u_0, \mathsf{Super})\}$, by multiple applications of Lemma 6 there exists $\langle UR'', H''\rangle$ such that:

$$\langle CA, CR\rangle, \mathcal{W} \rhd \langle UR, \bot\rangle \rightarrow^* \langle UR'', H''\rangle,$$

and $\langle UR'', H''\rangle \simeq UR_n$. Since $\mathsf{Done}[\checkmark] \in UR_n(u')$, there exists a user u'' such that $\mathsf{Author}[\checkmark] \in UR_n(u'')$, which implies $H''(\checkmark) = u''$ by $\langle UR'', H''\rangle \simeq UR_n$. Given that each event included in the history occurs in the trace by the definition of the reduction semantics, we can conclude. □

3.3 Proof of Theorem 3

We show that the preprocessing phase and the pruning rules do not affect the reachability of role $\mathsf{Done}[\checkmark]$, which is enough to prove the theorem.

Lemma 7 (Preprocessing). *Role* $\mathsf{Done}[\checkmark]$ *is reachable in* $[\![\mathcal{A}]\!]$ *if and only if it is reachable after preprocessing* $[\![\mathcal{A}]\!]$.

Proof. Recall that the processing amounts to two transformations of $[\![\mathcal{A}]\!]$:

1. each rule $(\mathsf{Super}, R_p, R_n, \mathsf{Allowed}[e])$ is replaced by the rules contained in the set $\{(\mathsf{Author}[e'], R_p, R_n, \mathsf{Allowed}[e]) \mid \mathsf{Done}[e'] \in R_p\}$;

2. all roles of the form $\mathsf{Allowed}[e]$ are removed from the negative preconditions of the can-assign rules.

We show the result separately for the two transformations:

1. for the "if" direction, we observe that $(\mathsf{Super}, R_p, R_n, \mathsf{Allowed}[e])$ can be activated only if there exists a user who is assigned (at least) all the roles $\mathsf{Done}[e'] \in R_p$. This means that, whenever the rule is activated, there must exist at least one user with role $\mathsf{Author}[e']$ for each e' such that $\mathsf{Done}[e'] \in R_p$, since $\mathsf{Done}[e']$ is initially unassigned and it can only be granted by a user with role $\mathsf{Author}[e']$, which is irrevocable. This implies that all rules in the set $\{(\mathsf{Author}[e'], R_p, R_n, \mathsf{Allowed}[e]) \mid \mathsf{Done}[e'] \in R_p\}$ can be activated, so it is possible to assign $\mathsf{Allowed}[e]$ when needed.

   For the "only if" direction, assume that one rule contained in the set $\{(\mathsf{Author}[e'], R_p, R_n, \mathsf{Allowed}[e]) \mid \mathsf{Done}[e'] \in R_p\}$ is activated. Since Super is always assigned to the super user in the initial user-to-role assignment and never revoked, we have that rule $(\mathsf{Super}, R_p, R_n, \mathsf{Allowed}[e])$ can also be activated, so it is possible to assign $\mathsf{Allowed}[e]$ when needed.

2. for the "if" direction, we observe that role reachability is preserved when dropping roles from the negative preconditions of a can-assign rule, since this makes the rule easier to activate.

   For the "only if" direction, assume that there exist two events e_1, e_2 such that $e_1 \# e_2$ and both $\mathsf{Allowed}[e_1]$ and $\mathsf{Allowed}[e_2]$ get assigned to the super user, which is not possible without the preprocessing step. We show that this possibility does not allow one to reach $\mathsf{Done}[\checkmark]$, if this was not possible before. In particular, notice that having both $\mathsf{Allowed}[e_1]$ and $\mathsf{Allowed}[e_2]$ assigned to the super user may lead to both $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ being assigned to some user of the system. This, in turn, may lead to the following scenarios:

   - $\mathsf{Done}[e_1]$ and $\mathsf{Done}[e_2]$ get both assigned to the super user: this was not possible without preprocessing, since only one between $\mathsf{Done}[e_1]$ and $\mathsf{Done}[e_2]$ could be granted before;
   - $\mathsf{Eq}[e_1, e_2]$ gets assigned to someone, whenever both $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ are given to the same user: this was not possible without

preprocessing, since only one between $\mathsf{Author}[e_1]$ and $\mathsf{Author}[e_2]$ could be granted before.

However, by definition of the encoding, having $\mathsf{Done}[e_1]$ and $\mathsf{Done}[e_2]$ together may only enable new transitions if there exist an event e and a configuration X such that $e_1 <_X e$ and $e_2 <_X e$. Similarly, having $\mathsf{Eq}[e_1, e_2]$ may only enable new transitions if there exist an event e' and a configuration Y such that $e_1 <_Y e'$ and $e_2 <_Y e'$. Both possibilities are excluded by the fact that $e_1 \# e_2$, hence e_1 and e_2 never occur together in a configuration.

□

Lemma 8 (Positive Stability)**.** *Let r be positively stable and let UR be a user-to-role-assignment such that $r \in UR(u)$ for some user u. Let r_t be a role, u_t be a user and assume there exists a sequence of user-to-role assignments $UR_0, \ldots, UR_n$ such that $UR_0 = UR$ and:*

$$(\forall i \leq n : \mathcal{P} \triangleright UR_{i-1} \rightsquigarrow UR_i) \land r_t \in UR_n(u_t).$$

Then, then there exists a sequence of user-to-role assignments $UR'_0, \ldots, UR'_m$ such that $UR'_0 = UR$ and:

$$(\forall i \leq m : \mathcal{P} \triangleright UR'_{i-1} \rightsquigarrow UR'_i \land r \in UR'_i(u)) \land r_t \in UR'_m(u_t).$$

Proof. Recall that a role r is positively stable if it is irrevocable or non-negative. If r is not revocable, the conclusion is trivial. If r is non-negative, there exists no transition blocked by the presence of r, hence we can leave it assigned without altering the reachability of r_t. □

Lemma 9 (Negative stability)**.** *Let r be negatively stable and let UR be a user-to-role-assignment such that $r \notin UR(u)$ for some user u. Let r_t be a role, u_t be a user and assume there exists a sequence of user-to-role assignments $UR_0, \ldots, UR_n$ such that $UR_0 = UR$ and:*

$$(\forall i \leq n : \mathcal{P} \triangleright UR_{i-1} \rightsquigarrow UR_i) \land r_t \in UR_n(u_t).$$

Then, then there exists a sequence of user-to-role assignments $UR'_0, \ldots, UR'_m$ such that $UR'_0 = UR$ and:

$$(\forall i \leq m : \mathcal{P} \triangleright UR'_{i-1} \rightsquigarrow UR'_i \land r \notin UR'_i(u)) \land r_t \in UR'_m(u_t).$$

Proof. Recall that a role r is negatively stable if it is not assignable or both non-positive and non-administrative. If r is not assignable, the conclusion is trivial. If r is both non-positive and non-administrative, there exists no transition enabled by the presence of r, hence we can leave it unassigned without altering the reachability of r_t. □

Lemma 10 (Precedence)**.** *Let r, r' be two roles such that $r \preceq r'$ and let UR be a user-to-role-assignment. Let r_t be a role, u_t be a user and assume there exists a sequence of user-to-role assignments $UR_0, \ldots, UR_n$ such that $UR_0 = UR$ and:*

$$(\forall i \leq n : \mathcal{P} \triangleright UR_{i-1} \rightsquigarrow UR_i) \wedge r_t \in UR_n(u_t).$$

Then, there exists a sequence of user-to-role assignments $UR'_0, \ldots, UR'_m$ such that $UR'_0 = UR$ and:

$$(\forall i \leq m : \mathcal{P} \triangleright UR'_{i-1} \rightsquigarrow UR'_i \wedge (\exists u' : r' \in UR'_i(u') \Rightarrow \exists u : r \in UR'_i(u))) \wedge r_t \in UR'_m(u_t).$$

Proof. Recall that $r \preceq r'$ if:

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 $r \in R_p \cup \{r_a\}$.

So we have two cases:

1. the role r is initially assigned and positively stable: by the positive stability lemma (Lemma 8) we know that we can find a trace where r is assigned in every UR'_j leading to r_t, so the conclusion follows;
2. the role r' is initially unassigned, r is positively stable and for all can-assign rule (r_a, R_p, R_n, r') we have $r \in \{r_a\} \cup R_p$. Assume r' gets assigned: in this case, there must exist a user with role r to trigger a can-assign rule granting r'. Since the role r is positively stable, by the positive stability lemma (Lemma 8) we can find a trace leading to r_t, where r is assigned in all UR'_j starting from the assignment of r'. This means that whenever r' is assigned, also r is assigned.

□

Lemma 11 (Rule 1)**.** *For any role $\hat{r}$, $\hat{r}$ is reachable in an ARBAC system $\mathcal{S}$ if and only it is reachable after applying rule 1.*

Proof. Let r_t be a non-negative role. Assume there exist a rule $ca = (r, R_p, R_n, r_t)$ and a rule $ca' = (r', R'_p \cup \{r_t\}, R'_n, r'_t)$ with $R_p \subseteq R'_p$, $R_n \subseteq R'_n$ and there exists $r'' \in R'_p \cup \{r'\}$ such that $r \preceq r''$. After the application of rule 1, we have that ca' is replaced by $ca'' = (r', R'_p, R'_n, r'_t)$.

For the "if" direction, we observe that role reachability is preserved when dropping roles from the positive preconditions of a can-assign rule, since this makes the rule easier to activate.

For the "only if" direction, assume ca'' is activated in a user-to-role assignment UR. This means there exist a user u_a such that $r' \in UR(u_a)$ and a user u such that $R'_p \subseteq UR(u)$ and $R'_n \cap UR(u) = \emptyset$. Since all the roles in the set $R'_p \cup \{r'\}$ are assigned in UR, we have that $r \preceq r''$ implies that there exists a user u'_a such that $r \in UR(u'_a)$ by Lemma 10. Hence, rule ca can be activated by u'_a to grant r_t to u and, afterwards, rule ca' can be activated by u_a to grant r'_t to u. Since role r_t is non-negative, the application of rule ca cannot break role reachability in the later transitions involving u. □

Lemma 12 (Rule 2). *For any role $\hat{r}$, $\hat{r}$ is reachable in an ARBAC system $\mathcal{S}$ if and only it is reachable after applying rule 2.*

Proof. Let r_t be a role. Assume there exist a rule $ca = (r, R_p, R_n, r_t)$ and a rule $ca' = (r', R'_p, R'_n, r_t)$ with $R_p \subseteq R'_p$, $R_n \subseteq R'_n$ and there exists $r'' \in R'_p \cup \{r'\}$ such that $r \preceq r''$. After the application of rule 2, we have that ca' is removed.

For the "if" direction, assume ca' is activated in a user-to-role assignment UR. This means there exist a user u_a such that $r' \in UR(u_a)$ and a user u such that $R'_p \subseteq UR(u)$ and $R'_n \cap UR(u) = \emptyset$. Since all the roles in the set $R'_p \cup \{r'\}$ are assigned in UR, we have that $r \preceq r''$ implies that there exists a user u'_a such that $r \in UR(u'_a)$ by Lemma 10. Hence, rule ca can be activated by u'_a to grant r_t to u, which makes the presence of rule ca' immaterial.

For the "only if" direction, we observe that role reachability is preserved when adding more can-assign rules to a policy. □

Lemma 13 (Rule 3). *For any role $\hat{r}$ which is not purely administrative, $\hat{r}$ is reachable in an ARBAC system $\mathcal{S}$ if and only it is reachable after applying rule 3.*

Proof. Let r_t be a purely administrative role and let $r \preceq r_t$. Assume there exist a rule $ca = (r, R_p, R_n, r_t)$ and a user u such that $R_p \subseteq UR(u)$ and $R_n \cap UR(u) = \emptyset$ for the initial user-to-role assignment UR, the roles in R_p are positively stable and the roles in R_n are negatively stable. After the application of rule 3, we remove ca and replace all the occurrences of r_t with r in the can-assign/can-revoke rules.

For the "if" direction, observe that:

1. the role r_t is purely administrative, so it will never occur in the positive/negative preconditions of any can-assign rule. This means that assigning r_t is only relevant to trigger further administrative actions available to r_t;

2. since $r \preceq r_t$, the precedence lemma (Lemma 10) ensures that r will always be assigned to some user if r_t is assigned.

For this reason we can delegate to r all the administrative privileges granted to r_t and remove the rule ca without affecting the reachability of any role different than r_t. Since r_t is purely administrative, this cannot falsify the conclusion.

For the "only if" direction, we observe that there exists a user u that satisfies the positive/negative preconditions of $ca = (r, R_p, R_n, r_t)$ in the initial user-to-role assignment UR and the roles in R_p/R_n are positively/negatively stable. By Lemma 8 and Lemma 9, this means that we can assume that u will always satisfy the positive/negative preconditions of ca. Assume ca is added to the ARBAC policy: the consequence of the rule is that u can be granted r_t whenever the role r is assigned to some user of the system. Thus, after this addition, we can delegate to r_t all the administrative privileges previously granted to r without affecting the reachability of any role different than r_t. Since r_t is purely administrative, this cannot falsify the conclusion. □

Lemma 14. *If a role r is not administrative, then is not administrative also after the pruning phase.*

Proof. By a simple inspection of the preprocessing and pruning rules. □

To prove Theorem 3, we can then proceed as follows:

Proof. It is enough to observe that:

1. Lemma 7 ensures that the reachability of $\mathsf{Done}[\checkmark]$ is not affected by the preprocessing;

2. Lemmas 11, 12, 13 ensures that the reachability of $\mathsf{Done}[\checkmark]$ is not affected by the pruning rules, as long as role $\mathsf{Done}[\checkmark]$ is not purely administrative;

3. role $\mathsf{Done}[\checkmark]$ is not administrative. This is indeed the case for all roles of the form $\mathsf{Done}[e]$ for some event e. Lemma 14 ensures that $\mathsf{Done}[\checkmark]$ never becomes administrative throughout the pruning phase. This, in particular, means that $\mathsf{Done}[\checkmark]$ never becomes purely administrative.

□

Chapter 4

Verification of Administrative Attribute-Based Access Control Policies

4.1 Presentation of the work

In this chapter we present a work on Administrative Attribute-Based Access Control systems and their verification.

Attribute-Based Access Control. Attribute based access control (ABAC) proved to be an intuitive and effective access control system. It provides many advantages of other access control systems, since permissions are given directly on formulas predicating on attributes of users, thus enabling a fine-grained level of control. Moreover, it avoids intermediate layers like in other systems, e.g., the RBAC, thus resulting more intuitive and expressive. Many works addressed the problem of giving an administrative version of ABAC [47], [46], [48], [38], [4], however most of them delegate the administration to a set of roles as in ARBAC. In this work we give an administrative model for attribute-based access control systems (AABAC) without using an intermediate layer. Instead we encode directly administrative operations on attributes predicating only on them. This is done in a way that is similar to how ARBAC is administrated [68]; indeed, to set an attribute we require two users, the administrator and the target user, that have to satisfy two boolean preconditions each predicating on their attributes. We will show in section 4.2 that the proposed model is general enough to encode ARBAC systems, as well as hierarchical ARBAC ones.

Verification. Security analysis of access control systems is widely recognized as a crucial problem, since it gives to administrators a way to check whether their policies satisfy some security relevant properties. This in ARBAC systems is traditionally encoded in terms of the so called role reachability problem. In it we want to prove if a given role could be ever assigned to a user in the system. Although this problem could appear simple, it has been proved that it is hard to solve due to the exponential explosion of the state space. For this reason, substantial researches have been carried out to address the verification of the role-reachability problem [71], [41], [31], [32], [33], [18], [64] and [40]. Moreover several works address the role-reachability problem in systems that are extensions of ARBAC: [65] and [73] address the verification of ARBAC policies with temporal constraints, while in [28] is defined a verification techniques for ARBAC systems with temporal and spatial constraints; in [4] is performed an analysis of an ARBAC system in presence of immutable attributes. Finally in [70] is conducted an analysis of parametrized ARBAC policies. In this work we present the dual of role reachability problem for AABAC systems, namely the satisfiability problem. In it we want to check if in an AABAC system a certain target formula could be ever satisfied.

Thanks to the generality of the presented model we are also able to verify all the problems for which there is a reduction to a standard ARBAC role reachability problem, such as the work about workflow security presented at chapter 2.

Contributions. In this chapter we make the following contributions:

- we provide a novel *specification* of an administrative version of the Attribute-Based Access Control system (AABAC) and we show that it is more general than ARBAC systems;
- we devise a technique to reduce the number of users in a system that could also be exploited to reduce systems with unbounded number of users to bounded ones;
- we give two techniques, one under-approximated and one over-approximated, for the *verification* of satisfiability problems for AABAC systems;
- we propose a simplification technique to reduce the size of the satisfiability problem without affecting its satisfiability;

- we implement all the aforementioned techniques in the VACSAT tool, and we test its effectiveness against a large set of benchmarks from the literature.

Structure of the chapter. This chapter is structured as follow:

- in section 4.2 we formally define an AABAC model, and give a notion of satisfiability problem;
- in section 4.3 we give a technique used to reduce the number of users in a satisfiability problem that could be exploited to reduce AABAC systems where the number of users is unbound to systems where such number is bound;
- in section 4.4 we introduce the syntax and the semantics of a language for concurrent programs that will be used in sections 4.5 and 4.6;
- in section 4.5 we devise a novel precise, but under-approximated analysis technique based on concurrent program verification;
- in section 4.6 we give an over-approximated analysis technique based on program verification;
- in section 4.7 we define a simplification technique aimed to reduce the size of the satisfiability problem before the verification;
- in section 4.8 we describe VACSAT: a tool for checking satisfiability problem in AABAC systems, and we show its results;
- in section 4.9 we draw our conclusions.

4.2 Model

In this section we introduce the novel Administrative Attribute-Based Access Control (AABAC) formal model.

An Administrative Attribute-Based Access Control (AABAC) is an access control system where we have a set of users each having a finite set of attributes. To each attribute are assigned values taken from a finite domain. Users are allowed to change the values of their attributes by means of administrative actions. Those actions are subject to two preconditions: the former, called administrative precondition, must be satisfied by any user in the system (administrator), while the latter must be satisfied by the user whose attribute will be changed (target user).

4.2.1 Components

Let U be a finite set of users, A a finite set of attributes and V a set of values, and let u, a and v range respectively on them.

Definition 14 (Range). *The function $Range : A \rightarrow 2^V$ is a map assigning all the possible values of each attribute a.*

Definition 15 (Attribute-value assignment). *An* attribute-value assignment *$AV : A \rightarrow V$ is a map assigning values to each attribute in A such that $\forall a \in A : AV(a) \in Range(a)$,.*

Definition 16 (User-attribute assignment). *A* user-attribute assignment *$UA : U \rightarrow AV$ is a map assigning values to each attribute of a user u.*

In the AABAC model, the user-attribute assignment can be changed by means of *administrative actions*, enabled by a user-attribute administration *policy* defining conditions for setting values to user attributes. All administrative actions are constrained by means of *preconditions* φ that must be satisfied by both administrator and target user. Preconditions are defined as follow:

Definition 17 (Precondition). *A precondition φ is a boolean formula predicating on a set of attributes of A:*

$$\varphi ::= true \mid a \sim v \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

The set of all possible precondition is Φ. We decided to leave the relation operators $\sim$ unspecified to support various comparison operators.

We introduce the domain $Dom(\varphi)$ of a precondition as the set of attribute appearing in it. Formally:

Definition 18 (Domain). *Let φ be a precondition, its domain $Dom(\varphi)$ is a set of attribute such that:*

$$Dom(\varphi) = \begin{cases} \emptyset & if\ \varphi = true \\ \{a\} & if\ \varphi = (a \sim v) \\ Dom(\varphi') & if\ \varphi = \neg\varphi' \\ Dom(\varphi') \cup Dom(\varphi'') & if\ \varphi = (\varphi' \wedge \varphi'') \\ Dom(\varphi') \cup Dom(\varphi'') & if\ \varphi = (\varphi' \vee \varphi'') \end{cases}$$

Let AV be an *attribute-value-assignment* and φ a precondition we say that $AV \vdash \varphi$, or AV satisfies φ, as defined in table 4.1.

Table 4.1 Semantics of preconditions

$$\text{P-TRUE}\ \frac{}{AV \vdash \mathit{true}} \qquad \text{P-REL}\ \frac{AV(a) \sim v}{AV \vdash a \sim v} \qquad \text{P-NOT}\ \frac{AV \nvdash \varphi}{AV \vdash \neg\varphi} \qquad \text{P-AND}\ \frac{AV \vdash \varphi \quad AV \vdash \varphi'}{AV \vdash \varphi \wedge \varphi'} \qquad \text{P-OR1}\ \frac{AV \vdash \varphi}{AV \vdash \varphi \vee \varphi'}$$

$$\text{P-OR2}\ \frac{AV \vdash \varphi'}{AV \vdash \varphi \vee \varphi'}$$

Here we give a notion of equivalence between two preconditions.

Definition 19 (Formulas equivalence)**.** *Let φ_1 and φ_2 be two preconditions. We say that φ_1 and φ_2 are equivalent $\varphi_1 \equiv \varphi_2$ iff $\forall AV : AV \vdash \varphi_1 \iff AV \vdash \varphi_2$.*

Intuitively φ_1 and φ_2 are considered equivalent if they are satisfied by the same attribute-value assignments.

Definition 20 (Policy)**.** *A* policy *is a tuple $P = (U, A, V, CS)$ where $CS \subseteq \Phi \times \Phi \times (A \times V)$.*

A can-set rule $(\varphi_a,\ \varphi_u : a = v) \in CS$ states that a user whose *attribute assignment* satisfies the administrative precondition φ_a can set the attribute a of any user who satisfies the precondition φ_u to value v.

Given a policy $P = (U, A, V, CS)$, we define $\Phi_a = \{\varphi_a \mid (\varphi_a,\ \varphi_u : a = v) \in CS\}$ and $\Phi_u = \{\varphi_u \mid (\varphi_a,\ \varphi_u : a = v) \in CS\}$ as the sets respectively of administrative/user preconditions of the can-set rules.

All the syntactic ingredients introduced so far define an AABAC *system* $S = (P, UA)$. Though most of the definitions in this section are parametric with respect to the choice of an AABAC system, when there is no ambiguity we implicitly consider a system S with the above structure and such that $P = (U, A, V, CS)$.

Definition 21 (User count)**.** *Given a AABAC system $S = ((U, A, V, CS), UA)$ and an attribute-value assignment AV, we define $count_S(AV) = |\{u \mid u \in U \wedge UA(u) = AV\}|$ as the number of users having AV as attribute-value assignment.*

We can now define the reduction semantics of AABAC. This is done by means of judgments of the form $UA \overset{r}{\rightsquigarrow}_P UA'$, reading as: the user-attribute assignment UA can be updated to UA' by means of an administrative action r allowed by the policy P. The definition of the judgment is given in Table 4.2. The table provides the formal counterpart of the intuitive semantics of attribute set as allowed by the policy P. As usual, we let $\rightsquigarrow_P^*$ stand for the reflexive-transitive closure of $\rightsquigarrow_P$.

Table 4.2 Semantics of Administrative Actions

R-APPLY

$$\frac{r = (\varphi_a,\ \varphi_u : a = v) \in CS \qquad UA(u_a) \vdash \varphi_a \qquad UA(u) \vdash \varphi_u}{UA \overset{r}{\rightsquigarrow}_P UA[u \to UA(u)[a \to v]]}$$

4.2.2 Satisfiability Problem

The *satisfiability problem* consists of checking whether a given formula φ_t, called satisfiability query, can be satisfied by some user of an AABAC system. Formally, an *instance* of the attribute-value satisfiability problem is a pair $I = (S, \varphi)$, where S is an AABAC system and φ is the satisfiability query.

Definition 22 (Satisfiability). *Let $I = (S, \varphi_t)$ be an instance of the satisfiability problem such that $S = (P, UA)$. We say that I is* satisfiable *iff there exist a user-attribute assignment UA_n and a user $u \in U$ such that $UA \rightsquigarrow_P^* UA_n$ and $UA_n(u) \vdash \varphi_t$.*

Though simple to formalize, such satisfiability problem is a very useful tool for checking the security of an AABAC system, since many security relevant properties could be encoded in term of formula φ. Indeed we can easily encode problems such as the attribute dual of role reachability problem; a more precise query stating if a given user u could have an attribute a_i set to value v; or again if a user u could have an attribute a with as value any element of the set B.

4.2.3 AABAC with unbounded number of users

In this section we will extend the introduced model giving the possibility to add/remove any number of users during the execution. This mimics the hiring and retiring of a company. Each user that is added to the system enters with a predetermined initial attribute-value configuration. We then show that, thanks to theorem presented in section 4.3, we can reduce it to a model where the number of users is bound.

To generalize the aforementioned AABAC model we have to support two different operations: in the first we allow users to exit the system, whereas in the second an unbounded number of users can enter the system. While to support users removal we do not have to do any syntactical change to the model, to support user addition we need to add two components to the AABAC model.

Definition 23 (AABAC$_\infty$ system)**.** *Let W be an unbounded set of users, and let $WA = \{AV_1, \dots, AV_n\}$ be a set of all the possibles attribute-value assignments of users entering the system: we define an AABAC$_\infty$ system $S_\infty = ((U, A, V, CS, W, WA), UA)$.*

The reduction semantics of user entering/leaving the system are given in table 4.3. In the first rule we allow a user in W, that is not part of the system ($u \notin U$) to enter it with the initial attribute-value assignment AV. The second one, instead says that a user can leave the system. If it does so, it will be removed from U, and its configuration will be removed from the user-attribute assignment.

Table 4.3 Semantics of user add/remove - $S_\infty \hookrightarrow S'_\infty$

USER-ADD
$$\frac{u \in W \qquad u \notin U \qquad AV \in WA}{((U, A, V, CS, W, WA), UA) \hookrightarrow ((U \cup \{u\}, A, V, CS, W, WA), UA[u \to AV])}$$

USER-REMOVE
$$\frac{u \in U}{((U, A, V, CS, W, WA), UA[u \to AV]) \hookrightarrow ((U \setminus \{u\}, A, V, CS, W, WA), UA)}$$

Notice that an AABAC system $S = ((U, A, V, CS), UA)$ is equivalent to an AABAC$_\infty$ one $S_\infty = ((U, A, V, CS, \emptyset, \emptyset), UA)$. Indeed the system S is an AABAC$_\infty$ one where no users can be added or removed.

4.2.4 ARBAC to AABAC translation

We will now show that the AABAC model defined here is sufficiently general to encode both the standard ARBAC model like the one introduced in chapter 2 and a generalization of it with the addition of roles hierarchies.

ARBAC. Here we will show how to reduce an Administrative Role-Based Access Control instance, as defined in section 2.2, to an AABAC one. Intuitively, it is enough to model users as individuals with a boolean-valued attribute for each role, identifying if the user has that role. Each can-assign rule will introduce a can-set one having the check on the administrative role, now attribute, a conjunction on positive and negative preconditions, and seting the value of the target attribute as *true*. Each can-revoke, similarly, will be encoded with a can-set rule checking only the administrative precondition (it has *true* as precondition) and seting the target attribute to *false* as effect. More formally let $\mathcal{S} = \langle \mathcal{P}, UR \rangle$ be an ARBAC system where $\mathcal{P} = \langle CA, CR \rangle$, we now construct a new AABAC $((U, A, V, CS), UA)$ model such that:

- U is the set of users in the ARBAC system;
- for each role r in the ARBAC system we create an attribute r with a boolean domain;
- $V = \{0, 1\}$ be the set of boolean values;
- for each can-assign rule of the ARBAC policy $(r_a, R_p, R_n, r_t) \in CA$ we add a can-set rule $(r_a = 1,\ (\bigwedge_{r_p \in R_p}(r_p = 1)) \wedge (\bigwedge_{r_n \in R_n}(r_n = 0)) : r_t = 1)$ to CS;
- for each can-revoke rule of the ARBAC policy $(r_a, r_t) \in CA$ we add a can-set rule $(r_a = 1,\ true : r_t = 0)$ to CS;
- for each user u and role r: $UA(u)(r) = (r \in UR(u))$.

Then we can easily encode each possible ARBAC role reachability problem, having role r_t as target, as a satisfiability problem $I = (((U, A, V, CS), UA), r_t = 1)$.

Hierarchical ARBAC. We can also encode, as an AABAC system, any instance of hierarchical ARBAC. This is a little bit trickier than the standard once since it comes with a partial order on roles $\preceq$ such that $r \preceq r'$, which implies that r' has at least as many rights as r. The main part of the encoding is identical to the one for standard ARBAC, the only difference is how we translate the can-assign and can-revoke rules. Given a role r, let $r^{\uparrow} = \{r' \mid r \prec r'\}$ be the set of roles higher than r. In this case, the can-assign rule (r_a, R_p, R_n, r_t) is by having:

$$(\bigvee_{r'_a \in r_a^{\uparrow}} (r'_a = 1),\ (\bigwedge_{r_p \in R_p} \bigvee_{r'_p \in r_p^{\uparrow}} (r'_p = 1)) \wedge (\bigwedge_{r_n \in R_n} \bigwedge_{r'_n \in r_n^{\uparrow}} (r'_n = 0)) : r_t = 1)$$

The encoding of the can-revoke rule (r_a, r_t) depends on the semantic of revocation. If weak revocation is chosen, the rule is encoded as follows:

$$(\bigvee_{r'_a \in r_a^{\uparrow}} (r'_a = 1),\ true : r_t = 0).$$

If strong revocation is chosen, the rule is instead encoded as follows:

$$(\bigvee_{r'_a \in r_a^{\uparrow}} (r'_a = 1),\ \bigwedge_{r'_n \in r_t^{\uparrow} \setminus \{r_t\}} (r'_n = 0) : r_t = 0)$$

This rule forces one to revoke all the senior roles before revoking r_t.

This two encodings give us the capability of using our model and verification techniques to check any other policy written in ARBAC, and moreover even other access control systems where the analysis is done in terms of reduction to ARBAC role reachability e.g., temporal ARBAC analysis [73] and the one done in chapter 2.

4.3 Bounding the number of user

In this section we introduce an important theorem that allows us to reduce the number of users tracked in a satisfiability problem. Moreover it allows us to transform an AABAC$_\infty$ satisfiability problem I_∞ to a bounded one. The theorem is a generalization for AABAC systems of [33, theorem 1]. For the sake of readability in the theorem and its proof we refer to an AABAC system, but we can apply it even to an AABAC$_\infty$ one.

Theorem 4. *Let $I = (((U, A, V, CS), UA), \varphi_t)$ be a satisfiability problem where and $\Phi_a^{\not\equiv} = \Phi_a/\equiv$ be the set of non equivalent administrative preconditions of CS, and $k = |\Phi_a^{\not\equiv}|$. If there exists a trace leading to the satisfaction of φ_t, then there exists another trace leading to satisfaction of φ_t where at most $k + 1$ users change their attribute-value assignment.*

Proof of this theorem mimics the one shown in [33, theorem 1] replacing administrative roles with administrative preconditions equivalence classes. For completeness we report it here.

Proof. For any run $\pi = UA_1 \overset{r_1}{\rightsquigarrow}_P UA_2 \overset{r2}{\rightsquigarrow}_P \ldots UA_n \overset{r_n}{\rightsquigarrow}_P UA_{n+1}$ of S, let $a(\pi) = \varphi_{a_1}, \varphi_{a_2}, \ldots, \varphi_{a_n}$ be the sequence of administrative preconditions where $r_j = (\varphi_{a_j}, \varphi_{u_j} : a_j = v_j)$ is the rule applied in the j^{th} transition of π. We say that a user u is *engaged* in π if and only if there exists a transition $UA_i \overset{r_i}{\rightsquigarrow}_P UA_{i+1}$ in π such that $UA_i(u) \neq UA_{i+1}(u)$. Furthermore, we say that u is *essential* in π iff it is *engaged*, and u is the only user in the system satisfying φ_{a_j} for $j \in \{1, \ldots, n\}$. Notice that thanks to the notion of equivalence of formulas, we know that if we have two administrative preconditions φ_{a_1}, φ_{a_2} and two users u_1 and u_2 such that $\varphi_{a_1} \neq \varphi_{a_2}$, $\varphi_{a_1} \equiv \varphi_{a_2}$, if u_1 satisfies φ_{a_1} and u_2 satisfies φ_{a_2} then both u_1 and u_2 are not *essential*. Finally, let $index_\pi(u)$ be the greatest $j \in \{1, \ldots, n\}$ such that u is the only user satisfying φ_{a_j} in UA_j.

We now show that, for each run π in which φ_t is satisfied by a user u_t, it is possible to construct another run π' where there are only $k+1$ *engaged* users. For simplicity we assume that u_t satisfies φ_t only in UA_{n+1}. From π we create π' by repeatedly applying two simplification rules. In the following part we assume $\pi_0 = \pi$ and π_i be the run obtained after i simplification steps.

1. If π_i contains an *engaged* user u such that $u \neq u_t$, which is not *essential*, then remove from π_i all transitions changing the attribute-value assignment of u;

2. If all *engaged* users in π_i are *essential*, then pick one of them different from u_t such that there exists a transition r_j changing u's attribute-value assignment, with $j \leq index_{\pi_i}(u)$, and remove from π_i all transitions r_l targeting u with $l \geq index_{\pi_i}(u)$.

We are guaranteed that this simplification process terminates since at each step we reduce the length of the run. Moreover if π_i is a valid run, each simplification step produces a new valid run.This is because we always leave a user in any administrative role to fire any transition of π_i.

Finally, to conclude the proof, we show that any simplified run π'_0 has at most $k+1$ engaged users. We know that each *engaged* user u, with $u \neq u_t$, in π' is *essential*, and u's attribute-value assignment does not change after $UA_{index_{\pi'}(u)}$. Thanks to this for any two distinct users $u_1 \neq u_2$ (both different from u_t), *engaged* in π', it holds that $\varphi_{a_{j_1}} \not\equiv \varphi_{a_{j_2}}$ with $j_1 = index_{\pi'}(u_1)$ and

$j_2 = index_{\pi'}(u_2)$. Thus, the number of engaged users in π' is at most equal to the number of non equivalent administrative preconditions $k = |\Phi_a^{\neq}|$ plus one representing u_t. □

4.3.1 User removal

Thanks the aforementioned theorem, we are now able to bound users in an AABAC$_\infty$ system, therefore obtaining a bounded AABAC one. Moreover this procedure can be used even to reduce the users of an AABAC system. Let $S_\infty = ((U, A, V, CS, W, WA), UA)$ be an AABAC$_\infty$ system and $k = |\Phi_a^{\neq}|$, we can reduce the system to a new AABAC with bounded number of users using the reduction function $\lfloor S_\infty \rfloor = S$ where $S = ((U, A, V, CS), UA)$. The function $\lfloor S_\infty \rfloor$ works applying to S_∞ the simplification steps described below:

1. for each attribute-value assignment $AV \in WA$, let $j = count_S(AV)$ be the number of users in U sharing the attribute-value assignment AV; if the j is less than $k + 1$, we then take $m = (k + 1) - j$ fresh users from W and we add them to the system with attribute-value assignment AV, finally we remove AV from WA, since no more users are required to be added with attribute-value AV;

2. for each attribute-value assignment AV, let $j = count_S(AV)$ be the number of users in U sharing the attribute-value assignment AV. If j is greater than $k + 1$ then remove $n = j - (k + 1)$ users from the system.

After the application of step (1) and (2) we obtain a new system $S'_\infty = ((U', A, V, CS, W, \emptyset), UA')$ where for each attribute-value assignment AV we have at maximum $k + 1$ users. Thanks to the aforementioned theorem we can remove all the users not in the AABAC$_\infty$ system, obtaining an AABAC one with bounded number of users $S = ((U', A, V, CS), UA')$.

4.4 Concurrent Programs

In this section we will introduce the syntax and an informal semantics of the language we will use for the analysis. Concurrent programs are imperative programs operating on variables, whose procedures can be concurrently executed by multiple threads.

4.4.1 Syntax and Informal Semantics

The syntax of concurrent programs is given in Table 4.4. A *program* p is a sequence of declarations of *global variables* $\overrightarrow{x}$ followed by a sequence of declarations of *procedures* $\overrightarrow{f}$. Procedures have a name *id*, a sequence of parameters $\overrightarrow{y}$ and a body s (a *statement*, see below). For simplicity, we identify the *local variables* available to procedures with their parameters. We assume that each procedure only operates on its local variables and on the global variables, and that all the global and local variables are pairwise distinct. Finally, we require that the names of the procedures are pairwise distinct to avoid ambiguities.

The syntax of statements is adapted from the syntax of CProver [50]. Before introducing it, it is convenient to describe the simple language of *expressions*, which can be evaluated to values during statement execution. An expression e may be: a value $v \in V$, a variable x, the non-deterministic choice operator $*$, which may evaluate to any value of its domain, a conditional (ternary) expression $e\ ?\ e\ :\ e$, a procedure call $id(\overrightarrow{e})$ passing as arguments the sequence $\overrightarrow{v}$ obtained from the evaluation of $\overrightarrow{e}$. It may also be a set of expression compositions such as comparison $e \sim e$, disjunction or conjunction of expressions $e \vee e$, $e \wedge e$, or negation of an expression $\neg e$.

The **skip** statement does nothing. The parallel assignment $\overrightarrow{x} := \overrightarrow{e}$ atomically evaluates the sequence of expressions $\overrightarrow{e}$ into a sequence of values $\overrightarrow{v}$ and assigns each v_i in $\overrightarrow{v}$ to the corresponding variable x_i in $\overrightarrow{x}$. The conditional (**if** e **then** s **else** s) and loop (**while** e **do** s) statements are standard. The assumption statement **assume**(e) checks whether e evaluates to *true*: if not, it terminates the program silently, while assertion statement **assert**(e), if e is false, moves the program into a failure state (represented by **fail**). The **spawn** $id(\overrightarrow{e})$ statement creates a new thread running the body of the procedure *id*, passing, as arguments, the sequence of values obtained by evaluating the expressions $\overrightarrow{e}$. Statements **atomic-begin** and **atomic-end** are used to enter and exit an atomic section, to avoid race conditions on the global memory by different threads. Notice that statements can be combined with the semi-colon operator to produce new statements.

4.4.2 Reachability Problem

The reachability problem for concurrent programs amounts to check whether it is possible to reach a failure state under a given program. This is a well

Table 4.4 Syntax of Concurrent Programs

$$
\begin{array}{llcl}
\textit{Programs} & p & ::= & \textbf{decl}\ \overrightarrow{x};\ \overrightarrow{f} \\
\textit{Procedures} & f & ::= & \textbf{procedure}\ id(\ \overrightarrow{y}\ )\ s\ \textbf{end procedure} \\
\textit{Statements} & s & ::= & \textbf{skip} \mid s;s \mid \overrightarrow{x} := \overrightarrow{e} \\
 & & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ e\ \textbf{do}\ s \\
 & & \mid & \textbf{assume}(e) \mid \textbf{assert}(e) \mid \textbf{fail} \\
 & & \mid & \textbf{spawn}\ id(\ \overrightarrow{e}\ ) \mid \textbf{atomic-begin} \mid \textbf{atomic-end} \\
\textit{Expressions} & e & ::= & v \mid * \mid x \mid e\ ?\ e\ :\ e \\
 & & \mid & e \sim e \mid \neg e \mid e \vee e \mid e \wedge e
\end{array}
$$

known problem, and the literature presents several approaches to solve it. In sections 4.5 and 4.6 we will create a program that simulates a given satisfiability problem, and then we will discharge the check of the obtained program to state of the art checker.

4.5 Precise analysis

4.5.1 Analysis

Given an instance of the satisfiability problem $I = (S, \varphi_t)$, we define an encoding of I as a concurrent program $[\![I]\!]$ such that I is satisfiable if and only if $[\![I]\!]$ can reach a failure state. We now introduce our construction and argue its correctness.

4.5.2 Structure of the Encoding

The key idea of the encoding is representing the reachable user-attribute assignments as thread pools, where each thread tracks all the attributes of a given user of the AABAC system. Specifically, each thread has one local variable l_a_i for each attribute a_i, satisfying the invariant that $l\_a_i = v_i$ if and only if the attribute a_i is set to value v_i to the user encoded by the thread $(UA(u)(a_i) = v_i)$.

The code of each thread defines how the value of the local variables can be updated to reflect the application of the administrative actions of the AABAC system, setting value to attributes. The tricky part of the encoding is how to efficiently check whether an administrative action can be performed. Indeed, although the local memory of each thread stores enough information to check whether the encoded user satisfies the precondition of a can-set rule

φ_u, it does not track whether there exists an administrator who can trigger the rule, since this information may only be available in the local memory of another thread. Thus, in the encoding, we rely on the global memory: for each administrative precondition φ_a we introduce a global variable g_φ_a satisfying the invariant that, if $g\_\varphi_a = true$, then there exists a user satisfying the precondition φ_a (but not necessarily vice-versa).

The reason why we sacrifice the converse of the presented invariant is efficiency. Indeed, ensuring a stronger invariant which additionally includes the converse would be costly: if an attribute of a user u appearing in φ_a is set to a value v such that the precondition is not valid anymore after the set, then one would need to check all the other users of the system to assess whether u was the last user satisfying φ_a, and set the global variable g_φ_a to $false$ only in that case. In the encoding, this check would require a communication between all threads, leading to a major impact on the verification times. The good news, however, is that the proposed weaker invariant does not introduce false negatives in the verification process, since we can always compensate the apparent mismatch thanks to a smart treatment of the can-set rules and a quantification over all the program runs (see below).

The structure of the encoding of the instance $I = (S, \varphi_t)$ is shown in Table 4.5. It is a program with just two procedures: the INIT procedure initializes all the global variables to $false$, which trivially satisfies the aforementioned invariant, and spawns one thread running the *user* procedure for each user of S, passing as parameters (local variables) the encoding of the attribute assigned to the corresponding user in the initial user-attribute assignment. The USER procedure, which is run by all the spawned threads, executes an infinite while loop, where three main operations are performed. (1) The global memory is updated: if the interpretation of a precondition $[\![\varphi_a]\!]$ is satisfied, then we are guaranteed that there exists at least one user satisfying it and we can safely set g_φ_a to $true$. (2) In the second loop of the function (rows 20-22), all the administrative actions allowed by the policy of S, call it $r \in CS$, are simulated by statements $[\![r]\!]$: the details of this compilation step are explained in the next section. (3) The procedure executes an assert statement to check whether the query is satisfied: if this is the case, the program enters a failure state. Notice that the program runs all operations using the global state atomically, to ensure the absence of race conditions on the global memory.

Table 4.5 Definition of the Encoding

Input: an instance of the satisfiability problem $I = (S, \varphi_t)$, where $S = (U, A, V, CS, UA)$, $U = \{u_1, \ldots, u_k\}$, $A = \{a_1, \ldots, a_n\}$.

Output: a concurrent program $[\![I]\!]$ with the following structure:

```
1: for $\varphi_a \in \Phi_a$ do
2:     decl $g\_\varphi_a$    ▷ one global variable for each administrative precondition
3: end for

4: procedure INIT()                                        ▷ initialization procedure
5:     atomic-begin
6:     for $\varphi_a \in \Phi_a$ do
7:         $g\_\varphi_a := false$        ▷ conservative global memory initialization
8:     end for
9:     for $u_j \in U$ do
10:        spawn USER($l\_a^j_1, \ldots, l\_a^j_n$)   ▷ $\forall i \in \{1, \ldots, n\} : l\_a^j_i = UA(u_j)(a_i)$
11:    end for
12:    atomic-end                  ▷ after this, $UA$ was encoded as a thread pool
13: end procedure

14: procedure USER($l\_a_1, \ldots, l\_a_n$)                    ▷ user thread code
15:    while $true$ do
16:        atomic-begin
17:        for $\varphi_a \in \Phi_a$ do
18:            $g\_\varphi_a := g\_\varphi_a \vee [\![\varphi_a]\!]$     ▷ user's conservative global update
19:        end for
20:        for $r \in CS$ do
21:            $[\![r]\!]$                      ▷ encoding of the administrative rule $r$
22:        end for
23:        assert($\neg[\![\varphi_t]\!]$)         ▷ check whether $\varphi_t$ is satisfied by the user
24:        atomic-end
25:    end while
26: end procedure
```

Precondition compilation Let φ be a precondition, $[\![\varphi]\!]$ is an expression representing the interpretation of φ in the local memory context. To do that we syntactically substitute all occurrences of any attribute a_i in φ with its local representation l_a_i.

4.5.3 Policy Compilation

Given a policy P, we define its compilation in terms of a loop which simulates the encoding all the administrative rules of $r \in P$. Policy compilation is shown in the second loop of the USER procedure (table 4.5, rows 20-22). Given the definition of the compilation of the rules, the order of execution is immaterial. In the following, we stick to the convention that g_φ_a is the global variable such that $g\_\varphi_a = true$ implies the existence of a user satisfying precondition φ_a, and l_a is the local variable such that $l\_a = v$ if and only if the user encoded by the running thread has the attribute a set to value v. The definition of the compilation of the individual rules is given in Table 4.6 and explained below.

Can-Set The can-set rule $(\varphi_a,\ \varphi_u : a_t = v_t) \in CS$ is compiled into a conditional statement as shown in table 4.6. Its guard is a conjunction of several expressions:

1. $*$: the non-deterministic choice operator, modeling that the rule may either be applied or not;
2. g_φ_a: the global variable g_φ_a must be *true*. This ensures the existence of an administrator who can apply the can-set rule;
3. $[\![\varphi_u]\!]$: the rule's precondition compilation under local memory. This ensures that the local memory of the running thread encodes the role assignment of a user satisfying the precondition of the can-set rule;
4. $\neg(l\_a_t = v)$: the local memory of the running thread should not have the attribute set to v. This is just used to avoid useless computations.

If the conjunction is satisfied, the consequent of the conditional is taken. We set the local variable l_a_t to v_t to reflect the attribute assignment. For all administrative precondition $\varphi_a{}'$ whose domain contains attribute a_t, we check if the current user satisfy them and we update the global memory $g\_\varphi_a{}'$ accordingly, since after the application of the can-set rule we do not know if there exists a user satisfying them. Notice, however, that this behaviour does not introduce false negatives: intuitively, the reason is that, if there exists another user satisfying an administrative precondition φ_a, the thread corresponding to that user will be eventually scheduled for execution and for this thread we would have $[\![\varphi_a]\!] = true$. Since the *user* procedure (in Table 4.5) sets g_φ_a to

$g\_\varphi_a \vee [\![\varphi_a]\!]$, the global variable g_φ_a will be eventually set to *true* as required for soundness.

Table 4.6 Compilation of the Administrative Rules

Input: a can-set rule $(\varphi_a,\ \varphi_u : a_t = v_t)$ where $\{{\varphi_a}'_1, \dots, {\varphi_a}'_j\} = \{{\varphi_a}'' \mid ({\varphi_a}'', {\varphi_u}'', a' = v') \wedge a_t \in Dom({\varphi_a}'')\}$
Output: a set of statements representing compilation of $[\![(\varphi_a,\ \varphi_u : a_t = v_t)_w]\!]$

```
if $* \wedge g\_\varphi_a \wedge [\![\varphi_u]\!] \wedge \neg(l\_a_t = v_t)$ then
    $l\_a_t := v_t$                                    ▷ Local memory update
    for ${\varphi_a}' \in \{{\varphi_a}'' \mid {\varphi_a}'' \in \Phi_a \wedge a_t \in Dom({\varphi_a}'')\}$ do
        $g\_{\varphi_a}' := [\![{\varphi_a}']\!]$              ▷ Conservative global memory update
    end for
end if
```

We are now ready to show the main property of the encoding $[\![I]\!]$.

Claim 1. *A satisfiability problem $I = (S, \varphi_t)$ is satisfiable if and only if an assertion of the program $[\![I]\!]$ fails.*

A failure in an assertion means that the local memory of the thread satisfies the target formula $[\![\varphi_t]\!]$. Since we work under the assumption that each thread satisfies the invariant stating that $l\_a_i = v_i$ if and only if the attribute a_i is set to value v_i to the user encoded by the thread ($UA(u)(a_i) = v_i$), we know that $UA(U) \vdash \varphi_t$. Formal proof is left as future work.

4.5.4 Bounded Analysis

Checking if a concurrent program fails any assertion is a well known problem in literature, and several solution are given, unfortunately even if the state space is finite, the verification problem is hard since complexity of program analysis and the exponential number of possible interleavings. For this reason several approximated approaches have been tried. We noticed, however, that all abstract-interpretation based solutions that we tested showed to be really inaccurate even in simple examples, since the high degree of nondeterminism of the encoded programs. For this reason we adopted another approach derived from Lazy-CSeq [39]: one of the state-of-the-art tool for BMC-based analysis of concurrent programs, that proved to be very efficient winning several times the concurrency category of SV-COMP. Such approach proved to be very precise and scalable in this specific setting.

Bounded sequentialization of the program.

The adopted approach mimics the one used in Lazy-CSeq: bounded sequentialization, but, before introducing it, let us make some preliminary considerations on the program generated by $[\![I]\!]$. The program obtained by $[\![I]\!]$ is a concurrent program where we have a set of threads each having a local variable l_a_i for each attribute a_i. In the INIT procedure we start initializing the global memory, then we spawn a thread for each user u_i, setting the local memory on the basis of the initial attribute-value assignment of the user $UA(u_i)$. At the end of INIT, each tread starts simulating users of the system. Every thread repeats forever a sequence of operations that consists of updating the global memory, performing any number of administrative operations and, finally, checking if they satisfies the φ_t with their local state. Notice that the loop body is executed atomically, so context switches could happen only at the end of each loop body.

Sequentialization In the sequentialization process defined thereafter we use, with abuse of notation, $[\![\cdot]\!]_{u_j}$, where $u_j \in U$ is a given user, to indicate the compilation $[\![\cdot]\!]$ on the memory of the user u_j ($\{l\_a^j_1, \dots, l\_a^j_m\}$).

The result of the sequentialization process is reported in table 4.7 and 4.8 and it is done through several operations. Firstly each local variable l_a_i of the thread of user u_j is added to the global state of the sequentialized program as $l\_a^j_i$. We grant that no thread function will use a variable not belonging to it since the only way to manage the local memory is through $[\![\cdot]\!]_{u_j}$. Then it is introduced a new initialization procedure INIT_SEQ (in table 4.7: rows 10-19) that, as the non sequential one, initializes the shared global memory, then instead of spawning threads with the initial local state, it initializes the sequentialized thread memory $l\_a^j_i := UA(u_j)(a_i)$.

We then add a new procedure USER_SEQ (in table 4.7: rows 20-30) that takes a user u as input and simulates the execution of one loop body of the thread representing the user u_j, but with one limitation: the number of administrative operation we perform is at most one, depending on the value of a program counter pc. Even if it seems that with this limitation we loose traces of the original program, this is not true thanks to the possibility to have an idle step where no administrative operations are executed. As an example consider a trace where the thread representing the user u_i was performing two administrative operations in the same loop iteration. This trace is identical to the one where u_i performs the first operation, then all other

Table 4.7 Sequentialized program

```
1: for $\varphi_a \in \Phi_a$ do
2:     decl $g\_\varphi_a$   ▷ one global variable for each administrative precondition
3: end for
4: for $u_j \in U$ do   ▷ user $u_j$ local variables
5:     for $a_i \in A$ do
6:         decl $l\_a_j^i$
7:     end for
8: end for
9: decl $pc$   ▷ one global variable as program counter

10: procedure INIT_SEQ()   ▷ sequential initialization procedure
11:     for $\varphi_a \in \Phi_a$ do
12:         $g\_\varphi_a := false$   ▷ conservative global memory initialization
13:     end for
14:     for $u_j \in U$ do   ▷ user $u_j$ local initialization
15:         for $a_i \in A$ do
16:             $l\_a_i^j := UA(u_j)(a_i)$
17:         end for
18:     end for
19: end procedure

20: procedure USER_SEQ($u_j$)   ▷ sequential encoding of individual user $u_j$
21:     for $\varphi_a \in \Phi_a$ do
22:         $g\_\varphi_a := g\_\varphi_a \vee [\![\varphi_a]\!]_{u_j}$   ▷ global memory update.
23:     end for
24:     for $r_k \in CS$ do
25:         if $pc = k$ then   ▷ if we want to apply $r_k$
26:             $[\![r_k]\!]_{u_j}$   ▷ encoding of the administrative rule $r_k$
27:         end if
28:     end for
29:     assert$(\neg[\![\varphi_t]\!]_{u_j})$   ▷ check whether $\varphi_t$ is satisfied by $u_j$
30: end procedure
```

users $u_j \neq u_i$ skips their turn performing no operations (except updating the global memory), and then u_i performs the second operation. This treatment introduces two additional advantages: the first one is that the shared global memory is updated more frequently, so it will better reflect the system status; the second is that we check more often if the assertion fails (proving satisfiability of the problem I).

The final step of the sequentialization is done adding a new MAIN procedure (in table 4.8) that performs the initialization phase calling INIT_SEQ and then moves, in an infinite loop, the sequentialized user thread functions. To

Table 4.8 Sequentialized main procedure

```
1: procedure MAIN()                                   ▷ main procedure
2:     INIT_SEQ()
3:     while $true$ do                                  ▷ for ever
4:         for $u_j \in U$ do
5:             $pc := *$        ▷ nondeterministically pick a can-set rule index
6:             USER_SEQ($u_j$)    ▷ simulate the step of $u$. It can be an idle step
7:         end for
8:     end while
9: end procedure
```

do so, for each user u_j in the system, it nondeterministically sets the program counter $pc := *$, and then it invokes the sequentialized thread procedure of user u_j.

The sequential program obtained simulates the round-robin schedules of $[\![I]\!]$ such that each iteration of the while loop represents a round where each thread is scheduled exactly once.

Set the bound. To perform the analysis of the sequentialized program using a bounded model checker approach, we need to unwind the program to a number of rounds k. Since the only unbounded loop in the program is the while in the MAIN procedure, we unwind it k times replacing it with k copy of its body obtaining the procedure MAIN_BOUND as shown in table 4.9. This step gives us a program that is sequentialized and bounded, so we can analyze it using a state of the art analysis tool such as CBMC [50] or ESBMC [56].

Table 4.9 Sequentialized and bounded MAIN procedure

```
procedure MAIN_BOUND()                         ▷ bounded main procedure
    INIT_SEQ()
    for $c = 1$ to $k$ do                        ▷ repeat the body $k$ times
        for $u_j \in U$ do
            $pc := *$        ▷ nondeterministically pick a can-set rule index
            USER_SEQ($u_j$)    ▷ simulate the step of $u$. It can be an idle step
        end for
    end for
end procedure
```

Unfortunately the bounding step will sacrifice one direction of the implication of claim introduced before. Indeed if the number of rounds k is too small we will not reach any violation that could be reached if we use a larger number of rounds.

Refinement Fortunately, since the state space of the problem is finite, to refine this analysis technique we have to increase the number k until (1) an assertion fails (2) we checked all reachable states. If we reach (1) we can prove the satisfiability of the problem, while (2) proves that the problem is unsatisfiable. To be sure we visited all reachable states we need to check that each state obtained at the end of each round has not been reached before. If for each trace there is at least one state repeated and we do not have any violation on the assertions, we can conclude that even with a larger k we will not reach any violations, and this proves that the satisfiability problem I is not satisfiable.

4.6 Over-approximated analysis

In this section we introduce a novel over-approximated approach for checking a satisfiability problem $I = (S, \varphi_t)$. The encoding $(\!|I|\!)$ represent a sequential program that is safe (i.e. cannot reach any failure state) only if I is not satisfiable. Such technique makes two over-approximating assumption.

The first assumption consists of considering always satisfied the administrative precondition of all can-set rules, so we can track only one user of the system.

The key idea of the second assumption, instead, is to encode the problem as a program where are tracked only administrative actions that set each attribute "for the last time", while assuming any possible action in the untracked operations. Indeed in all traces of a user u leading to the satisfaction of φ_t, we know that for each attribute that appears in the domain of the target formula, there should be an intermediate configuration where the attribute is set "for the last time", and after such set it will not change anymore. Using such property we put constraints on the execution of a trace only to find an ordering of "last set operations" that satisfies the target formula.

The advantage of this is that even if the length of a trace satisfying a formula could be exponential in the number of steps, we can track only the last assignment of the attributes appearing in the formula, so we need to track only $k = |Dom(\varphi_t)|$ steps.

4.6.1 Structure of the over-approximated encoding

The encoding tracks an execution of a nondeterministically picked user, leading to the satisfaction of the formula φ_t, tracking only the steps where each attributes $a_i \in Dom(\varphi_t)$ are assigned "for the last time". All other steps are abstracted assigning nondeterministic values to attributes, respecting the fact that if an attribute is assigned for the last time, it will not be re-assigned with different values later. At the end all attributes appearing in the domain of the target formula that have been changed during the trace must be set "for the last time" at some point of the run.

Variables. In the encoding we use several variables, listed in table 4.10. We have a variable u representing the chosen user that will be tracked. Then for each attribute $a_i \in A$: val_a_i is the variable storing the value of the attribute; set_a_i, is the boolean variable tracking if the attribute has been set "for the last time"; finally $changed\_a_i$ is the variable tracking the fact that an attribute has been changed at some point during the execution. In this encoding we refer to the set $\{val\_a_1, \ldots, val\_a_n\}$ as *state*.

Table 4.10 Over-approximation variables

```
1: decl $u$                        ▷ one variable representing the tracked user
2: for $a_i \in A$ do                                  ▷ for each attribute
3:     decl $val\_a_i$                       ▷ variable storing $a_i$ value (state)
4:     decl $set\_a_i$        ▷ variable stating if $a_i$ was set "for the last time"
5:     decl $changed\_a_i$            ▷ variable stating if $a_i$ has been changed
6: end for
```

Encoding. The encoding represents a number of untracked operations ending with a final tracked assignment. As said before we are not tracking all the operations of a trace, instead we are recording only the last assignment of each attributes used in the target formula.

To simulate all the untracked administrative operations occurring from each tracked step to the subsequent we assign a nondeterministic value to all the attribute *state* variables that could be changed. This clearly happens only if the attribute is not set for the last time ($set\_a_i = false$).

The structure of the over-approximated encoding of the instance $I = (S, \varphi_t)$ is a sequential program with two procedures defined as showed in Tables 4.11 and 4.13.

For the sake of readability we use the procedure call syntax even if is not supported by the language introduced in section 4.4. Indeed since all procedures are not recursive we can easily inline them.

INIT procedure. The INIT procedure (table 4.11) initializes all the variables of the program. First it nondeterministically picks the user that will be the one tracked during the run; secondly it initializes each attribute variable a_i with the value taken from the user-attribute assignment of the chosen user. This second set is done in three steps: (1,2) since a_i has not been set "for the last time" nor it has been changed yet, both set_a_i and $changed\_a_i$ takes value $false$; (3) the state of a_i is set assigning to val_a_i the value of the attribute taken from the initial user-attribute assignment.

Table 4.11 Over-approximation encoding: INIT procedure

```
1: procedure INIT()                          ▷ initialization procedure
2:     $u := *$                               ▷ nondeterministically pick a user
3:     for $a_i \in A$ do
4:         $set\_a_i := false$                ▷ $a_i$ is not set "for the last time"
5:         $changed\_a_i := false$            ▷ $a_i$ has not been changed yet
6:         $val\_a_i := UA(u)(a_i)$           ▷ set the initial state of the tracked user
7:     end for
8: end procedure
```

Formula compilation. Now we introduce the compilation of a formula. Let φ be a formula, $(\!|\varphi|\!)$, is an expression representing the interpretation of φ in the current *state* context. To do that we syntactically substitute all occurrences of any attribute a_i in φ with its corresponding *state* variable val_a_i.

Policy compilation. Before introducing the main part of the encoding we show how all the administrative rules contained in the policy are encoded. In details, given a policy P, we define its over-approximated compilation $(\!|P|\!)$ in terms of a *for loop* which simulates the set "for the last time" of any number of attribute using the administrative rules of P. The compilation, shown in table 4.12, works by iterating on each rule and applying its encoding. The result of the compilation of each individual rules is shown in the body of the loop given in table 4.12 and detailed below.

Can-Set compilation. The can-set rule $r = (\varphi_a, \varphi_u : a_t = v_t) \in CS$ is compiled into a conditional statement simulating the last set of attribute a_t

through r. The conditional statement guard is a conjunction of several expressions:

1. $*$: the non-deterministic choice operator, modeling that the rule may either be applied or not;
2. $(\!|\varphi_u|\!)$: the check of rule precondition under local state. This ensures that the local *state* of the program satisfies the precondition of the can-set rule;
3. $\neg(set\_a_t)$: checks that the attribute has not been set "for the last time" yet.

If the conjunction is satisfied, the consequent of the conditional is taken. To simulate the final set of attribute a_t to value v_t we perform three assignments: (1) we set variable val_a_t to v_t to reflect the attribute assignment; (2) we set variable set_a_t to *true* since the attribute now is set "for the last time"; (3) we set variable $changed\_a_t$ to *true* since the attribute's value has been changed.

Table 4.12 Compilation of the Administrative Rules

```
1: for $(\varphi_a, \varphi_u : a_t = v_t) \in CS$ do
2:     if $* \wedge (\!|\varphi_u|\!) \wedge \neg(set\_a_t)$ then
3:         $val\_a_t := v_t$                ▷ Update attribute value
4:         $set\_a_t := true$               ▷ Attribute is set "for the last time"
5:         $changed\_a_t := true$           ▷ Attribute has been changed
6:     end if
7: end for
```

MAIN procedure. The MAIN procedure (in table 4.13), is the central part of the encoding. In it the program is initialized calling procedure INIT, then the last assignment of all attributes $a_i \in Dom(\varphi_t)$ is simulated through the outermost loop (rows 3-11), finally, after the loop, the satisfiability check takes place.

The loop body is executed $k = |Dom(\varphi_t)|$ times, where each iteration represent any number of untracked administrative operations terminating with the potential set "for the last time" of one or more attributes. In each loop body two operations are performed: (1) each attribute that has not been not marked as final ($set\_a_i = false$) can be updated with a nondeterministic value to over-approximate all possibles untracked administrative actions happened since the last tracked point. If an attribute a_i is updated its

Table 4.13 Definition of the over-approximation encoding

```
1: procedure MAIN()                                   ▷ encoding of the run
2:     INIT()
3:     for $i = 1$ to $|Dom(\varphi_t)|$ do
4:         for $a_i \in A$ do
5:             if $* \wedge set\_a_i = false$ then   ▷ we want to update $a_i$ an we can
6:                 $val\_a_i := *$                    ▷ update state value
7:                 $changed\_a_i := true$             ▷ update changed value
8:             end if
9:         end for
10:        $(\!|P|\!)$                   ▷ encoding of the administrative rules of $CS$
11:    end for
12:    assume($(\!|\varphi_t|\!)$)       ▷ discard all the traces not satisfying $\varphi_t$
13:    assume($\forall a_i \in Dom(\varphi_t) : changed\_a_i \Rightarrow set\_a_i$)
14:    fail                          ▷ check if that statement is reachable or not
15: end procedure
```

changed_a$_i$ variable is set to *true* to record that it has been changed; (2) any number of attributes are set "for the last time" using administrative actions allowed by the policy of S, by the statements $(\!|P|\!)$ as explained before.

At the end of the loop, finally, the satisfiability check is simulated (rows 12-14).

Final check. The final check, that is executed at the end of the MAIN procedure, is used to check if φ_t is not satisfied by any configuration obtained from UA tracking only the final set of each attribute. As stated before, if the program is safe then the formula φ_t is never satisfiable. To perform such check need three statements:

1. **assume**($(\!|\varphi_t|\!)$): with this statement we require that the trace obtained by the main loop satisfies φ_t. In this way all traces not satisfying it are terminated without error. If φ_t is never satisfiable then all traces terminate silently and the program will never reach a failure state;

2. **assume**($\forall a_i \in Dom(\varphi_t) : changed\_a_i \Rightarrow set\_a_i$): with this condition we require that each attribute a_i that has been changed during the execution $changed\_a_i$, have also been set "for the last time" at some point set_a_i. A trace that does not satisfy such condition is silently terminated. If all traces are terminated, again, we know that it is impossible to reach a failure state;

3. **fail**: if this statement is reached, then the program enter in a failure state. This statement is reachable only if at least a trace satisfy both statements 1 and 2. Indeed in case that both checks holds we know that there exists one trace leading to satisfaction of φ_t where all the attribute that have been changed have also been set "for the last time". Since the over-approximating assumptions, in this case, we cannot conclude anything.

Since we are under an over-approximating assumption, impossibility to reach a **fail** statement implies that the given problem is not satisfiable, while the converse does not hold.

Claim 2. *Let $I = (S, \varphi_t)$ be a satisfiability problem. If $(\!|I|\!)$ cannot reach a **fail** statement then the problem I is not satisfiable.*

4.6.2 Examples

To better understand the proposed encoding, especially the two checks described before, consider the following situations. A trace that does not satisfy (1) is obviously not a candidate trace for the satisfiability problem instance I and it is obviously discarded. As an additional consequence if all traces are discarded at this point, then no traces satisfies I, and we can conclude that I is not satisfiable.

Table 4.14 Examples of safe policies

$U = \{u_1\}$ $A = \{a_1\}$ $UA = \{u_1 \rightarrow \{a_1 : 0\}\}$ $CS = \emptyset$ $\varphi_t = (a_1 = 1)$ Unsatisfiable problem 1	$U = \{u_1\}$ $A = \{a_1, a_2\}$ $UA = \{u1 \rightarrow \{a_1 : 0; a_2 : 0\}\}$ $CS = \{(true,\ a_2 \neq 1 : a_1 = 1),$ $(true,\ a_1 \neq 1 : a_2 = 1)\}$ $\varphi_t = (a_1 = 1) \wedge (a_2 = 1)$ Unsatisfiable problem 2
$U = \{u_1\}$ $A = \{a_1, a_2\}$ $UA = \{u1 \rightarrow \{a_1 : 1; a_2 : 0\}\}$ $CS = \{(true,\ a_1 = 1 : a_2 = 1)\}$ $\varphi_t = (a_2 = 1)$ Satisfiable problem 3	$U = \{u_1\}$ $A = \{a_1, a_2\}$ $UA = \{u1 \rightarrow \{a_1 : 0; a_2 : 0\}\}$ $CS = \{(true,\ a_1 = 1 : a_2 = 1)\}$ $\varphi_t = (a_2 = 1)$ Satisfiable problem 4 (spurious)

A trace that does not satisfy (2) is a trace where at least an attribute in $Dom(\varphi_t)$ has been changed, but has not been set "for the last time". If all traces does not satisfy (2) then in each was not possible to set "for the last time" at least one attribute that has been changed. This means that the satisfiability problem I is safe. A trivial example where (2) is never satisfied is shown in table 4.14 on the upper left. In it, the attribute $a_1 \in Dom(\varphi_t)$ has to be changed to a value 1 during the execution, but there are no can-set rules setting $a_1 = 1$. Another more interesting example is reported in table 4.14 on the upper right. In this case the encoded satisfiability problem requires both a_1 and a_2 to be set to value 1, but setting one to value 1 prevents the other to be set to such value. In the loop of the encoded MAIN procedure we can set a_1 or a_2 during the first iteration, but since the attribute set so far is then set "for the last time" in the second iteration we cannot change its value, and so the precondition of the can-set rule targeting the other attribute cannot be satisfied, and consequently we are not able to set to *true* the variable set_a_i. This makes the assumption (2) to fail in all traces proving the safety of the program and consequently the unsatisfiability of the given problem.
An example of a satisfiable problem is reported in table 4.14 on the lower left. Here all traces satisfying φ_t have to set the value of the attribute a_2 to 1. This is trivially possible and there is no need to change any attribute value nondeterministically. Indeed we can directly fire the can-set rule $(true,\ a_1 = 1 : a_2 = 1)$ setting a_2 "for the last time" since we satisfy its precondition. This will set $val\_a_2 = 2$ and $set\_a_2 = true$, thus the assumption (1) is satisfied as well as (2), since a_2 is the only attribute that changed his value from the initial configuration and it has been set "for the last time". This will make the **fail** statement reachable. Finally in table 4.14 on the lower right there is a problem for which the over-approximated analysis gives a spurious result. Indeed in the given problem, similarly to the previous one, we require a_2 to be set to value 1, and this is possible only if a_1 is set to 1. In this case, however, a_1 is initially set to 0 and we cannot set it to 1. Anyway, we can use the nondeterministic assignment to set a_1 to 1, even if it was not possible in a precise simulation. We can then use it to satisfy the precondition of the can-set rule $(true,\ a_1 = 1 : a_2 = 1)$ setting a_2 to 1 "for the last time". Even in this case (1) holds and (2) is also true since we are restricting the check only to attributes $a_i \in Dom(\varphi_t)$. We will show now how to refine the analysis to be able to prove unsatisfiable problems like this one.

4.6.3 Recursive refinement

We now introduce a refinement technique to enhance precision of the analysis reducing spurious results. To understand it better, consider a single step of the over-approximated analysis: in it we have a *state* obtained by the preceding steps UA'; then we have a nondeterministic assignment leading to the satisfaction of a precondition φ_u. To refine the analysis we can replace the nondeterministic assignment with the over-approximated encoding of a new satisfiability problem $I' = (((U, A, V, CS), UA'), \varphi_u)$. This obviously adds constraints to the analysis improving its precision. When we enter in refinement step (push), we save all the variables' value and create two new sets of variables $\{set\_a'_1, \dots, set\_a'_n\}$ and $\{changed\_a'_1, \dots, changed\_a'_n\}$ derived from the old one. When we exit (pop), instead, we restore the old values updating them to reflect the applied refinement step.

Here we show in detail the operation performed to enter and exit each refinement step.

Push. When we enter a new refinement step we perform three main operations: (1) we save all variables; (2) we create a new set of variables $\{set\_a'_1, \dots,$ $set\_a'_n\}$ where $set\_a'_i := set\_a_i$, since all the attributes that have already been set "for the last time" will not change; (3) we create a new set of variables $\{changed\_a'_1, \dots, changed\_a'_n\}$ where $changed\_a' := false$ since in the tracked step no attributes have been changed so far.

Pop. At the end of the refinement step we restore and update all the variables. This is done in two steps: (1) all the saved variables are restored; (2) all the variables $changed\_a_i$ are updated in order to reflect the fact that an attribute value could have been changed in the refinement step: $changed\_a_i :=$ $changed\_a_i \vee changed\_a'_i$. Notice that in this case we do not update any set_a_i variable since assignments "for the last time" are relative only of the refinement step, and not for the outer one.

With this refinement we can prove the example of table 4.14 on the lower right is not satisfiable. With a refinement step, indeed, we avoid that the attribute a_1 is ever set to 1, since there are no rules able to do so. With this the refinement step we are able to prove that it is not possible to satisfy the precondition of the required can-set rule.

We can then notice that this refinement procedure has a double benefit. The first one is that we constrain the nondeterminism of the solution, while the second benefit is given by the fact that the information on which attribute has been changed during the execution is more precise, thus the final check expression is enforced. Another interesting point is that we can apply this refinement technique in a recursive way achieving greater precision.

4.7 Pruning

In this section we introduce a set of pruning rules aimed at reducing the size of the policy removing attributes and rules not relevant for the given satisfiability problem. The rules defined here heavily inspired by the ones given in [33], [32], [45]. For this reason, in several occasions, we decided to keep the same name used in such works.

In the following section we refer to the given satisfiability problem with unbounded number of users as $I_\infty = (S_\infty, \varphi_t)$, where $S_\infty = ((U, A, V, CS, W, WA), UA)$.

The simplification is composed of two steps: a preprocessing phase and the pruning rules application.

4.7.1 Preprocessing preconditions

In this phase we preprocess all the preconditions using boolean algebra rules to remove all the formulas negations. The advantage of this is that each "normalized" formula is equivalent to the original, but it has no negation, so, while we apply the pruning rules, we can simplify its checks $a \sim v$ in an easier way.

4.7.2 Pruning rules

Here we introduce the pruning rules that are applied to the system and argue their correctness.

Backward slicing. With this rule we remove a set of *uninteresting* attributes from the policy. Intuitively an attribute a_u is *uninteresting* if it is never required for the satisfaction of the target formula φ_t, i.e. it is not required by any rule applied in any trace that leads to the satisfaction of φ_t. Here we propose an over-approximated technique to find *uninteresting* attributes. Let IA be the smallest set satisfying these two requirements:

1. $Dom(\varphi_t) \subseteq \mathcal{IA}$
2. $a \in \mathcal{IA} \implies \forall(\varphi_a, \varphi_u : a = v) : Dom(\varphi_a) \subseteq \mathcal{IA} \land Dom(\varphi_u) \subseteq \mathcal{IA}$

The set $\mathcal{IA}$ contains all the attributes that are possibly used by the rules required for the satisfaction of φ_t. We then simplify the original policy removing (1) all the attributes not in $\mathcal{IA}$, (2) all the rules having as target an attribute not in $\mathcal{IA}$ (3) the initial value of all roles not in IA from UA.

This procedure does not alter the satisfiability of the target formula φ_t, since we remove only attributes that are not in φ_t, nor in any preconditions of can-set rules required in a trace leading to satisfaction of φ_t.

Attribute used with value. In many of the rules defined thereafter we use a particular property that says if an attribute is ever required with a certain value.

Definition 24. *Let a an attribute and v be a value, we say that a is used with value v, say* $(a \downarrow v)$ *in the policy if and only if*

$$\exists AV, \varphi \in \Phi_a \cup \Phi_u \cup \{\varphi_t\} : (AV[a \rightarrow v] \vdash \varphi) \land (AV[a \not\rightarrow v] \nvdash \varphi)$$

Intuitively $a \downarrow v$ if there exists a precondition φ and an attribute-value assignment AV such that φ is satisfied when $a = v$, and not when $a \neq v$. This means that attribute a is required to have value v in at least the attribute assignment AV to satisfying φ.

Immaterial administrative precondition. As described in section 4.3, the number of user to track ($k + 1$ where $k = |\Phi_a^{\neq}|$) depend on the number of non equivalent administrative preconditions. To reduce such bound here we give three sufficient conditions, that grants that an administrative formula φ_a is always satisfied, making it *immaterial*. When an administrative formula is *immaterial* we substitute it with the constant *true*.

In the first sufficient condition states that if in the system we have at least $k + 2$ users with the same initial configuration UA, that satisfies the administrative precondition φ_a of a can-set rule, then φ_a is *immaterial*. Indeed, since we track just $k + 1$ users, we know that at least one user satisfying φ_a will be excluded from the satisfiability analysis and its configuration will not change during the trace leading to the satisfaction of φ_t, so such user can always trigger the can-set rule. The same applies even if there exists an attribute-value assignment $AV \in WA$ such that $AV \vdash UA$. Even in this case, since we can add to the system as many users we want with attribute-value assignment AV, we know that a user satisfying φ_a will always exists.

The second condition says that if in a can-set rule $(\varphi_a,\ \varphi_u : a = v)$ the administrative precondition φ_a is implied by the user precondition φ_u, then we know that every time we want to apply the can-set rule, the target user can also satisfy the administrative precondition, and then trigger it. This makes φ_a *immaterial*.

The key idea of the last condition is that we want to check if there exists a user u satisfying φ_a in the initial configuration $UA(u)$, that has no need to change its configuration to one $UA'(u)$ that does not satisfy φ_a anymore. If such user exists then we are guaranteed that it can always satisfy φ_a, making the precondition *immaterial*. To apply such condition we need to check if there exists a user u able to keep φ_a satisfied without preventing any another formula $\varphi \in (\Phi_a \cup \Phi_u \cup \{\varphi_t\}) \setminus \{\varphi_a\}$ from being satisfied.

This last condition is verified restricting the values that each attribute $a_i \in Dom(\varphi_a)$ can assume in the formula and checking if a user satisfies it. This is done creating a new formula φ' where each attribute check $a \sim v$ is enforced by an additional constraint defined as:

$$\varphi'' = \forall v \in Range(a_i) : (a \downarrow v) \Rightarrow (a = v)$$

This constraint restrict the possible values for the attribute a in order to guarantee that no other preconditions will fail if φ_a is satisfied. Indeed if a is required to have two different values v_i and v_j in two formulas φ_a and φ, then we cannot ensure that a user will always satisfy φ_a since it could have to set a to v_j to satisfy φ. Instead, if it is never used as values different than v_i, we are sure we can use $a = v_i$ in φ_a without preventing other formulas to be satisfied. Thus if there exists a user u such that it can satisfy φ' in the initial user-attribute assignment UA, we are guaranteed that it has no reason to change its user-attribute assignment making φ_a not satisfiable anymore. For this reason we say that φ_a can be always satisfied and so *immaterial*. Even if this condition seems very strict, we noticed that in practice it is very effective in spotting *immaterial* preconditions.

Infinite users BMC. In an AABAC$_\infty$ system we can apply this rule to remove administrative preconditions from the policy. Indeed if we have any number of classes of unbounded users, thanks to the theorem of section 4.3, in the satisfiability problem we track just $k+1$ of them. All the other infinite users that are not tracked, however, could be used to satisfy administrative preconditions since they will not change the satisfiability of the target formula φ_t. Intuitively in any trace, we can take an arbitrary number of users from any class of unbounded users, move them without changing the tracked users attribute-assignment until a set of administrative precondition $\Phi_\infty = \{\varphi_{a_1}, \ldots, \varphi_{a_n}\}$ is satisfied, then we start the tracked user trace with the advantage that the administrative formulas in Φ_∞ are always satisfied. A sufficient condition to prove this is to create a new AABAC satisfiability problem derived from I_∞: $I_{\varphi_{a_i}} = (((U', A, V, CS), UA'), \varphi_{a_i})$ for each administrative preconditions $\varphi_{a_i} \in \Phi_a$, where U' and UA' are created taking for each $AV \in WA$ a set U_{AV} of $k+1$ users from W, adding them to U' and updating UA' such that for each $u_j \in U_{AV}$, $UA'(u_j) = AV$. For each of the so obtained satisfiability pro satisfiability problem $I_{\varphi_{a_i}}$ we check if $[\![I_{\varphi_{a_i}}]\!]$ is satisfiable or not as shown in section 4.5. If $[\![I_{\varphi_{a_i}}]\!]$ is satisfiable then we can remove φ_a from the policy and substitute it with the constant *true*.

Irrelevant attributes. An attribute a is *irrelevant* if every can-set rule can be fired regardless the value assumed by a. This happens if every time we require a value v for it, we are also able to set a to v. When an attribute is *irrelevant* we can remove it from the policy without affecting the satisfiability of the target formula φ_t.

When we spot an *irrelevant* attribute we can remove it from the policy with the following steps: (1) all the preconditions sub-expressions where the attribute a is used ($a \sim v$) are simplified substituting to them the constant *true*; (2) all can-set rules targeting a from CS are removed; (3) the attribute a is removed from the initial configuration UA. While steps 2 and 3 are straightforward, we know that we can apply the step 1 since: (a) we know that every time we need the attribute with a specific value, we are able to set it to such value, so all checks on it will succeed; (b) we work with "normalized" preconditions, so we do not have any negation in the formula and, for this reason, in order to consider a check always satisfied, we can simply replace it with the constant *true*.

The condition to mark an attribute a as *irrelevant* is defined in table 4.15. It is composed of two parts: the former predicates on administrative usages of a, while the latter on regular ones. In the first part, regarding the administrative usage of a, we check if there exists a scenario where: (1) a user u has an attribute-value assignment $AV = UA(u)$ such that a has not the value v in it ($AV(a) \neq v$), (2) the administrative precondition φ_a is not satisfied by AV ($AV \nvdash \varphi_a$), but (3) it is satisfied after setting a to v ($AV[a \to v] \vdash \varphi_a$), (4) all can-set rules ($(\varphi_a{}', \varphi_u{}' : a = v)$) setting a to v with exception of r, since we are trying to trigger it, are not fireable by u ($(AV \nvdash \varphi_a{}') \vee (AV \nvdash \varphi_u{}')$). If the such scenario exists we cannot remove a, since in at least one case it cannot be set to the required value, and the condition fails. The second part of the condition, regarding the regular usage of a, is similar to the first one, but in it there is an additional user u_a involved that has an attribute-value assignment AV_A able to trigger the can-set requiring a with value v ($AV_A \vdash \varphi_a$), an that can trigger himself the administrative precondition of the rule assigning a to v ($AV_A \vdash \varphi_a{}'$).

Table 4.15 Irrelevant condition for attribute a.

$$
\begin{array}{lll}
\forall\, v \in Range(a) : & & \\
\quad \nexists AV : & AV(a) \neq v \wedge & \\
& \forall\, r = (\varphi_a,\ \varphi_u : a' = v') \in CS : & \\
& \quad AV[a \to v] \vdash \varphi_a \wedge (\neg AV \vdash \varphi_a) \wedge & \\
& \quad \bigwedge_{(\varphi_a{}',\ \varphi_u{}' : a = v) \in CS \setminus r} : & \\
& \qquad (\neg(AV \vdash \varphi_a{}') \vee \neg(AV \vdash \varphi_u{}')) & \\
\wedge\, \forall\, v \in Range(a) : & & \\
\quad \nexists AV, AV_A : & AV(a) \neq v \wedge & \\
& \forall\, r = (\varphi_a,\ \varphi_u : a' = v') \in CS : & \\
& \quad AV[a \to v] \vdash \varphi_u \wedge (\neg AV \vdash \varphi_u) \wedge & \\
& \quad AV_A \vdash \varphi_a \wedge & \\
& \quad \bigwedge_{(\varphi_a{}',\ \varphi_u{}' : a = v) \in CS \setminus r} : & \\
& \qquad (\neg(AV_A \vdash \varphi_a{}') \vee \neg(AV \vdash \varphi_a{}') \vee \neg(AV \vdash \varphi_u{}')) &
\end{array}
$$

Rule merge. Given two can-set rules $r1 = (\varphi_a,\ \varphi_u : a = v)$ and $r2 = (\varphi_a{}',\ \varphi_u{}' : a = v)$ we can merge them if (1) they have the same target, (2) the administrative precondition is equivalent ($\varphi_a \equiv \varphi_a{}'$). The first condition ensures us that all users satisfying the former administrative precondition also satisfies the latter. If such condition holds, we can combine $r1$ and $r2$ in a new rule $r3 = (\varphi_a,\ \varphi_u \vee \varphi_u{}' : a = v)$.

Implied rules. Given two rules $r1 = (\varphi_a, \varphi_u : a = v)$ and $r2 = (\varphi_a{}', \varphi_u{}' : a = v)$ with the same target. We say that $r1$ implies $r2$ if every time a user u is able to trigger $r2$, than it can also fire $r1$. Here we give a sufficient condition to check if $r1$ implies $r2$.

$$\forall AV, AV_A : (AV_A \vdash \varphi_a{}' \wedge AV \vdash \varphi_u{}') \implies (AV_A \vdash \varphi_a \wedge AV \vdash \varphi_u)$$

Indeed if such condition holds we know that every time we are ready to fire $r2$, we are also able to trigger $r1$. For this reason we can remove $r2$ from CS.

Precondition simplification. With this rule we give a sufficient condition to simplify a precondition in order to remove all disjunction branches that are not necessary. Intuitively a disjunction branch is not necessary if no attribute-value assignments satisfies it, or, when they do, even the other branch is satisfied. If one branch of a disjunction is unnecessary we can replace the whole disjunction with the other branch.

More formally let φ be a formula, having a disjunction $\varphi_1 = (\varphi_2 \vee \varphi_3)$ as sub-formula. Let φ' be another formula that is equal to φ with the exception that the sub-formula φ_1 is substituted by $\varphi_1' = (\varphi_2 \vee false)$. Then if there exists no attribute-value assignment AV such that $(AV \vdash \varphi) \wedge (AV \vdash \varphi')$ then we know that φ_3 is never required for the satisfiability of φ, thus we can replace φ with φ' removing φ_3. The same clearly applies for φ_2.

Even though this rule seems pretty simple and not very effective, in section 4.7.4 we show how it could be strengthened becoming dramatically effective.

Not fireable rules. This rule gives us a way to prove that a rule $r = (\varphi_a, \varphi_u : a = v)$ could never be fired. Intuitively a rule could be fired if there exists two users satisfying respectively φ_a and φ_u. Checking if this condition is true is hard as solving the original satisfiability problem, but we can use the over-approximated analysis introduced in section 4.6 to check if φ_a and φ_u can be satisfied. To do so, given the AABAC system $((U', A', V', CS'), UA') = \lfloor S_\infty \rfloor$ derived by the original AABAC$_\infty$ one, we create two new satisfiability problems $I_a = (((U', A', V', CS' \setminus \{r\}), UA'), \varphi_a)$ and $I_u = (((U', A', V', CS' \setminus \{r\}), UA'), \varphi_u)$ and check if $(\!|I_a|\!)$ or $(\!|I_u|\!)$ are not satisfiable. If they are not, we know that in any trace we can not fire r. Notice that we remove r from the policy of both I_a and I_u because we are tracing all executions leading to the first activation of rule r.

4.7.3 Pruning algorithm

The pruning algorithm for an administrative attribute-based access control system takes a satisfiability problem I as input, it preprocess it, and then applies all the aforementioned rules until no further simplifications are possible as we reached a fixpoint. It then returns the simplified problem I'. Notice that since in all pruning rules we remove attributes, rules or we simplify the precondition, the size of the satisfiability problem is reduced (or not altered) at any step. Even though the fixpoint is not unique, since the order of application of the rules changes the result, we are ensured that it will converge.

4.7.4 Restricting attribute-value assignments.

All the pruning rules described so far proved to be effective in reducing the size of the policy without affecting the satisfiability of the formula φ_t. However in most of them (except the last one), we limit ourself to check if there exists any attribute-value assignment AV, and we do not pose any reachability restriction on them. Unfortunately often it happens that the required attribute-value assignment AV is spurious, i.e. not reachable from the initial user-attribute assignment UA. This severely limits the effectiveness of the pruning. Taking inspiration from the "Not fireable rules" condition, we make an additional refinement step by lifting many pruning rules to a semantic level. The idea is to restrict the existential quantification over the possible attribute-value assignment AV to only the reachable ones. Even in this case a precise restriction is not feasible since it would be as hard as solving the original satisfiability problem, but the good news is that we can use the over-approximated analysis technique introduced in section 4.6 to perform the restriction. Indeed, having a pruning rule that is applicable if a certain precondition φ holds, we can encode a new satisfiability problem $I' = (((U', A', V', CS'), UA'), \varphi)$, where $((U', A', V', R'), UA') = \lfloor S_\infty \rfloor$, and check if it is satisfiable using over-approximation $(\!|I'|\!)$. To avoid useless computation we apply this refinement step only when all the rules described so far are not applicable, thus increasing scalability of the pruning.

Table 4.16 Pruning standard ARBAC

Name	Original policy			Pruned policy			Time
	#Users	#Attr	#Rules	#Users	#Attr	#Rules	
Bank	2000	533	5142	0	0	0	39.01 s
Bank	2000	533	5142	0	0	0	37.93 s
Bank	2000	533	5142	2	2	1	39.77 s
Bank	2000	533	5142	2	2	1	8.39 s
Hospital	1093	15	25	0	0	0	0.01 s
Hospital	1093	15	25	0	0	0	0.01 s
Hospital	1093	15	25	2	2	1	0.01 s
Hospital	1093	15	25	1	1	1	0.02 s
University	944	34	449	0	0	0	0.02 s
University	944	34	449	4	4	2	0.03 s
University	944	34	449	0	0	0	0.20 s
University	944	34	449	9	9	3	0.25 s

4.8 Experimental results

We implemented all the techniques for the analysis of sections 4.5 and 4.6 as well as the pruning algorithm described in section 4.7 in a experimental tool: VACSAT. To implement the pruning, as well as both analysis technique, we used a set of state of the art SMT solvers for quantifier-free boolean and bitvectors logic formulas (QF_BV) such as YICES [29], Z3 [26], Boolector [60], MathSat [22]. We tested VACSAT against three different set benchmarks taken from the literature: the first is the one used in [31, 71], the second is the set of policies generated by the reduction from workflow systems described in chapter 2 where the pruning technique of section 2.4 has not been applied. The last set contains ARBAC policies with hierarchies where no flattening have been applied. The first and the second sets contain only ARBAC policies, while the third set contains an AABAC policy, obtained by the translation of the hierarchical ARBAC, where each attribute has a boolean domain. This because we decided to avoid randomly-generated test cases, since these often do not represent realistic systems. Results of our experiments are reported in tables 4.16, 4.17, 4.18 and 4.19. For the analysis we choose YICES as SMT backend, since it was the one with better performances among the others on our benchmarks. However even if YICES is the default backend, VACSAT leaves the choice to select another one of the aforementioned solvers. All the tests are run on a Linux desktop with an i7-6700 processor and 12 GB of RAM.

Table 4.17 Pruning workflow

Name	Original policy			Pruned policy			Time
	#Users	#Attr	#Rules	#Users	#Attr	#Rules	
Choice	121	55	66	37	43	43	0.84 s
Choice	121	55	66	78	42	38	0.94 s
Choice	121	55	66	0	0	0	0.12 s
Choice	121	46	57	23	34	34	0.42 s
Choice	121	46	57	0	0	0	0.08 s
Choice	121	46	57	49	27	23	0.37 s
First aid	121	49	65	25	37	37	0.76 s
First aid	121	49	65	38	33	33	0.73 s
First aid	121	49	65	0	0	0	0.15 s
Sequential	121	56	66	74	38	39	1.64 s
Sequential	121	55	64	80	43	43	1.51 s
Sequential	121	55	64	0	0	0	0.08 s
Sequential	121	47	57	56	30	31	1.05 s
Sequential	121	46	55	65	34	34	0.85 s
Sequential	121	46	55	0	0	0	0.08 s
Parallel	121	56	66	74	38	39	1.55 s
Parallel	121	56	66	0	0	0	0.12 s
Parallel	121	55	64	50	27	23	1.64 s
Parallel	121	46	55	65	34	34	0.88 s
Parallel	121	46	55	0	0	0	0.06 s
Parallel	121	47	57	0	0	0	0.08 s

4.8.1 VACSAT tool

The implemented tool, VACSAT, consists of approximatively 5000 lines of C++ code. It could take as input an AABAC$_\infty$ satisfiability problem, or an ARBAC role reachability one. If the input is an ARBAC role reachability problem, the tool translates it to an AABAC instance, as described in section 4.2.4. Then the AABAC$_\infty$ satisfiability problem is simplified applying all the pruning rules defined in 4.7 until no more simplifications could be applied. VACSAT then checks if the problem is satisfiable by: firstly applying the over-approximated analysis with 3 levels of refinement, then, if the given problem is not proved unsatisfiable, it performs a bounded analysis with 30 rounds as bound.

Analysis. In the first versions of VACSAT, to perform both precise and over-approximated analysis, we were generating a C program, as described in sections 4.5 and 4.6. Then we were discharging the so generated program to CBMC [50] or ESBMC [56], two well known state of the art checkers for C programs. Unfortunately we noticed that this approach was introducing a huge overhead, since both ESBMC and CBMC were applying lots of general

Table 4.18 Pruning ARBAC with hierarchy

Name	Original policy			Pruned policy			Time
	#Users	#Attr	#Rules	#Users	#Attr	#Rules	
Healthcare	100	15	27	2	2	1	0.01 s
Healthcare	100	15	27	2	2	1	0.01 s
Healthcare	100	15	27	2	2	1	0.01 s
Healthcare	100	15	27	2	2	1	0.01 s
Healthcare	100	15	27	2	2	1	0.01 s
Healthcare	100	15	27	2	2	1	0.01 s
University	101	36	59	2	2	1	0.05 s
University	101	36	59	2	2	1	0.05 s
University	101	36	59	2	2	1	0.05 s
University	101	36	59	5	4	3	0.01 s
University	101	36	59	4	4	3	0.01 s
University	101	36	59	5	4	2	0.01 s
University	101	36	59	4	4	3	0.01 s
University	101	36	59	2	2	1	0.08 s
University	101	36	59	2	2	1	0.06 s
University	101	36	59	2	2	1	0.06 s
University	101	36	59	4	4	3	0.05 s
University	101	36	59	4	4	3	0.01 s
University	101	36	59	4	4	2	0.05 s
University	101	36	59	5	4	3	0.05 s

optimizations that were not required in our specific case. To overcome this overhead we decided to implement the analysis directly encoding the problem as an SMT formula, and then discharging it to the SMT backend. This improved dramatically the performance of the analysis by several order of magnitude.

Pruning. The simplification procedure, described in section 4.7, has been implemented through a set of queries that are solved by the chosen SMT backend. All the logic constructs that are not fully supported by some of the backends, such as the universal quantifier, are rewritten in terms of supported constructs. The order of application of the simplification rules has been chosen after an experimental evaluation to enhance scalability of the analysis, thus slower rules are applied only when the simpler, but faster ones are not applicable anymore.

4.8.2 Results discussions

Pruning evaluation. Tables 4.16, 4.17 and 4.18 reports in details times and results of the simplification process in the three different sets of benchmarks. We can notice that the simplification process is dramatically effective on the

Table 4.19 Total analysis time

	Name	Without pruning	With pruning	Res		Name	Without pruning	With pruning	Res
ARBAC	Bank	-	38.90 s	N	WORKFLOW	Parallel	1.94 s	3.22 s	S
	Bank	-	37.99 s	N		Parallel	0.94 s	0.12 s	N
	Bank	-	39.81 s	S		Parallel	26.68 s	3.45 s	?
	Bank	-	8.27 s	S		Parallel	0.95 s	1.66 s	S
	Hospital	0.3 s	0.01 s	N		Parallel	0.60 s	0.06 s	N
	Hospital	0.02 s	0.01 s	N		Parallel	0.62 s	0.08 s	N
	Hospital	0.97 s	0.01 s	S	HIERARCHY	Healthcare	0.06 s	0.01 s	S
	Hospital	1.74 s	0.02 s	S		Healthcare	0.05 s	0.01 s	S
	University	0.11 s	0.02 s	N		Healthcare	0.07 s	0.01 s	S
	University	2.25 s	0.03 s	S		Healthcare	0.08 s	0.01 s	S
	University	0.02 s	0.20 s	N		Healthcare	0.01 s	0.04 s	S
	University	12.20 s	0.24 s	S		Healthcare	0.05 s	0.01 s	S
WORKFLOW	Choice	1.48 s	2.66 s	S		University	0.44 s	0.05 s	S
	Choice	1.44 s	2.80 s	S		University	0.44 s	0.05 s	S
	Choice	0.91 s	0.13 s	N		University	0.43 s	0.05 s	S
	Choice	1.60 s	1.20 s	S		University	0.77 s	0.02 s	S
	Choice	0.63 s	0.08 s	N		University	0.64 s	0.02 s	S
	Choice	1.31 s	0.70 s	S		University	0.77 s	0.01 s	S
	First aid	3.03 s	2.10 s	S		University	0.67 s	0.02 s	S
	First aid	3.06 s	1.44 s	S		University	0.45 s	0.10 s	S
	First aid	0.71 s	0.15 s	N		University	0.46 s	0.06 s	S
	Sequential	15.10 s	3.12 s	S		University	0.48 s	0.07 s	S
	Sequential	1.51 s	3.22 s	S		University	0.80 s	0.05 s	S
	Sequential	0.58 s	0.07 s	N		University	0.66 s	0.02 s	S
	Sequential	1.49 s	1.67 s	S		University	0.82 s	0.05 s	S
	Sequential	1.69 s	1.58 s	S		University	0.68 s	0.05 s	S
	Sequential	0.88 s	0.08 s	N		-	-	-	-

ARBAC policies of table 4.16, as well as in the ARBAC with hierarchy ones reported in table 4.18, returning in all cases policies with less than 4 can-set rules. The pruning technique, instead, was not so effective in the case of some policies reported in table 4.17. This is because the reduction from workflow to role reachability introduces lots of synchronizations that are hard to simplify with the approaches given in section 4.7. Notice that, after the simplification phase, no rules nor users are left in the system, this happens when the problem is not satisfiable. We can also notice that the times for the simplification process are negligible in almost all cases, with exception to the Bank policies of table 4.16. This slow simplification time is due to the great size in terms of attributes and rules of those policies, anyway in table 4.19 we can notice that those policies are not tractable without the simplification step. Finally is worth of mention that even if pruning times are greater than the ones obtained by previous works [33], our simplification technique works on a more general model where we need to use a SMT solver, since a faster syntactical approach is not feasible.

Analysis evaluation. Table 4.19 reports times and results of the whole analysis for each set of benchmarks with and without the pruning phase. Here we can notice that the analysis is able to prove that the problems are satisfiable (S) or not satisfiable (N) in all cases except one workflow instance. We can also notice that the simplification process enhances the scalability of the analysis in almost all examples, and when this is not true, the analysis time is however very small. Another interesting point is that, as mentioned before, in all cases where the problem is not satisfiable, the time for the analysis with pruning phase roughly coincides with the simplification time. This is because the simplification phase is able to remove all the can-set rules from the policy, thus trivially proving the unsatisfiability of the problem.

4.9 Conclusions

In this chapter we provided a novel *specification* of an administrative version of Attribute-Based Access Control Model system (AABAC) and we showed that it is more general than ARBAC systems. We devised a technique to reduce the number of users in a system that could also be exploited to reduce systems with unbounded number of users to bounded ones. We also gave two techniques for the *verification* of satisfiability problems for AABAC systems. Moreover we proposed a simplification technique to reduce the size of the satisfiability problem without affecting its satisfiability. Finally we implemented all the aforementioned techniques in the VACSAT tool, and we tested its effectiveness against a large set of benchmarks from the literature.

Future work. As future works, we plan to formally prove correctness of the pruning and of the presented analysis techniques. Moreover we plan to extend the over-approximated analysis technique in two ways: (1) we want to track all the users in the system, thus enabling the possibility to check even the administrative preconditions; (2) we want to find a way to reduce the number of spurious results by dynamically selecting and refining all the inaccurate steps. We also want to produce a human readable counterexample for the satisfiability problem even in presence of pruning. Finally we would like to extend the model in order to support more sophisticated administrative policies.

Chapter 5

Conclusions

In this thesis we attempted to provide formal methods for the *specification* and *verification* of access control related systems. The contributions we gave are the following:

- a model for workflows based on stable event structures where the security is enforced by an Administrative Role-Based Access Control system with the addition of binding and separation of duty constraints;
- a notion of security against collusion on workflow systems, and a static analysis technique to state if a workflow satisfies it;
- a novel administrative model for Attribute-Based Access Control systems. We proved also that it is more general than the ARBAC one, and we defined the satisfiability problem for AABAC systems that can be exploited to verify if the given policy satisfies many security related properties;
- a model for AABAC systems with unbounded number of users and we gave a technique to reduce it to one with bounded number of users;
- two scaleable refineable approximated *verification* techniques, one under-approximated and the other over-approximated, to solve the satisfiability problem, and a pruning technique aimed at reducing the size of the problem before starting the verification.

Each contribution has been formally introduced and then implemented in a tool to verify the effectiveness of the techniques for the analysis, and have been tested against many realistic case studies from different domains.

Future work. As future works, we plan to continue the research in workflow systems in order to extend our theory to support workflows including loops. Furthermore we plan to extend the work by designing and implementing a translator from high-level business process description languages such as BPMN or BPEL into event structures.

We also plan to complete the work on Administrative Attribute-Based Access Control by formally proving soundness of the pruning and the presented analysis techniques. Moreover we plan to extend the over-approximated analysis technique in order to find a way to reduce the number of spurious results by dynamically selecting and refining the steps that are inaccurate. We want to produce a human readable counterexample for the satisfiability problem even in presence of pruning. Finally we want to extend the model in order to support more sophisticated administrative policies.

Appendices

Appendix A

Fine-grained Detection of Privilege Escalation Attacks on Browser Extensions

A.1 Introduction

Browser extensions customize and enhance the functionalities of standard web browsers by intercepting and reacting to a number of events triggered by navigation, page rendering or updates to specific browser data structures. While many extensions are simple and just installed to customize the navigation experience, other extensions serve security-critical tasks and have access to powerful APIs, providing access to the download manager, the cookie jar, or the navigation history of the user. Hence, the security of the web browser (and the assets stored therein) ultimately hinges on the security of the installed browser extensions. Just like browsers, extensions typically interact with untrusted and potentially malicious web pages: thus, all modern browser extension architectures rely on robust security principles, such as *privilege separation* [67].

Browser Extension Architecture. Privilege separated architectures require programmers to structure their code in separated modules, running with different privileges. In the realm of browser extensions, privilege separation is implemented by structuring the extension in two different types of components: a privileged *background page*, which has access to the browser APIs and runs isolated from web pages; and a set of unprivileged *content scripts*, which are injected into specific web pages, interact with them and are at a higher risk of attacks [8, 20]. The permissions available to the background page are defined at installation time in a manifest file, to limit the dangers connected to the compromise of the background page. Content scripts interacting with different web pages are isolated one from each other by the same-origin policy of the browser, while process isolation protects the background page. The message passing interface available to extensions only allows the exchange of serialized JSON objects[1] between different components, hence pointers cannot cross trust boundaries.

Language Support for Privilege Separation. We are interested here in understanding to which extent current browser extension development frameworks, such as the Google Chrome extension APIs, naturally support privilege separation and comply with the underlying security architecture. Worryingly, we notice that in these frameworks a single privileged module typically offers a unified entry point to security-sensitive functionalities to all the other extension components, even though not all the components need to access the same functionalities and different trust relationships exist between different components.

To make matters worse, current programming patterns adopted in browser extensions do not safeguard the programmer against *compromised* components, even though the underlying privilege separated architecture was designed with compromise in mind. Compromise adds another layer of complexity to security-aware extension development, since corrupted extension components may get access to surprisingly powerful privileges.

A.1.1 Motivating Example

We illustrate our argument with a simple, but realistic example, inspired by one of the many cookie managers available in the Chrome Web Store (e.g., `EditThisCookie`). Consider an extension which allows users to add, delete or modify any cookie stored in the browser through an intuitive user interface. Additionally, it allows web pages to specify a set of security policies for the cookies they register: these client-side security policies are enforced by the extension and can be used to significantly strengthen web authentication [14, 15].

The extension is composed of three components: two content scripts C and O, and a background page B. The background page is given the `cookies` permission, which grants it access to the browser cookie jar. The content

[1] `http://json.org`

script O is injected in the `options.html` page packaged with the extension and it provides facilities for cookie editing; when the user is done with his changes, O sends B a message and instructs it to update the cookie jar. The content script C, instead, is injected in the DOM of any HTTPS web page P opened by the browser: it is essentially a proxy, which forwards to B the security policies specified by P using the message passing interface. The messages sent by P are extended by C with an additional information: the website which specified the security policy.

A possible run involving all the described components is the following, where the last message is triggered by a user click:

```
P → C  :  {tag: "policy", spec: "read-only"}
C → B  :  {tag: "policy", site: "paypal.com", spec: "read-only"}
O → B  :  {tag: "upd",
             ck: {dom: "a.com", name: "res", value: "1440x900"}}
```

Using the Google Chrome extension API, the components are programmed in JavaScript, typically by registering appropriate listeners for incoming messages. For instance, the content script C can be programmed as follows:

```
window.addEventListener("message", function(event) {
  /* Accept only internal messages */
  if (event.source != window) { return; }
  /* Get the payload of the message */
  var obj = event.data;
  /* Extend the message with the site and forward it */
  obj.site = window.location.hostname;
  chrome.runtime.sendMessage (obj);
}, false);
```

Web pages can communicate with C by using the `window.postMessage` method available in JavaScript, thus opting-in to custom client-side protection.

The background page B, instead, is typically programmed as follows:

```
chrome.runtime.onMessage.addListener(
  function (msg, sender, sendResp) {
    /* Handle the reception of new policies */
    if (msg.tag == "policy") {
      /* Store a new (valid) policy for the site */
      if (is_valid (msg.spec))
        localStorage.setItem (msg.site, msg.spec);
      else console.log ("Invalid policy");
    }
    /* Handle requests for cookie updates */
    else if (msg.tag == "upd") {
      chrome.cookies.set (msg.ck);
    }
    else console.log ("Invalid message");
});
```

This tag-based coding style featuring a single entry point to the background page is very popular, since it is easy to grasp and allows for fast prototyping, but it also fools programmers into underestimating the attack surface against the extensions they write. In this example, a malicious web page can compromise the integrity of the cookie jar by exploiting the poorly programmed content script C through the following method invocation:

```
window.postMessage ({tag: "upd", ck: {dom: "google.com",
                              name: "SID", value: "aQe73ajs..."}});
```

This allows the web page to carry out dangerous attacks, like session fixation or login CSRF on arbitrary websites [15]. The issue can be rectified by including a *sanitization* in the code of C and by ensuring that only messages with the `"policy"` tag are delivered to the background page.

The revised code is more robust than the original one and it safeguards the extension against the threats posed by malicious (or compromised) web pages. Unfortunately, it does not yet protect the background page against a compromised content script: if an attacker is able to exploit a code injection vulnerability in C, he may force the content script into deviating from the intended communication protocol. Specifically, an attacker with scripting capabilities in C may forge arbitrary messages to the background page and taint the cookie jar.

A much more robust solution then consists in introducing two distinct communication ports for C and O, and dedicating these ports to the reception of the two different message types (see Section A.5). This is relatively easy to do in this simple example, but, in general, decoupling the functionalities available to the background page to shield it against privilege escalation is complex, since n different content scripts or extensions may require access to m different, possibly overlapping sets of privileged functionalities.

A.1.2 Contributions

Our contributions can be summarized as follows:

1. we model browser extensions in a formal language that embodies the essential features of JavaScript, together with a few additional constructs dealing with the security aspects specific to the browser extension architecture;

2. we formalize a fine-grained characterization of the privileges which can be escalated by an active opponent through the message passing interface, assuming the compromise of some untrusted extension components;

3. we propose a flow logic specification estimating the safety of browser extensions against the threats of privilege escalation and we prove its soundness, despite the best efforts of an active opponent. We show how the static analysis works on the example above and supports its secure refactoring;

4. we present CHEN (CHrome Extension aNalyser), a prototype tool that implements our flow logic specification, providing an automated security analysis of existing Google Chrome extensions. The tool opens the way to an automatic security-oriented refactoring of existing extensions. We show CHEN at work on ShareMeNot [66], a real extension for Google Chrome, and we discuss how the tool spots potentially dangerous programming practices.

A.2 Related Work

Browser Extension Security. Carlini *et al.* performed a security evaluation of the Google Chrome extension architecture by means of a manual review of 100 popular extensions [20]. Liu *et al.* further analysed the Google Chrome extension architecture, highlighting that it is inadequate to provide protection against malicious extensions [53]. Guha *et al.* [35] proposed a methodology to write provably secure browser extensions, based on refinement typing; the approach requires extensions to be coded in Fine, a dependently-typed ML dialect. Karim *et al.* developed Beacon, a static detector of capability leaks for Firefox extensions [49]. A capability leak happens when a component exports a pointer to a privileged piece of code. These leaks violate the desired modularity of Firefox extensions, but they cannot be directly exploited by content scripts, since the message passing interface prevents the exchange of pointers. Finally, information flow control frameworks have been proposed for browser extensions [27, 6].

Privilege Escalation Attacks. Privilege escalation attacks have been extensively studied in the context of Android applications, starting with [25, 63]. Fragkaki *et al.* formalized protection against privilege escalation in Android applications as a noninterference property, which is then enforced by a dynamic reference monitor [34]. Bugliesi *et al.* presented a stronger security notion and discussed a static type system for Android applications, which provably enforces protection against privilege escalation [16]. The present work generalizes both these proposals, by providing a fine-grained view of the privileges leaked to an arbitrarily powerful opponent. Akhawe *et al.* [3] pointed out severe limitations in how privilege separation is implemented in browser extension architectures. Their work has been very inspiring for the present paper, which provides a formal counterpart to many interesting observations contained therein. For instance, [3] defines *bundling* as the collection of disjoint functionalities inside a single module running with the union of the privileges required by each functionality. Our formal notion of privilege leak captures the real dangers of permission bundling.

Formal Analysis of JavaScript. Maffeis *et al.* formalized the first detailed operational semantics for JavaScript [54] and used it to verify the (in)security of restricted JavaScript subsets [55]. Jensen *et al.* proposed an abstract interpretation framework for JavaScript in the realm of type analysis [43]. Guha *et al.* defined λ_{JS} as a relatively small core calculus based on a few well-understood constructs, where the numerous quirks of JavaScript can be encoded with a reasonable effort [36]. The adequacy of the semantics has been assessed by extensive automatic testing. The calculus has been used to support static analyses to detect type errors in JavaScript [37] and to verify the correctness of JavaScript sandboxing [62]. We also develop our flow analysis on top of λ_{JS}, extending it to reason about browser extension security. An alternate solution would have been to base our work on S5 [61]. This approach would have allowed to analyse browser extensions using ECMA5-specific features, but at the cost of significantly complicating the formal development.

A.3 Modelling Browser Extensions

Our language embodies the essential features of JavaScript, formalized as in λ_{JS} [36], up to a number of changes needed to deal with the security aspects specific to the browser extension architecture. In our model, several expressions run in parallel with different permissions and are isolated from each other: communication is based on asynchronous message exchanges.

A.3.1 Syntax

We assume disjoint sets of channel names $\mathcal{N}$ (a, b, m, n) and variables $\mathcal{V}$ (x, y, z). We let r range over a set of references $\mathcal{R}$, and we assume a lattice of permissions $(\mathcal{P}, \sqsubseteq)$, letting ρ range over $\mathcal{P}$. The syntax of the language is given below:

$$\begin{array}{llll}
\textit{Constants} & c & ::= & num \mid str \mid bool \mid \textbf{unit} \mid \textbf{undefined}, \\
\textit{Values} & v & ::= & n \mid x \mid c \mid r_\ell \mid \lambda x.e \mid \{\overrightarrow{str_i : v_i}\} \\
\textit{Expressions} & e & ::= & v \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e\,e \mid op(\overrightarrow{e_i}) \mid \textbf{while}\ (e)\ \{\ e\ \} \\
 & & \mid & \textbf{if}\ (e)\ \{\ e\ \}\ \textbf{else}\ \{\ e\ \} \mid e; e \mid e[e] \mid e[e] = e \\
 & & \mid & \textbf{delete}\ e[e] \mid \textbf{ref}_\ell\ e \mid \textbf{deref}\ e \mid e := e \\
 & & \mid & \bar{e}\langle e \triangleright \rho \rangle \mid \textbf{exercise}(\rho)
\end{array}$$

$$\begin{array}{llll@{\qquad}llll}
\textit{Systems} & s & ::= & \mu; h; i & \textit{Memories} & \mu & ::= & \emptyset \mid \mu, r_\ell \overset{\rho}{\mapsto} v \\
\textit{Handlers} & h & ::= & \emptyset \mid h, a(x \triangleleft \rho : \rho').e & \textit{Instances} & i & ::= & \emptyset \mid i, a\{\!|e|\!\}_\rho
\end{array}$$

All the value forms are standard, we just note that references r_ℓ bear a label ℓ, taken from a set of labels $\mathcal{L}$. Labels identify the program point where references are created: this is needed for the static analysis and plays no role in the semantics. As usual, the lambda abstraction $\lambda x.e$ binds x in e.

As to expressions, the first three lines correspond to standard constructs inherited from λ_{JS}, including function applications, basic control-flow operators, and the usual operations on records (field selection, field update/creation, field deletion) and references (allocation, dereference and update). As anticipated, reference allocation comes with an annotation ℓ. We leave unspecified the precise set of primitive operations op. The expression **let** $x = e$ **in** e' binds x in e'.

The last line of the productions includes the new constructs added to λ_{JS}. The expression $\bar{a}\langle v \triangleright \rho \rangle$ sends the value v on channel a. In order for the sender to protect the message, the expression specifies that the value can be received by any *handler* with at least permission ρ that is listening on a. The expression $\mathbf{exercise}(\rho)$ exercises the privilege ρ. This construct uniformly abstracts any security-sensitive operation, such as the call to a privileged API, which requires the permission ρ to successfully complete the task.

We let h range over multisets of *handlers* of the form $a(x \triangleleft \rho : \rho').e$. The handler $a(x \triangleleft \rho : \rho').e$ listens for messages on the channel a. When a value v is sent over a, a new *instance* of the handler is spawned to run the expression e with permission ρ', with the bound variable x replaced by v. The handler protects its body against untrusted senders by specifying that only instances with permission ρ can be granted access. Intuitively, the body of a handler corresponds to the function passed as a parameter to the `addListener` method of `chrome.runtime.onMessage`. Different handlers can listen on the same channel: in this case, only one handler is non-deterministically dispatched. We often refer to a handler with the name of the channel where it is registered.

We let i range over multisets of running *instances* of the form $a\{|e|\}_\rho$. The instance $a\{|e|\}_\rho$ is a running expression e, which is granted permission ρ. The instance is annotated with the channel name a corresponding to the handler which spawned it.

We let μ range on *memories*, i.e., sets of bindings of the form $r_\ell \overset{\rho}{\mapsto} v$. A memory is a partial map from (labelled) references to values. The annotation ρ on the arrow records the permission of the instance that created the reference, and at the same time tracks the permissions required to have read-/write access on the reference. Given a memory μ, we let $dom(\mu) = \{r \mid r_\ell \overset{\rho}{\mapsto} v \in \mu\}$.

Finally, a *system* is defined as a triple $s = \mu; h; i$. Intuitively, a system evolves by letting running instances (*i*) communicate through the memory μ when they are granted exactly the same permissions, (*ii*) spawn new instances by sending messages to handlers in h, and (*iii*) perform internal computations.

A.3.2 Semantics

The small-step operational semantics of the calculus is defined in terms of a labelled reduction relation between systems $s \xrightarrow{\alpha} s'$. Labels play no role in the semantics of systems: they are just used to track useful information that

Table A.1 Small-step operational semantics of systems ($s \xrightarrow{\alpha} s'$)

$$\text{(R-SYNC)}\quad \frac{h = h', b(x \triangleleft \rho_s : \rho_b).e \qquad \rho_s \sqsubseteq \rho_a \qquad \rho_r \sqsubseteq \rho_b \qquad v \text{ serializable}}{\mu; h; a\{\!| E\langle \overline{b}\langle v \triangleright \rho_r \rangle\rangle |\!\}_{\rho_a} \xrightarrow{\langle a:\rho_a, b:\rho_b\rangle} \mu; h; a\{\!| E\langle \mathbf{unit}\rangle |\!\}_{\rho_a}, b\{\!| e[v/x] |\!\}_{\rho_b}}$$

$$\text{(R-SET)}\quad \frac{\mu; h; i \xrightarrow{\alpha} \mu'; h'; i'}{\mu; h; i, i'' \xrightarrow{\alpha} \mu'; h'; i', i''}$$

$$\text{(R-EXERCISE)}\quad \frac{\rho \sqsubseteq \rho_a}{\mu; h; a\{\!| E\langle \mathbf{exercise}(\rho)\rangle |\!\}_{\rho_a} \xrightarrow{a:\rho_a \gg \rho} \mu; h; a\{\!| E\langle \mathbf{unit}\rangle |\!\}_{\rho_a}}$$

$$\text{(R-INTERNAL)}\quad \frac{\mu; e \hookrightarrow_{\rho} \mu'; e'}{\mu; h; a\{\!| e |\!\}_{\rho} \xrightarrow{\cdot} \mu'; h; a\{\!| e' |\!\}_{\rho}}$$

$$\begin{array}{rcl} E & ::= & \bullet \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \mid E\,e \mid v\,E \mid op(\overrightarrow{v_i}, E, \overrightarrow{e_j}) \mid \mathbf{if}\ (E)\ \{e\}\ \mathbf{else}\ \{e\} \\ & \mid & E[e] \mid v[E] \mid E[e] = e \mid v[E] = e \mid v[v] = E \mid E; e \mid \overline{E}\langle e \triangleright \rho\rangle \mid \overline{v}\langle E \triangleright \rho\rangle \\ & \mid & \mathbf{delete}\ E[e] \mid \mathbf{delete}\ v[E] \mid \mathbf{ref}_{\ell}\ E \mid \mathbf{deref}\ E \mid E := e \mid v := E. \end{array}$$

is needed in the proofs. The syntax of labels α is defined as follows:

$$\alpha ::= \cdot \mid a{:}\rho_a \gg \rho \mid \langle a{:}\rho_a, b{:}\rho_b\rangle.$$

The label $a{:}\rho_a \gg \rho$ records the exercise of the privilege ρ by an instance a running with permissions ρ_a. The send label $\langle a{:}\rho_a, b{:}\rho_b\rangle$ records that an instance a with permissions ρ_a is sending a message to a handler b with permissions ρ_b. Finally, the empty label $\cdot$ tracks no information. We denote traces by $\overrightarrow{\alpha}$ and we write $\overset{\overrightarrow{\alpha}}{\Longrightarrow}$ for the reflexive-transitive closure of $\xrightarrow{\alpha}$. Table A.1 collects the reduction rules for systems and the definition of evaluation contexts. We write $E\langle e\rangle$ when the hole $\bullet$ in E is filled with the expression e.

Rule (R-SYNC) implements a security cross-check between the sender a and the receiver b: by specifying a permission ρ_r on the send expression, the instance a requires the handler b to have at least ρ_r, while by specifying a permission ρ_s in its definition, the handler b requires the instance a to have at least ρ_s. If the security check succeeds, a new instance of b is created and the sent value v is substituted to the bound variable x in the body of the handler. Communication is restricted to *serializable* values, according to the following definition.

Definition 25 (Serializable Value)**.** *A value v is* serializable *iff either (1) v is a name n or a constant c; or (2) $v = \{\overrightarrow{str_i : v_i}\}$ and each v_i is serializable.*

This restriction is consistent with the browser extension security architecture, which prevents the exchange of pointers between different components [20].

Table A.2 Small-step operational semantics of expressions ($\mu; e \hookrightarrow_\rho \mu'; e'$)

(JS-EXPR)
$$\frac{e_1 \hookrightarrow e_2}{\mu; e_1 \hookrightarrow_\rho \mu; e_2}$$

(JS-REF)
$$\frac{r \notin dom(\mu) \qquad \mu' = \mu, r_\ell \overset{\rho}{\mapsto} v}{\mu; \mathbf{ref}_\ell\ v \hookrightarrow_\rho \mu'; r_\ell}$$

(JS-DEREF)
$$\frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v}{\mu; \mathbf{deref}\ r_\ell \hookrightarrow_\rho \mu; v}$$

(JS-SETREF)
$$\frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v'}{\mu; r_\ell := v \hookrightarrow_\rho \mu', r_\ell \overset{\rho}{\mapsto} v; v}$$

(JS-CONTEXT)
$$\frac{\mu; e_1 \hookrightarrow_\rho \mu'; e_2}{\mu; E\langle e_1 \rangle \hookrightarrow_\rho \mu'; E\langle e_2 \rangle}$$

Rule (R-EXERCISE) reduces the expression **exercise**(ρ). Reduction takes place only when the expression runs in an instance a which is granted permission $\rho_a \sqsupseteq \rho$. Rule (R-SET) allows for reducing any of the parallel instances running in a system, while rule (R-INTERNAL) performs an internal reduction step based on the auxiliary transition relation $\mu; e \hookrightarrow_\rho \mu'; e'$, annotated with the permission ρ granted to the instance. The internal reduction relation is defined in Table A.2; it relies on a basic reduction $e \hookrightarrow e'$, which is directly inherited from λ_{JS} and lifted to the internal reduction by rule (JS-EXPR). The definition of the basic reduction is standard and given in the full version [17]..

A reference is allocated by means of rule (JS-REF). According to this rule, two references may have the same label (e.g., when reference allocation occurs inside a program loop) but each reference is guaranteed to have a distinct name. Since read/write operations on memory ultimately depend on the reference name, this ensures that labels on references do not play any role at runtime.

Finally, rules (JS-SETREF) and (JS-DEREF) deal with reference update and dereference. Observe that, according to these rules, both read and write access to memory requires *exactly* the permission ρ annotated on the reference. In other words, instances with different privileges cannot communicate through the memory. This corresponds to the heap separation policy implemented in modern browser extension architectures.

A.3.3 Privilege Leak

We now define the notion of *privilege leak*, which dictates an upper bound to the privileges which can be escalated by an opponent when interacting with the system. We start by defining when a system exercises a given permission.

Definition 26 (Exercise). *Given a system s, we say that s* exercises ρ *iff there exist s' and $\overrightarrow{\alpha}$ such that $s \overset{\overrightarrow{\alpha}}{\Longrightarrow} s'$ and $a{:}\rho_a \gg \rho \in \{\overrightarrow{\alpha}\}$.*

In our threat model, an opponent can mount an attack against the system by registering new handlers, which may intercept messages sent to trusted components, and/or by spawning new instances, which may tamper with the system by writing in shared memory cells and by using the message passing interface.

Formally, an opponent is defined as a pair (h, i), with an upper bound ρ for the permissions granted to h and i. For technical reasons, we assume that the set of variables $\mathcal{V}$ is partitioned into the sets $\mathcal{V}_t$ and $\mathcal{V}_u$ (trusted and untrusted variables). We stipulate that all the variables occurring in the system are drawn from $\mathcal{V}_t$, while all the variables occurring in the opponent code belong to $\mathcal{V}_u$.

Definition 27 (Opponent). *A ρ-*opponent *is a closed pair (h, i) where*

- *for any handler $a(x \lhd \rho : \rho').e \in h$, we have $\rho' \sqsubseteq \rho$;*
- *for any instance $a\{\!|e|\!\}_{\rho'} \in i$, we have $\rho' \sqsubseteq \rho$;*
- *for any $x \in vars(h) \cup vars(i)$, we have $x \in \mathcal{V}_u$.*

Definition 28 (Privilege Leak). *A (initial) system $s = \mu; h; \emptyset$* leaks ρ against ρ' *(with $\rho \not\sqsubseteq \rho'$) iff, for any ρ'-opponent (h_o, i_o), the system $s' = \mu; h, h_o; i_o$ exercises at most ρ.*

Our security property is given over *initial* systems, that is systems with no running instances, since we are interested in understanding the interplay between the exercised permissions and the communication interface exposed by the handlers in the system. Intuitively, a system s is "more secure" than another system s' if it leaks fewer privileges than s' against any possible ρ.

A.3.4 Encoding the Example

To illustrate, we encode in our formal language the example in Section A.1.1. Consider the system $s = \mu; h_c, h_o, h_b; \emptyset$, where the handlers h_c, h_o and h_b encode the two content scripts and the background page. The memory μ encodes the private memory of the background page, and it is used to store library functions. We grant the background page two different permissions:

MemB to access the references under its control and Cookies to access the cookie jar.

Let $\mathsf{B} = \mathsf{MemB} \sqcup \mathsf{Cookies}$, we let $\mu = lib_\ell \overset{\mathsf{B}}{\mapsto} obj$, where:

$$\begin{aligned} obj = \{&\text{“}set\text{”} : \lambda x.\mathbf{exercise}(\mathsf{Cookies}); \textit{set/update the cookie } x,\\ &\text{“}is\_valid\text{”} : \lambda x.\textit{check validity of policy } x,\\ &\text{“}store\text{”} : \lambda x.\lambda y.\mathbf{exercise}(\mathsf{MemB}); \textit{bind policy } y \textit{ to site } x,\\ &\text{“}log\text{”} : \lambda x.\textit{print message } x\} \end{aligned}$$

We omit the internal logic of the functions, we just observe that we put in place the exercise expressions corresponding to the usage of the required privileges. The definition of the handler h_b modelling the background page is given below, where C and O are the permissions granted to the two content scripts in order to let them contact B through the message passing interface.

```
$h_b \triangleq$ $b(x \triangleleft \mathsf{C} \sqcap \mathsf{O} : \mathsf{B}).$
    $\mathbf{let}\ mylib = \mathbf{deref}\ lib_\ell\ \mathbf{in}$
    $\mathbf{if}\ (x[\text{“}tag\text{”}] == \text{“}policy\text{”})\ \{$
        $\mathbf{if}\ (mylib[\text{“}is\_valid\text{”}]\ (x[\text{“}spec\text{”}]))\ \{$
            $(mylib[\text{“}store\text{”}]\ (x[\text{“}site\text{”}]))\ (x[\text{“}spec\text{”}])$
        $\}$
        $\mathbf{else}\ \{\ mylib[\text{“}log\text{”}]\ \text{“}invalid\ policy\text{”}\ \}$
    $\}$
    $\mathbf{else}\ \{$
        $\mathbf{if}\ (x[\text{“}tag\text{”}] == \text{“}upd\text{”})\ \{\ (mylib[\text{“}set\text{”}])\ (x[\text{“}ck\text{”}])\ \}$
        $\mathbf{else}\ \{\ mylib[\text{“}log\text{”}]\ \text{“}invalid\ message\text{”}\ \}$
    $\}$
```

The handler can be accessed by both C and O, as modelled by the guard $\mathsf{C} \sqcap \mathsf{O}$.

A simplified encoding of the content scripts, corresponding to the handlers h_c and h_o respectively, is given below. This simple encoding will be enough to explain the most important aspects of the flow analysis in Section A.4.3.

$$\begin{aligned} h_c &\triangleq c(y \triangleleft \mathsf{P} : \mathsf{C}).\mathbf{let}\ y' = (y[\text{“}site\text{”}] = \dots)\ \mathbf{in}\ \overline{b}\langle y' \triangleright \mathsf{B}\rangle\\ h_o &\triangleq o(z \triangleleft \top : \mathsf{O}).\mathbf{let}\ z' = \{\text{“}tag\text{”} : \text{“}upd\text{”}, \text{“}ck\text{”} : \dots\}\ \mathbf{in}\ \overline{b}\langle z' \triangleright \mathsf{B}\rangle \end{aligned}$$

The only notable point here is that h_o is protected with permission $\top$, since it is injected in the trusted options page of the extension, while h_c is protected with permission P, modelling access to `window.postMessage` method used to communicate with C from a web page. As a consequence, any P-opponent has the ability to activate h_c through the message passing interface.

Based on the encoding, we estimate the robustness against privilege escalation attacks. It turns out that the system s leaks B against P, since a P-opponent can force h_c into forwarding an arbitrary (up to the choice of the *"site"* field) message to h_b, hence all the privileges available to h_b may be escalated.

Assume then that h_c is replaced by a new handler h'_c, defined as follows:

$$\begin{aligned} h'_c \triangleq \ \ c(y \triangleleft \mathsf{P} : \mathsf{C}).\ \ & \mathbf{let}\ y_{new} = \{\text{"}tag\text{"} : \text{"}policy\text{"}, \text{"}site\text{"} : \ldots\}\ \mathbf{in} \\ & \mathbf{let}\ y' = (y_{new}[\text{"}spec\text{"}] = y[\text{"}spec\text{"}])\ \mathbf{in}\ \overline{b}\langle y' \triangleright \mathsf{B}\rangle \end{aligned}$$

The new system $s_{tag} = \mu; h'_c, h_o, h_b; \emptyset$ leaks MemB against P, since a P-opponent can only communicate with h_b through the proxy h'_c, which ensures that only messages tagged with *"policy"* are delivered to the background page and the integrity of the cookie jar is preserved. However, s_{tag} leaks B against C, since a C-opponent can send arbitrary messages to h_b and thus escalate all the available privileges.

A.3.5 Fixing the Example

The key observation here is that there is no good reason to let C and O share the same entry point to B, since they request distinct functionalities. We can then split the logic of h_b into two different handlers: h_{b_1} protected by permission C, and h_{b_2} protected by permission O.

```
$b_1(x \triangleleft \mathsf{C} : \mathsf{B})$.                                $b_2(x \triangleleft \mathsf{O} : \mathsf{B})$.
  let $mylib$ = deref $lib_\ell$ in                        let $mylib$ = deref $lib_\ell$ in
  if ($x$["tag"] == "policy") { ... }                   if ($x$["tag"] == "upd") { ... }
  else {$mylib$["log"] "invalid policy"}               else {$mylib$["log"] "invalid message"}
```

Clearly, the code of h_c and h_o must also be changed to communicate on the new channels b_1 and b_2 respectively: call these new handlers $\hat{h}_c$ and $\hat{h}_o$. Now the handler h_{b_1} is only accessible by $\hat{h}_c$, while the handler h_{b_2} can only be accessed by $\hat{h}_o$, hence, if O is not compromised, the integrity of the cookie jar is preserved.

Unfortunately, the current extension architecture does not support a fine-grained assignment of permissions to different portions of the background page [3], hence we are forced to violate the principle of least privilege and assign to both h_{b_1} and h_{b_2} the full set of permissions $\mathsf{B} = \mathsf{MemB} \sqcup \mathsf{Cookies}$ available to the original h_b, even though h_{b_1} and h_{b_2} only require a subset of these permissions. Still, the system $s_{chan} = \mu; \hat{h}_c, \hat{h}_o, h_{b_1}, h_{b_2}; \emptyset$ only leaks MemB against C.

Notice that this refactoring can be performed on existing Google Chrome extensions by using the `chrome.runtime.connect` API for the dynamic creation of communication ports towards the background page.

A.4 Security Analysis: Flow Logic

To precisely reason about privilege escalation, it is crucial to statically capture the interplay between the format of the exchanged messages and the exercised privileges: we then resort to the flow logic framework [57]. The main judgement of our flow analysis is $\mathcal{E} \Vdash s$ **despite** ρ, meaning that the environment $\mathcal{E}$ represents an acceptable analysis estimate for s, even when s interacts with a ρ-opponent. This implies that any ρ-opponent will at most escalate privileges up to an upper bound which can be immediately computed from $\mathcal{E}$ (see Theorem 5).

A.4.1 Analysis Specification

Abstract Values. We let $\hat{V}$ stand for the set of abstract values $\hat{v}$, defined as sets of abstract pre-values (we often omit brackets around singletons):

$$\begin{array}{lrcl} \textit{Abstract pre-values} & \hat{u} & ::= & n \mid \hat{c} \mid \ell \mid \lambda x^{\rho} \mid \langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{E},\rho} \\ \textit{Abstract values} & \hat{v} & ::= & \{\hat{u}_1, \ldots, \hat{u}_n\}. \end{array}$$

Channel names n are abstracted into themselves. The abstract pre-value $\hat{c}$ stands for the abstraction of the constant c. We dispense from listing all the abstract pre-values corresponding to the constants of our calculus, but we assume that they include at least **true**, **false**, **unit** and **undefined**.

A reference r_ℓ is abstracted into the label ℓ. A function $\lambda x.e$ is abstracted into the simpler representation λx^{ρ}, keeping track of the privileges ρ exercised by the expression e. The abstract pre-value $\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{E},\rho}$ is the abstract representation of the concrete record $\{\overrightarrow{str_i : v_i}\}$ in the environment $\mathcal{E}$, assuming that the record is created in a context with permission ρ. We do not fix any apriori abstract representation for records, e.g., both field-sensitive and field-insensitive representations are admissible.

We associate to each concrete operation *op* an abstract counterpart $\widehat{op}$ on abstract values. We also assume three abstract operations $\widehat{get}$, $\widehat{set}$ and $\widehat{del}$, mirroring the standard get, set and delete field operations on records. Finally, we assume that abstract values are ordered by a pre-order $\sqsubseteq$ containing set inclusion, with the intuition that smaller abstract values are more precise (we overload the symbol used to order permissions, to keep the notation lighter). All the abstract operations and the abstract value pre-order can be chosen arbitrarily, as long as they satisfy some relatively mild and well-established conditions needed in the proofs. For instance, we require abstract operations to be monotonic and to soundly over-approximate their concrete counterparts (see the full version [17] for details).

Abstract Environments. The judgements of the analysis are specified relative to an abstract environment $\mathcal{E} = \hat{\Upsilon}; \hat{\Phi}; \hat{\Gamma}; \hat{\mu}$, consisting of the following four components, where $\Lambda = \{\lambda x \mid x \in \mathcal{V}\}$ is used to store the abstract return value for lambdas:

Abstract variable environment	$\hat{\Gamma}$	:	$\mathcal{V} \cup \Lambda \to \hat{V}$
Abstract memory	$\hat{\mu}$	:	$\mathcal{L} \times \mathcal{P} \to \hat{V}$
Abstract stack	$\hat{\Upsilon}$	:	$\mathcal{N} \times \mathcal{P} \to \mathcal{P} \times \mathcal{P}$
Abstract network	$\hat{\Phi}$	:	$\mathcal{N} \times \mathcal{P} \to \hat{V}$.

Abstract variable environments are standard: they associate abstract values to variables and to functions, corresponding to the abstraction of their return value. Abstract memories are also standard: they associate abstract values to labels denoting references. Specifically, if $\hat{\mu}(\ell, \rho) = \hat{v}$, then $\hat{v}$ is a sound abstraction of any value stored in a reference labelled with ℓ and protected with permission ρ.

Abstract stacks are novel and are central to the privilege escalation analysis. This part of the environment is used to keep track of the permissions required to get access to each handler and the privileges which are exercised (also *transitively*, i.e., by communicating with other components) by the handlers themselves. Specifically, if $\hat{\Upsilon}(a, \rho_a) = (\rho_s, \rho_e)$, then the handler a with permission ρ_a can be accessed by any component with permission ρ_s and it will be able to exercise privileges up to ρ_e, possibly by calling other handlers in the system.

Table A.3 Flow analysis for values

$$\text{(PV-N{\scriptsize AME})}\ \frac{n \in \hat{v}}{\mathcal{E} \Vdash_\rho n \rightsquigarrow \hat{v}} \qquad \text{(PV-V{\scriptsize AR})}\ \frac{\mathcal{E}_{\hat{\Gamma}}(x) \sqsubseteq \hat{v}}{\mathcal{E} \Vdash_\rho x \rightsquigarrow \hat{v}} \qquad \text{(PV-C{\scriptsize ONS})}\ \frac{\{\hat{c}\} \sqsubseteq \hat{v}}{\mathcal{E} \Vdash_\rho c \rightsquigarrow \hat{v}} \qquad \text{(PV-R{\scriptsize EF})}\ \frac{\ell \in \hat{v}}{\mathcal{E} \Vdash_\rho r_\ell \rightsquigarrow \hat{v}}$$

$$\text{(PV-F{\scriptsize UN})}\ \frac{\lambda x^{\rho_e} \in \hat{v} \qquad \mathcal{E} \Vdash_\rho e : \hat{v}' \gg \rho' \qquad \hat{v}' \sqsubseteq \mathcal{E}_{\hat{\Gamma}}(\lambda x) \qquad \rho' \sqsubseteq \rho_e}{\mathcal{E} \Vdash_\rho \lambda x.e \rightsquigarrow \hat{v}} \qquad \text{(PV-R{\scriptsize EC})}\ \frac{\{\langle\!| \overrightarrow{str_i : v_i} |\!\rangle_{\mathcal{E},\rho}\} \sqsubseteq \hat{v}}{\mathcal{E} \Vdash_\rho \{\overrightarrow{str_i : v_i}\} \rightsquigarrow \hat{v}}$$

Finally, abstract networks are adapted from flow logic specifications for process calculi [59] and they are used to keep track of the messages sent to the handlers in the system. For instance, if we have $\hat{\Phi}(a, \rho_a) = \hat{v}$, then $\hat{v}$ is a sound abstraction of any message received by the handler a with permission ρ_a. Given an abstract environment $\mathcal{E}$, we denote by $\mathcal{E}_{\hat{\Gamma}}, \mathcal{E}_{\hat{\mu}}, \mathcal{E}_{\hat{\Upsilon}}, \mathcal{E}_{\hat{\Phi}}$ its four components.

Flow Analysis for Values and Expressions. The flow analysis for values and expressions consists of two mutually inductive judgements: $\mathcal{E} \Vdash_\rho v \rightsquigarrow \hat{v}$ and $\mathcal{E} \Vdash_\rho e : \hat{v} \gg \rho'$. The first judgement means that, assuming permission ρ, the concrete value v is mapped to the abstract value $\hat{v}$ in the abstract environment $\mathcal{E}$. The judgement $\mathcal{E} \Vdash_\rho e : \hat{v} \gg \rho'$ means that in the context of a handler (or an instance) with permission ρ, and under the abstract environment $\mathcal{E}$, the expression e may evaluate to a value abstracted by $\hat{v}$ and exercise at most ρ'.

The rules to derive $\mathcal{E} \Vdash_\rho v \rightsquigarrow \hat{v}$ are collected in Table A.3. Most of these rules are straightforward. The only rule worth commenting on here is (PV-FUN), which can be explained as follows: to abstract $\lambda x.e$ into $\hat{v}$, we first analyse the function body e to compute an approximation $\hat{v}'$ of the value it may evaluate to and an upper bound ρ' for the exercised privileges. Then, we check that $\lambda x^{\rho_e} \in \hat{v}$ for some $\rho_e \sqsupseteq \rho'$, i.e., we ensure that the exercised privileges are over-approximated in $\hat{v}$. Finally, we check that $\hat{v}' \sqsubseteq \mathcal{E}_{\hat{\Gamma}}(\lambda x)$, i.e., we guarantee that the abstract variable environment correctly over-approximates the return value of the function.

The analysis rules for expressions are collected in Table A.4. We comment on some representative rules below. Rule (PE-LET) can be explained as follows: to analyse **let** $x = e_1$ **in** e_2, we first analyse e_1 to compute an approximation $\hat{v}_1$ of the value it may evaluate to and an upper bound ρ_1 for the

Table A.4 Flow analysis for expressions

(PE-VAL)
$$\frac{\mathcal{E} \Vdash_{\rho_s} v \rightsquigarrow \hat{v}}{\mathcal{E} \Vdash_{\rho_s} v : \hat{v} \gg \rho}$$

(PE-LET)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \mathcal{E}_{\hat{\Gamma}}(x) \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \hat{v} \gg \rho}$$

(PE-APP)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \forall \lambda x^{\rho_e} \in \hat{v}_1 .\, \hat{v}_2 \sqsubseteq \mathcal{E}_{\hat{\Gamma}}(x) \wedge \mathcal{E}_{\hat{\Gamma}}(\lambda x) \sqsubseteq \hat{v} \wedge \rho_e \sqsubseteq \rho\end{array}}{\mathcal{E} \Vdash_{\rho_s} e_1\, e_2 : \hat{v} \gg \rho}$$

(PE-SEQ)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho\end{array}}{\mathcal{E} \Vdash_{\rho_s} e_1 ; e_2 : \hat{v} \gg \rho}$$

(PE-OP)
$$\frac{\forall i .\, \mathcal{E} \Vdash_{\rho_s} e_i : \hat{v}_i \gg \rho_i \sqsubseteq \rho \qquad \widehat{op}(\overrightarrow{\hat{v}_i}) \sqsubseteq \hat{v}}{\mathcal{E} \Vdash_{\rho_s} op(\overrightarrow{e_i}) : \hat{v} \gg \rho}$$

(PE-COND)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \sqsubseteq \rho \\ \textbf{true} \in \hat{v}_0 \Rightarrow \mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \hat{v} \gg \rho_1 \sqsubseteq \rho \\ \textbf{false} \in \hat{v}_0 \Rightarrow \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{if}\ (e_0)\ \{\ e_1\ \}\ \textbf{else}\ \{\ e_2\ \} : \hat{v} \gg \rho}$$

(PE-WHILE)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \textbf{true} \in \hat{v}_1 \Rightarrow \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \textbf{false} \in \hat{v}_1 \Rightarrow \textbf{undefined} \in \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{while}\ (e_1)\ \{\ e_2\ \} : \hat{v} \gg \rho}$$

(PE-GETFIELD)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \widehat{get}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} e_1[e_2] : \hat{v} \gg \rho}$$

(PE-SETFIELD)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \widehat{set}(\hat{v}_0, \hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} e_0[e_1] = e_2 : \hat{v} \gg \rho}$$

(PE-DELFIELD)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \\ \widehat{del}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{delete}\ e_1[e_2] : \hat{v} \gg \rho}$$

(PE-REF)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e : \hat{v}' \gg \rho' \sqsubseteq \rho \\ \hat{v}' \sqsubseteq \mathcal{E}_{\hat{\mu}}(\ell, \rho_s) \qquad \ell \in \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{ref}_{\ell}\ e : \hat{v} \gg \rho}$$

(PE-DEREF)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e : \hat{v}' \gg \rho' \sqsubseteq \rho \\ \forall \ell \in \hat{v}' .\, \mathcal{E}_{\hat{\mu}}(\ell, \rho_s) \sqsubseteq \hat{v}\end{array}}{\mathcal{E} \Vdash_{\rho_s} \textbf{deref}\ e : \hat{v} \gg \rho}$$

(PE-SETREF)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho \\ \forall \ell \in \hat{v}_1 .\, \hat{v}_2 \sqsubseteq \mathcal{E}_{\hat{\mu}}(\ell, \rho_s)\end{array}}{\mathcal{E} \Vdash_{\rho_s} e_1 := e_2 : \hat{v} \gg \rho}$$

(PE-SEND)
$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho' \\ \mathcal{E} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho' \\ \forall m \in \hat{v}_1 . \forall \rho_m \sqsupseteq \rho .\, \mathcal{E}_{\hat{\Upsilon}}(m, \rho_m) = (\rho_r, \rho_e) \wedge \rho_r \sqsubseteq \rho_s \ \Rightarrow \\ (\rho_e \sqsubseteq \rho' \wedge \hat{v}_2 \sqsubseteq \mathcal{E}_{\hat{\Phi}}(m, \rho_m) \wedge \textbf{unit} \in \hat{v})\end{array}}{\mathcal{E} \Vdash_{\rho_s} \overline{e_1}\langle e_2 \triangleright \rho \rangle : \hat{v} \gg \rho'}$$

(PE-EXERCISE)
$$\frac{\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \rho' \wedge \textbf{unit} \in \hat{v}}{\mathcal{E} \Vdash_{\rho_s} \textbf{exercise}(\rho) : \hat{v} \gg \rho'}$$

exercised privileges. We then ensure that the abstract variable environment $\mathcal{E}_{\hat{\Gamma}}(x)$ contains an over-approximation of $\hat{v}_1$ for the bound variable x, and we analyse e_2 to approximate its value as $\hat{v}_2$ and the exercised privileges as ρ_2. The analysis is acceptable if the abstract value $\hat{v}$ given to the let expression is an over-approximation of $\hat{v}_2$ and the estimated exercised privileges ρ are an upper bound for $\rho_1 \sqcup \rho_2$.

Rule (PE-APP) deals with function applications: it states that, to analyse $e_1\, e_2$, we first analyse the e_i's to compute the approximations $\hat{v}_i$ of the value they may evaluate to and the upper bounds ρ_i for the exercised privileges. We then focus on each λx^{ρ_e} contained in $\hat{v}_1$ and we check that: (1) the abstract variable environment binds x to an over-approximation of the abstraction of the actual argument of the function, (2) the abstract value $\hat{v}$ given to the application is an over-approximation of the abstract return value of the function $\mathcal{E}_{\hat{\Gamma}}(\lambda x)$, and (3) the exercised privileges $\rho_1 \sqcup \rho_2 \sqcup \rho_e$ are bounded above by the privileges ρ assigned to the application.

The rules in the central portion of the table should be relatively easy to understand. Notice that the rules for control flow operators, i.e., (PE-COND) and (PE-WHILE), allow for excluding from the static analysis some program branches which are never reached at runtime. The rules for references use the information ρ_s annotated on the turnstile, corresponding to the privileges granted to the handler/instance that is accessing the reference. These rules ensure that any value stored in a reference is correctly over-approximated by the abstract memory; and dually, that any value retrieved from a reference is abstracted with an over-approximation of the content of the abstract memory. This ensures that any value which is first stored in a reference and then retrieved from it is over-approximated correctly by the flow logic.

Rule (PE-SEND) first analyses e_1 and e_2 to compute the approximations of the recipient ($\hat{v}_1$) and the sent message ($\hat{v}_2$). Then, the last premise enforces two invariants: (1) the privileges ρ_e escalated by communicating with other handlers in the system are bounded above by the privileges ρ' assigned to the send expression, and (2) the abstraction of the sent message $\hat{v}_2$ is over-approximated by the information in the abstract network for each possible recipient. We also check that **unit** is included in the abstract value assigned to the expression, accordingly to the operational semantics of the send construct. Finally, rule (PE-EXERCISE) ensures that, whenever an instance with permission ρ_s exercises $\rho \sqsubseteq \rho_s$, then ρ is bounded above by the privileges ρ' assigned to the expression.

Flow Analysis for Systems. Finally, we extend the flow analysis to systems by defining the main judgement $\mathcal{E} \Vdash s$ **despite** ρ, which follows from similar judgements for memories, handlers and instances. The definition is given in Table A.5.

Table A.5 Flow analysis for systems

(PM-EMPTY)

$$\mathcal{E} \Vdash \emptyset \textbf{ despite } \rho$$

(PM-SINGLE)

$$\frac{\mathcal{E} \Vdash_{\rho_r} v \rightsquigarrow \hat{v} \qquad \hat{v} \sqsubseteq \mathcal{E}_{\hat{\mu}}(\ell, \rho_r)}{\mathcal{E} \Vdash r_\ell \overset{\rho_r}{\mapsto} v \textbf{ despite } \rho}$$

(PM-MANY)

$$\frac{\mathcal{E} \Vdash \mu_1 \textbf{ despite } \rho \qquad \mathcal{E} \Vdash \mu_2 \textbf{ despite } \rho}{\mathcal{E} \Vdash \mu_1, \mu_2 \textbf{ despite } \rho}$$

(PH-EMPTY)

$$\mathcal{E} \Vdash \emptyset \textbf{ despite } \rho$$

(PH-MANY)

$$\frac{\mathcal{E} \Vdash h \textbf{ despite } \rho \qquad \mathcal{E} \Vdash h' \textbf{ despite } \rho}{\mathcal{E} \Vdash h, h' \textbf{ despite } \rho}$$

(PH-SINGLE)

$$\frac{\begin{array}{c}\mathcal{E}_{\hat{\Upsilon}}(a, \rho_a) = (\rho'_s, \rho'_e) \qquad \rho_a \not\sqsubseteq \rho \Rightarrow \rho'_s = \rho_s \\ \mathcal{E}_{\hat{\Phi}}(a, \rho_a) \neq \emptyset \;\Rightarrow\; \mathcal{E}_{\hat{\Gamma}}(x) \sqsupseteq \mathcal{E}_{\hat{\Phi}}(a, \rho_a) \wedge \mathcal{E} \Vdash_{\rho_a} e : \hat{v} \gg \rho_e \wedge (\rho_a \not\sqsubseteq \rho \Rightarrow \rho'_e = \rho_e)\end{array}}{\mathcal{E} \Vdash a(x \lhd \rho_s : \rho_a).e \textbf{ despite } \rho}$$

(PI-EMPTY)

$$\mathcal{E} \Vdash \emptyset \textbf{ despite } \rho$$

(PI-SINGLE)

$$\frac{\begin{array}{c}\mathcal{E} \Vdash_{\rho_a} e : \hat{v} \gg \rho_e \\ \rho_a \not\sqsubseteq \rho \Rightarrow \exists \rho_s. \mathcal{E}_{\hat{\Upsilon}}(a, \rho_a) = (\rho_s, \rho_e)\end{array}}{\mathcal{E} \Vdash a\{\!|e|\!\}_{\rho_a} \textbf{ despite } \rho}$$

(PI-MANY)

$$\frac{\mathcal{E} \Vdash i \textbf{ despite } \rho \qquad \mathcal{E} \Vdash i' \textbf{ despite } \rho}{\mathcal{E} \Vdash i, i' \textbf{ despite } \rho}$$

(PS-SYS)

$$\frac{\begin{array}{cc}\mathcal{E} \Vdash \mu \textbf{ despite } \rho & \mathcal{E} \Vdash h \textbf{ despite } \rho \\ \mathcal{E} \Vdash i \textbf{ despite } \rho & \mathcal{E} \text{ is } \rho\text{-conservative}\end{array}}{\mathcal{E} \Vdash \mu; h; i \textbf{ despite } \rho}$$

In the rules for memories we just need to ensure (cf. rule (PM-SINGLE)) that, whenever a value v is stored in a reference r_ℓ protected with permission ρ_r, then v can be abstracted to some $\hat{v}$ over-approximated by the abstract memory entry $\mathcal{E}_{\hat{\mu}}(\ell, \rho_r)$. As for instances, rule (PI-SINGLE) computes an approximation of the privileges ρ_e exercised by the running expression. Then, if the instance is granted permission $\rho_a \not\sqsubseteq \rho$, i.e., if it is not compromised, we check that the abstract stack correctly approximates with ρ_e the privileges exercised by the instance body. This check is not enforced for instances that might be under the control of the opponent, according to the idea that any opponent must be accepted by a sufficiently weak abstract environment. This is needed to prove an *opponent acceptability* result (Lemma 16), which allows for a convenient soundness proof technique for the analysis [1, 12].

Handlers are accepted by rule (PH-SINGLE), which states that, to analyse $a(x \triangleleft \rho_s : \rho_a).e$ despite ρ-opponents, we first lookup the abstract stack $\hat{\Upsilon}$: let $\hat{\Upsilon}(a, \rho_a) = (\rho'_s, \rho'_e)$. If we are not analysing a (possibly) compromised handler, i.e., if $\rho_a \not\sqsubseteq \rho$, we ensure that the permission ρ'_s in the abstract stack matches the permission ρ_s guarding access to the handler. We then lookup the abstract network $\hat{\Phi}$: if $\hat{\Phi}(a, \rho_a) = \emptyset$, no instance of the system will ever communicate with the handler and we can skip the analysis of its body. Otherwise, we ensure that the abstract variable environment maps the bound variable x to an over-approximation of the incoming message, abstracted by $\hat{\Phi}(a, \rho_a)$, and we analyse the body of the handler, to detect the exercised privileges ρ_e. If we are not analysing the opponent, we further ensure that ρ_e matches the permissions ρ'_e annotated in the abstract stack, i.e., we guarantee that the abstract stack contains reliable information.

Finally, rule (PS-SYS) states that a system $s = \mu; h; i$ is acceptable for $\mathcal{E}$ only whenever μ, h and i are all acceptable for $\mathcal{E}$, and $\mathcal{E}$ is a *ρ-conservative* abstract environment. This notion corresponds to the informal idea of "sufficiently weak abstract environment" needed to prove the opponent acceptability result. In order to define ρ-conservativeness, we first define the notion of *static leak* for an abstract environment.

Definition 29 (Static Leak)**.** *We define the* static leak *of $\mathcal{E}$ against ρ as:*
$SLeak_\rho(\mathcal{E}) = \bigsqcup_{\rho_e \in L} \rho_e$, where $L = \{\rho_e \mid \exists a, \rho_a, \rho_s. \mathcal{E}_{\hat{\Upsilon}}(a, \rho_a) = (\rho_s, \rho_e) \wedge \rho_s \sqsubseteq \rho\}$.

Intuitively, $SLeak_\rho(\mathcal{E})$ is the upper bound of all the permissions ρ_e that can be (transitively) exercised by any handler that can be called by a ρ-opponent. We then define the set $\mathcal{V}_\rho(\mathcal{E})$ of the opponent-controlled variables as:

$$\mathcal{V}_\rho(\mathcal{E}) = \mathcal{V}_u \cup \{x \mid \exists \rho_e, \ell, \rho_r \sqsubseteq \rho. \lambda x^{\rho_e} \in \mathcal{E}_{\hat{\mu}}(\ell, \rho_r)\}.$$

The set contains all the variables $\mathcal{V}_u$ occurring in the opponent code, together with all the variables bound in lambda abstractions stored in references under the control of the opponent. All these variables can be instantiated at runtime with values chosen by the opponent. We use this set of variables also to define a sound abstraction of any value which can be generated by/flow to the opponent.

Definition 30 (Canonical Disclosed Abstract Value)**.** *Given an abstract environment $\mathcal{E}$ and a permission ρ, the* canonical disclosed abstract value *is defined as:* $\hat{v}_\rho(\mathcal{E}) = \{\hat{u} \mid vars(\hat{u}) \subseteq \mathcal{V}_\rho(\mathcal{E})\}$.

The canonical disclosed abstract value is a canonical representation of any abstract value under the control of a ρ-opponent in a system accepted by $\mathcal{E}$. It is the set of all the pre-values which contain only opponent-controlled variables.

Based on the notions above, we define ρ-conservativeness.

Definition 31 (ρ-Conservative Abstract Environment)**.** *An abstract environment $\mathcal{E}$ is* ρ-conservative *if and only if all the following conditions hold true:*

1. $\forall n \in \mathcal{N}, \forall \rho' \sqsubseteq \rho. \mathcal{E}_{\hat{\Upsilon}}(n, \rho') = (\bot, SLeak_\rho(\mathcal{E}))$;
2. $\forall n \in \mathcal{N}, \forall \rho' \sqsubseteq \rho. \mathcal{E}_{\hat{\Phi}}(n, \rho') = \hat{v}_\rho(\mathcal{E})$;
3. $\forall n \in \mathcal{N}, \forall \rho_n, \rho_s, \rho_e. \mathcal{E}_{\hat{\Upsilon}}(n, \rho_n) = (\rho_s, \rho_e) \wedge \rho_s \sqsubseteq \rho \Rightarrow \mathcal{E}_{\hat{\Phi}}(n, \rho_n) = \hat{v}_\rho(\mathcal{E})$;
4. $\forall \ell \in \mathcal{L}, \forall \rho' \sqsubseteq \rho. \mathcal{E}_{\hat{\mu}}(\ell, \rho') = \hat{v}_\rho(\mathcal{E})$;
5. $\forall x \in \mathcal{V}_\rho(\mathcal{E}). \mathcal{E}_{\hat{\Gamma}}(x) = \mathcal{E}_{\hat{\Gamma}}(\lambda x) = \hat{v}_\rho(\mathcal{E})$.

In words, an abstract environment is ρ-conservative whenever: (1) any handler that can be under the control of the opponent is in fact assumed to be accessible by the opponent and to escalate up to the static leak; (2) any handler that can be under the control of the opponent, or (3) that can be contacted by the opponent, is assumed to receive the canonical disclosed abstract value $\hat{v}_\rho(\mathcal{E})$; (4) any reference possibly under the control of the opponent is assumed to contain $\hat{v}_\rho(\mathcal{E})$; and (5) the argument of any function which can be called by the opponent is assumed to contain the canonical disclosed abstract value $\hat{v}_\rho(\mathcal{E})$ and similarly these functions are assumed to return $\hat{v}_\rho(\mathcal{E})$.

A.4.2 Formal Results

Our main formal result defines an upper bound for the privileges which can be escalated by the opponent in a system accepted by the flow analysis. Complete proofs are the full version [17]; here, we start proving the soundness of the flow logic specification by means of a subject reduction result, which ensures that the acceptability of the analysis is preserved upon reduction.

Lemma 15 (Subject Reduction)**.** *If $\mathcal{E} \Vdash s$ **despite** ρ and $s \xrightarrow{\alpha} s'$, then $\mathcal{E} \Vdash s'$ **despite** ρ.*

The next lemma states that any ρ-opponent is accepted by a ρ-conservative abstract environment. Intuitively, the combination of this result with subject reduction ensures that the acceptability of the analysis is preserved at runtime, even when the analysed system interacts with the opponent.

Lemma 16 (Opponent Acceptability). *If (h, i) is a ρ-opponent and $\mathcal{E}$ is ρ-conservative, then $\mathcal{E} \Vdash h$* **despite** *ρ and $\mathcal{E} \Vdash i$* **despite** *ρ.*

Moreover, proving the safety theorem requires to explicitly track the call chains carried out by the system reduction, to collect the privileges transitively exercised by system components. The next lemma then relies on the following definition of call chain to prove that the abstract stack contains a static approximation of the privileges which are exercised by each system component either directly or by communicating with other components.

Definition 32 (Call Chain). *A* call chain *$(\overrightarrow{\alpha}, a{:}\rho_a \gg \rho')$ is a trace of length $(n+1)$ such that:*

1. *the trace $\overrightarrow{\alpha} = \langle a_1{:}\rho_{a_1}, b_1{:}\rho_{b_1}\rangle, \dots, \langle a_n{:}\rho_{a_n}, b_n{:}\rho_{b_n}\rangle$ is a sequence of send labels where the sender occurring in each label is the receiver occurring in the previous label, i.e., $\forall i \in [1, n-1]. \, a_{i+1} = b_i \wedge \rho_{a_{i+1}} = \rho_{b_i}$, and*
2. *the component exercising the privilege ρ' at the end of the call chain corresponds to the last receiver, i.e., $b_n = a \wedge \rho_{b_n} = \rho_a$.*

A trace $\overrightarrow{\beta}$ includes *a call chain $\overrightarrow{\alpha}$ iff $\overrightarrow{\alpha}$ is a sub-trace of $\overrightarrow{\beta}$.*

According to the intuition given above, proving the soundness of the abstract stack amounts to showing that, given a call chain leading to the exercise of some privilege ρ' not available to the opponent, the abstract stack $\mathcal{E}_{\hat{\Upsilon}}$ approximates the privileges exercised by any component involved in the chain with a permission greater than or equal to ρ'. The proof uses the subject reduction result.

Lemma 17 (Soundness of the Abstract Stack). *If $\mathcal{E} \Vdash s$* **despite** *ρ and $s \overset{\overrightarrow{\beta}}{\Longrightarrow} s'$ for a trace $\overrightarrow{\beta}$ including the call chain $(\overrightarrow{\alpha}, a{:}\rho_a \gg \rho')$ for some $\rho' \not\sqsubseteq \rho$, then for each label $\alpha_j = \langle a_j{:}\rho_{a_j}, b_j{:}\rho_{b_j}\rangle \in \{\overrightarrow{\alpha}\}$ we have $\mathcal{E}_{\hat{\Upsilon}}(b_j, \rho_{b_j}) = (\rho_{s_{b_j}}, \rho_{e_{b_j}})$ with $\rho' \sqsubseteq \rho_{e_{b_j}}$ and $\mathcal{E}_{\hat{\Upsilon}}(a_j, \rho_{a_j}) = (\rho_{s_{a_j}}, \rho_{e_{a_j}})$ with $\rho' \sqsubseteq \rho_{e_{a_j}}$.*

Theorem 5 (Flow Safety). *Let $s = \mu; h; \emptyset$. If $\mathcal{E} \Vdash s$* **despite** *ρ, then s leaks $SLeak_\rho(\mathcal{E})$ against ρ.*

Proof. By contradiction. Let $\hat{s}$ be the system obtained by composing s with a ρ-opponent and assume that $\hat{s}$ eventually reaches a state s' such that s' exercises privileges ρ_{bad}, with $\rho_{bad} \not\sqsubseteq \rho$ and $\rho_{bad} \not\sqsubseteq SLeak_{\rho}(\mathcal{E})$.

By inverting rule (PS-SYS) on the hypothesis $\mathcal{E} \Vdash s$ **despite** ρ, we have that $\mathcal{E}$ is ρ-conservative. Using Lemma 16 (Opponent Acceptability), we show that $\mathcal{E} \Vdash \hat{s}$ **despite** ρ. Given that $\rho_{bad} \not\sqsubseteq \rho$, the privileges ρ_{bad} cannot be directly exercised by the opponent, hence there must exist a call chain leading to ρ_{bad} from $\hat{s}$. Let a_i range over the components in the call chain and ρ_i range over their corresponding permissions. Consider now the first sender a_1 in the call chain: given that the original system s does not have running instances, it turns out that a_1 must be the opponent, hence $\rho_1 \sqsubseteq \rho$. Since $\mathcal{E}$ is ρ-conservative and $\rho_1 \sqsubseteq \rho$, we have $\mathcal{E}_{\hat{\Upsilon}}(a_1, \rho_1) = (\bot, SLeak_{\rho}(\mathcal{E}))$. By Lemma 17 (Soundness of the Abstract Stack), for each component a_i with permissions ρ_i occurring in the call chain we must have $\mathcal{E}_{\hat{\Upsilon}}(a_i, \rho_i) = (\rho_{s_i}, \rho_{e_i})$ for some ρ_{s_i} and some $\rho_{e_i} \sqsupseteq \rho_{bad}$. But then we get $\rho_{bad} \sqsubseteq SLeak_{\rho}(\mathcal{E})$, which is contradictory. $\square$

A.4.3 Analysing the Example

We now show the analysis at work on our running example in its three variants, namely the systems s, s_{tag} and s_{chan} introduced in Section A.3. We assume that the abstract domain for strings includes all the string literals syntactically occurring in the program code, plus the distinguished symbol * to represent all the other strings (or any string which we cannot statically reconstruct). We let $\widehat{str}$ range over elements of this abstract domain and we assume that $\widehat{str} \sqsubseteq *$ for any $\widehat{str}$. As to records, we choose the field-sensitive representation $\langle\!| \overrightarrow{\widehat{str}_i : \hat{v}_i} |\!\rangle$ where both the field names and contents are inductively abstracted. In the following we mostly focus on the intuitions behind the analysis: additional details, including the formal definitions of the expected abstract record operations and the abstract value pre-order, are given in the full version [17].

The Original System. We start by studying the robustness of the original system s against a P-opponent, i.e., an opponent with the only ability to dispatch the content script C attached to untrusted web pages. We have that $\mathcal{E} \Vdash s$ **despite** P, where $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ satisfies the following assumptions:

$$\hat{\Phi}(c, \mathsf{C}) = \hat{v}_{\mathsf{P}}(\mathcal{E}) \qquad \hat{\Phi}(o, \mathsf{O}) = \emptyset \qquad \hat{\Phi}(b, \mathsf{B}) = \{\langle\!| \text{“}site\text{”} : \hat{v}_{\mathsf{P}}(\mathcal{E}), * : \hat{v}_{\mathsf{P}}(\mathcal{E}) |\!\rangle\}$$
$$\hat{\Upsilon}(c, \mathsf{C}) = (\mathsf{P}, \mathsf{B}) \qquad \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) \qquad \hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B})$$

Since C can be accessed by the opponent, the value of $\hat{\Phi}(c, \mathsf{C})$ must be equal to $\hat{v}_{\mathsf{P}}(\mathcal{E})$ to ensure the P-conservativeness of $\mathcal{E}$. Conversely, O can never be accessed by the opponent or by any other component in the system, hence $\hat{\Phi}(o, \mathsf{O}) = \emptyset$. By rule (PH-SINGLE), this implies that there is no need to analyse the body of O, which allows for ignoring the format of the messages sent by O: this explains why the value of $\hat{\Phi}(b, \mathsf{B})$ includes just one element, corresponding to the message sent by C. Indeed, observe that $\widehat{set}(\hat{v}_{\mathsf{P}}(\mathcal{E}), \text{“}site\text{”}, str) \sqsubseteq \{\langle\!|\text{“}site\text{”} : \hat{v}_{\mathsf{P}}(\mathcal{E}), * : \hat{v}_{\mathsf{P}}(\mathcal{E})|\!\rangle\}$ for any str to accept the send expression in the body of C.

Now observe that $\{\text{“}policy\text{”}, \text{“}upd\text{”}\} \sqsubseteq \widehat{get}(\langle\!|\text{“}site\text{”} : \hat{v}_{\mathsf{P}}(\mathcal{E}), * : \hat{v}_{\mathsf{P}}(\mathcal{E})|\!\rangle, \text{“}tag\text{”})$, hence both branches of the conditional in the body of B are reachable and the conditional expression may exercise B; we then let $\hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B})$ by rule (PH-SINGLE). Given that C communicates with B, the privileges exercised by C must be greater or equal than B by rule (PE-SEND), and propagated into $\hat{\Upsilon}(c, \mathsf{C})$ by rule (PH-SINGLE). Since $SLeak_{\mathsf{P}}(\mathcal{E}) = \mathsf{B}$, we know that the system s leaks B against P by Theorem 5.

The System with Tags. Let us focus now on the system s_{tag} and a P-opponent. We have that $\mathcal{E} \Vdash s_{tag}$ **despite** P, where $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ is such that:

$$\begin{array}{lll} \hat{\Phi}(c, \mathsf{C}) = \hat{v}_{\mathsf{P}}(\mathcal{E}) & \hat{\Phi}(o, \mathsf{O}) = \emptyset & \\ \multicolumn{3}{l}{\hat{\Phi}(b, \mathsf{B}) = \{\langle\!|\text{“}tag\text{”} : \text{“}policy\text{”}, \text{“}site\text{”} : *, \text{“}spec\text{”} : \hat{v}_{\mathsf{P}}(\mathcal{E})|\!\rangle\}} \\ \hat{\Upsilon}(c, \mathsf{C}) = (\mathsf{P}, \mathsf{MemB}) & \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) & \hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{MemB}) \end{array}$$

Based on this information, rule (PE-COND) allows for analysing only the program branch of B corresponding to the processing of a message with tag *“policy”*, which only exercises the privilege MemB: this motivates the precise choice of $\hat{\Upsilon}(b, \mathsf{B})$. Since $SLeak_{\mathsf{P}}(\mathcal{E}) = \mathsf{MemB}$, the system leaks MemB against P.

Assume now an opponent with permission C, then $\mathcal{E}' \Vdash s_{tag}$ **despite** C, where $\mathcal{E}' = \hat{\Gamma}'; \hat{\mu}'; \hat{\Upsilon}'; \hat{\Phi}'$ is such that:

$$\begin{array}{lll} \hat{\Phi}'(c, \mathsf{C}) = \hat{v}_{\mathsf{C}}(\mathcal{E}') & \hat{\Phi}'(o, \mathsf{O}) = \emptyset & \hat{\Phi}'(b, \mathsf{B}) = \hat{v}_{\mathsf{C}}(\mathcal{E}') \\ \hat{\Upsilon}'(c, \mathsf{C}) = (\bot, \mathsf{B}) & \hat{\Upsilon}'(o, \mathsf{O}) = (\top, \bot) & \hat{\Upsilon}'(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B}) \end{array}$$

With respect to the previous scenario, the abstract network entry for B contains $\hat{v}_{\mathsf{C}}(\mathcal{E}')$, abstracting all the values which may be generated by a C-opponent: this is needed for C-conservativeness. The consequence is that all the program branches of B are reachable, hence B may exercise its full set of privileges B. Since $SLeak_{\mathsf{C}}(\mathcal{E}') = \mathsf{B}$, the system leaks B against C by Theorem 5.

The System with Channels. We are able to prove $\mathcal{E} \Vdash s_{chan}$ **despite** C for an abstract environment $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ such that:

$$\hat{\Phi}(c, \mathsf{C}) = \hat{v}_{\mathsf{C}}(\mathcal{E}) \quad \hat{\Phi}(o, \mathsf{O}) = \emptyset \quad \hat{\Phi}(b_1, \mathsf{B}) = \hat{v}_{\mathsf{C}}(\mathcal{E}) \quad \hat{\Phi}(b_2, \mathsf{B}) = \emptyset$$
$$\hat{\Upsilon}(c, \mathsf{C}) = (\bot, \mathsf{MemB}) \quad \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) \quad \hat{\Upsilon}(b_1, \mathsf{B}) = (\mathsf{C}, \mathsf{MemB}) \quad \hat{\Upsilon}(b_2, \mathsf{B}) = (\mathsf{O}, \bot)$$

For the new abstract environment $\mathcal{E}$ we have $SLeak_{\mathsf{C}}(\mathcal{E}) = \mathsf{MemB}$, which ensures that the new system only leaks MemB against C. Since the privilege $\mathsf{Cookies}$ cannot be escalated by a compromised C anymore, there is no way to corrupt the cookie jar without compromising the background page B itself (or the options page O). Interestingly, this is a formal characterization of the dangers connected to the development of *bundled* browser extensions in a realistic setting [3].

A.5 Implementation: CHEN

CHEN is a prototype Google Chrome extension analyser written in F#. Given a Chrome extension, CHEN translates it into a corresponding system in our formalism and computes an acceptable flow analysis estimate by constraint solving. CHEN can be used by programmers to evaluate the robustness of their extensions against privilege escalation attacks and to support their security refactoring.

A.5.1 Flow Logic Implementation

Implementing the flow logic specification amounts to defining an algorithm that, given a system s and a permission ρ characterizing the power of the opponent, computes an abstract environment $\mathcal{E}$ such that $\mathcal{E} \Vdash s$ **despite** ρ. Following a standard approach [58], we first define a verbose variant of the flow logic, which associates an analysis estimate to each sub-expression of s, and then we devise a constraint-based formulation of the analysis. Any solution of the constraints is an abstract environment $\mathcal{E}$ which accepts s.

We initially implemented in CHEN a simple worklist algorithm for constraint solving. However, consistently with what has been reported by Jensen *et al.* in the context of JavaScript analysis [44], we observed that this solution does not scale, taking hours to perform the analysis even on small examples.

Therefore, in our implementation we use a variant of the worklist algorithm where most of the constraint generation is performed *on demand* during the solving process. Even though this approach does not allow us to reuse existing solvers, it leads to a dramatic improvement in the performances of the analysis.

The current prototype implements a context-insensitive analysis, which is enough to capture the privileges escalated by the content scripts, provided that some specific library functions introduced by the desugaring process from JavaScript to λ_{JS} (see below) are inlined. The choice of the abstract pre-values for constants is standard: in the current implementation, we represent numbers with their sign and we approximate strings with finite prefixes [23]. The representation of records is field-sensitive, but we collapse into a single label * all the entries bound to approximate labels (string prefixes). As to the ordering, we consider a standard pre-order $\sqsubseteq_p$ on abstract pre-values, and we lift it to abstract values using a lower powerset construction, i.e., we let $\hat{v} \sqsubseteq \hat{v}'$ if and only if $\forall \hat{u} \in \hat{v}. \exists \hat{u}' \in \hat{v}'. \hat{u} \sqsubseteq_p \hat{u}'$.

A.5.2 Using CHEN to assess Google Chrome Extensions

Given an extension, CHEN takes as an input a sequence of *component* names, along with the JavaScript files corresponding to their implementation. Components represent isolation domains, in that different components must be able to communicate only using the message passing interface. Different content scripts which may injected in the same web page should be put inside the same component, since Google Chrome does not separate their heaps. The background page should be put in a separate component, since it runs in an isolated process[2].

From JavaScript to the Model. Let c be a component name and $f_1, \ldots, f_n$ the corresponding JavaScript files: our tool concatenates $f_1, \ldots, f_n$ into a single file f, which is desugared into a closed λ_{JS} expression using an existing tool [36]. The adequacy of the translation from JavaScript to λ_{JS} has been assessed by extensive automatic testing, hence safety guarantees for JavaScript programs can be provided just by analysing their λ_{JS} translation; see [36] for further details.

[2]An appropriate mapping of JavaScript files to components can be derived from the manifest file of the extension, but the current prototype does not support this feature.

The obtained λ_{JS} expression is then transformed into a set of handlers: more precisely, for any function $\lambda x.e'$ passed as argument to the `addListener` method of `chrome.runtime.onMessage`, we introduce a new handler on a channel with the same name of the component, whose body is obtained by closing e' with the introduction of all the bindings defined before the registration of the listener. For each component we introduce a unique permission for memory access, granted to each handler in the component; handlers corresponding to the background page are also given the permissions specified in the manifest of the extension. Any invocation of `chrome.runtime.sendMessage` in the definition of a content script is translated to a send expression over a channel with the name of the component corresponding to the background page.

Notice that CHEN exploits an existing tool to translate JavaScript to λ_{JS}, but our target language has two new constructs: message sending and privilege exercise. In JavaScript, both operations correspond to function calls to the Chrome extension API, hence, to introduce the syntactic forms corresponding to them in the translation to our formalism, we extend the JavaScript code to redefine the functions of interest in the Chrome API with *stubs*. For instance, `chrome.cookies.set` is redefined to a function including the special tag `"#Cookies#"`, which is preserved when desugaring JavaScript to λ_{JS}: we then post-process the λ_{JS} expression to replace this tag with the function $\textbf{exercise}(\mathsf{Cookies})$.

Running the Analysis. The tool supports two analyses. With the option `-compromise` CHEN is instructed to analyse the privileges which may be escalated by an opponent assuming the full compromise of an arbitrary content script, i.e., it estimates the safety of the system despite the permission that protects the background page. If the background page requests some permission ρ intended for internal use, but ρ is available to some content script according to the results of the analysis, then the developer is recommended to review the communication interface.

Alternatively, the option `-target` n allows to get an approximation of the privileges available to the content scripts in the component n in absence of compromise. We model absence of compromise by considering a $\perp$-opponent as the threat model, since this opponent cannot directly communicate with the background page: if the option `-target` n is specified, CHEN transforms the system by protecting with permission $\perp$ all the handlers included in n, and computes a permission ρ such that the system is ρ-safe despite $\perp$. This allows to estimate which privileges are enabled by messages sent from n, so as to identify potential room for a security refactoring, as we discuss below.

Both the analyses additionally support the option `-flag` p, which allows to define a dummy permission p assigned to the background page. The programmer may then annotate specific program points with the tag `"#p#`, corresponding to the exercise of this dummy permission; by checking the presence of the flag among the escalated privileges, CHEN can be used to implement an opponent-aware reachability analysis on the extension code.

Supporting a Security Refactoring. To exemplify, we analyse with CHEN our motivating example. By first specifying the option `-target` O, the tool detects that the options page O is only accessing the privilege Cookies as part of its standard functionalities, even though the background page B is given the permissions MemB $\sqcup$ Cookies. To support least privilege, the developer is thus recommended to introduce a distinct communication port for B. Notably, the permission gap arises from the presence in the code of B of program branches which are never triggered by messages sent by O in absence of compromise: in principle, CHEN could then automatically introduce the new port, replicate the code from the handler of the background page, and improve its security against compromise by eliminating the dead branches, even though the current prototype does not implement this feature.

Then, by using the option `-target` C, the tool outputs that the privilege MemB $\sqcup$ Cookies can be escalated by the content script C. Hence, no automated refactoring is possible, but the output of the analysis is still helpful for a careful developer, who realizes that C should not be able to access the Cookies privilege. Based on the output of the analysis, the developer may opt for a manual reviewing and refactoring of the extension.

Current Limitations. Being a proof-of-concept implementation, the current version of CHEN lacks a full coverage of the Chrome extension APIs. Moreover, CHEN cannot analyse extensions which use ports to communicate: in our model, ports are just channels and do not pose any significant problem to the analysis. Unfortunately, the current Chrome API makes it difficult to support the analysis of extensions using ports, since the underlying programming patterns make massive usage of callbacks. Based on our experience and a preliminary investigation, however, ports are not widely used in practice, hence many extensions can still be analysed by CHEN.

A.5.3 Case Study: ShareMeNot

ShareMeNot [66] is a popular privacy-enhancing extension developed at the University of Washington. The extension looks for social sharing buttons in the web pages and replaces them with dummy buttons: only when the user clicks one of these buttons, its original version is loaded and the cookies registered by the corresponding social networks are sent. This means that the social network can track the user only when the user is willing to share something.

ShareMeNot consists of four components: a content script, a background page, an option page and a popup, for a total of approximately 1,500 lines of JavaScript code. The background page offers a unique entry point to all the other extension components and handles seven different message types. Interestingly, one of these messages allows to unblock all the trackers in an arbitrary tab, by invoking the `unblockAllTrackersOnTab` function: this message should only be sent by the popup page. We then put a flag in the body of the function and we performed the analysis of ShareMeNot with the `-compromise` option, observing that the flag is reachable: hence, a compromised content script could entirely deactivate the extension. The analysis took around 150 seconds on a standard commercial machine.

We then ran the analysis with the `-target` C option, where C is the name of the component including only the content script, and we observed that the flag was not reachable. This means that C does not need to access the function `unblockAllTrackersOnTab` as part of its standard functionalities, hence the code should be refactored to comply with the principle of the least privilege and prevent a potential security risk. The analysis took around 210 seconds on the same machine.

A.6 Conclusions

We presented a core calculus to reason about browser extensions security and we proposed a flow analysis aimed at detecting which privileges may be leaked to an opponent which compromises some (arbitrarily chosen) untrusted extension components. The analysis has been proved sound and it has been implemented in CHEN, a prototype static analyser for Google Chrome extensions. We discussed how CHEN can assist developers in writing more robust extensions.

As future work, we plan to further engineer CHEN, to make it support more sophisticated communication patterns used in Google Chrome extensions. We ultimately plan to evolve CHEN into a compiler, which automatically refactors the extension code to make it more secure, by unbundling functionalities based on their exercised permissions. Based on a preliminary investigation, this will require a non-trivial programming effort.

Bibliography

[1] Abadi, M.: Secrecy by typing in security protocols. J. ACM 46, 749–786 (1999)

[2] Ahn, G., Sandhu, R.S., Kang, M.H., Park, J.S.: Injecting RBAC to secure a web-based workflow system. In: ACM Workshop on Role-Based Access Control. pp. 1–10 (2000)

[3] Akhawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: USENIX Security Symposium. pp. 429–444 (2012)

[4] Alberti, F., Armando, A., Ranise, S.: Efficient symbolic automated analysis of administrative attribute-based rbac-policies. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011. pp. 165–175 (2011)

[5] Azghandi, N.G.: Petri nets, probability and event structures. Ph.D. thesis, University of Edinburgh (2014)

[6] Bandhakavi, S., Tiku, N., Pittman, W., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with VEX. Communications of the ACM 54(9), 91–99 (2011)

[7] Barbon, G., Cortesi, A., Ferrara, P., Steffinlongo, E.: DAPA: degradation-aware privacy analysis of android apps. In: Security and Trust Management - 12th International Workshop, STM 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings. pp. 32–46 (2016)

[8] Barth, A., Porter Felt, A., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: NDSS (2010)

[9] Basin, D.A., Burri, S.J., Karjoth, G.: Obstruction-free authorization enforcement: Aligning security and business objectives. Journal of Computer Security 22(5), 661–698 (2014)

[10] Bertino, E., Ferrari, E., Atluri, V.: The specification and enforcement of authorization constraints in workflow management systems. ACM Trans. Inf. Syst. Secur. 2(1), 65–104 (1999)

[11] Bertolissi, C., dos Santos, D.R., Ranise, S.: Automated synthesis of run-time monitors to enforce authorization policies in business processes. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015. pp. 297–308 (2015)

[12] Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)

[13] Botha, R.A., Eloff, J.H.P.: Separation of duties for access control enforcement in workflow environments. IBM Systems Journal 40(3), 666–682 (2001)

[14] Bugliesi, M., Calzavara, S., Focardi, R., Khan, W.: Automatic and robust client-side protection for cookie-based sessions. In: ESSoS. pp. 161–178 (2014)

[15] Bugliesi, M., Calzavara, S., Focardi, R., Khan, W., Tempesta, M.: Provably sound browser-based enforcement of web session integrity. In: CSF. pp. 366–380 (2014)

[16] Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: Towards security type-checking of Android applications. In: FMOODS/FORTE. pp. 289–304 (2013)

[17] Calzavara, S., Bugliesi, M., Crafa, S., Steffinlongo, E.: Fine-grained detection of privilege escalation attacks on browser extensions (full version). Available at `http://www.dais.unive.it/~calzavara/papers/esop15-full.pdf`

[18] Calzavara, S., Rabitti, A., Bugliesi, M.: Compositional typed analysis of ARBAC policies. In: IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015. pp. 33–45 (2015)

[19] Calzavara, S., Rabitti, A., Steffinlongo, E., Bugliesi, M.: Static detection of collusion attacks in ARBAC-based workflow systems. Tech. rep. (2016), available at `https://sites.google.com/site/warbacanalyser/`

[20] Carlini, N., Porter Felt, A., Wagner, D.: An evaluation of the Google Chrome extension security architecture. In: USENIX Security Symposium. pp. 97–111 (2012)

[21] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings. pp. 359–364 (2002), tool available at http://nusmv.fbk.eu/. Last accessed: 11-12-2017

[22] Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. pp. 93–107 (2013)

[23] Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: ICFEM. pp. 505–521 (2011)

[24] Crampton, J., Khambhammettu, H.: Delegation and satisfiability in workflow systems. In: SACMAT 2008, 13th ACM Symposium on Access Control Models and Technologies, Estes Park, CO, USA, June 11-13, 2008, Proceedings. pp. 31–40 (2008)

[25] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on Android. In: ISC. pp. 346–360 (2010)

[26] De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS. pp. 337–340 (2008)

[27] Dhawan, M., Ganapathy, V.: Analyzing information flow in JavaScript-based browser extensions. In: ACSAC. pp. 382–391 (2009)

[28] Dinh, K.K.Q., Tran, T.D., Truong, A.: Security analysis of administrative role-based access control policies with contextual information. In: Proceedings of the 4th International Conference Future Data and Security

Engineering, FDSE 2017, Ho Chi Minh City, Vietnam, November 29 - December 1, 2017. pp. 243–261 (2017)

[29] Dutertre, B.: Yices 2.2. In: Computer-Aided Verification (CAV'2014). Lecture Notes in Computer Science, vol. 8559, pp. 737–744 (July 2014)

[30] Ferraiolo, D.F., Sandhu, R.S., Gavrila, S.I., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. 4(3), 224–274 (2001)

[31] Ferrara, A.L., Madhusudan, P., Nguyen, T.L., Parlato, G.: Vac - Verifier of administrative role-based access control policies. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 184–191 (2014)

[32] Ferrara, A.L., Madhusudan, P., Parlato, G.: Security analysis of role-based access control through program verification. In: 25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012. pp. 113–125 (2012)

[33] Ferrara, A.L., Madhusudan, P., Parlato, G.: Policy analysis for self-administrated role-based access control. In: Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. pp. 432–447 (2013)

[34] Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: ESORICS. pp. 1–18 (2012)

[35] Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: 32nd IEEE Symposium on Security and Privacy. pp. 115–130 (2011)

[36] Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of JavaScript. In: ECOOP. pp. 126–150 (2010)

[37] Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: ESOP. pp. 256–275 (2011)

[38] Gupta, M., Sandhu, R.: The \mathrm gura_g GURA G administrative model for user and group attribute assignment. In: Network and System Security - 10th International Conference, NSS 2016, Taipei, Taiwan, September 28-30, 2016, Proceedings. pp. 318–332 (2016)

[39] Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings. pp. 585–602 (2014)

[40] Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M., Chapin, S.: Automatic error finding in access-control policies. In: Proceedings of the 18th ACM Conference on Computer and Communications Security. pp. 163–174. CCS '11 (2011)

[41] Jayaraman, K., Tripunitara, M.V., Ganesh, V., Rinard, M.C., Chapin, S.J.: Mohawk: Abstraction-refinement and bound-estimation for verifying access control policies. ACM Trans. Inf. Syst. Secur. 15(4), 18 (2013)

[42] Jeannet, B.: Interproc analyzer for recursive programs with numerical variables pp. 01–11 (2010), INRIA, software and documentation are available at `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html`. Last accessed: 11-12-2017

[43] Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: SAS. pp. 238–255 (2009)

[44] Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: SAS. pp. 320–339 (2010)

[45] Jha, S., Li, N., Tripunitara, M.V., Wang, Q., Winsborough, W.H.: Towards formal verification of role-based access control policies. IEEE Trans. Dependable Sec. Comput. 5(4), 242–255 (2008)

[46] Jin, X., Krishnan, R., Sandhu, R.: A role-based administration model for attributes. In: Proceedings of the First International Workshop on Secure and Resilient Architectures and Systems. pp. 7–12. SRAS '12 (2012)

[47] Jin, X., Krishnan, R., Sandhu, R.S.: A unified attribute-based access control model covering dac, MAC and RBAC. In: Data and Applications

Security and Privacy XXVI - 26th Annual IFIP WG 11.3 Conference, DB-Sec 2012, Paris, France, July 11-13,2012. Proceedings. pp. 41–55 (2012)

[48] Jin, X., Sandhu, R., Krishnan, R.: Rabac: Role-centric attribute-based access control. In: Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security: Computer Network Security. pp. 84–96. MMM-ACNS'12 (2012)

[49] Karim, R., Dhawan, M., Ganapathy, V., Shan, C.: An analysis of the Mozilla Jetpack extension framework. In: ECOOP. pp. 333–355 (2012)

[50] Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 389–391 (2014)

[51] Langerak, R., Brinksma, E., Katoen, J.: Causal ambiguity and partial orders in event structures. In: CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings. pp. 317–331 (1997)

[52] Li, N., Tripunitara, M.V., Bizri, Z.: On mutually exclusive roles and separation-of-duty. ACM Trans. Inf. Syst. Secur. 10(2) (2007)

[53] Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome extensions: Threat analysis and countermeasures. In: NDSS (2012)

[54] Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: APLAS. pp. 307–325 (2008)

[55] Maffeis, S., Taly, A.: Language-based isolation of untrusted JavaScript. In: CSF. pp. 77–91 (2009)

[56] Morse, J., Ramalho, M., Cordeiro, L.C., Nicole, D.A., Fischer, B.: ESBMC 1.22 - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings. pp. 405–407 (2014)

[57] Nielson, F., Nielson, H.R.: Flow logic and operational semantics. Electronic Notes on Theoretical Computer Science 10, 150–169 (1997)

[58] Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)

[59] Nielson, H.R., Nielson, F., Pilegaard, H.: Flow logic for process calculi. ACM Computing Surveys 44(1), 1–39 (2012)

[60] Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation 9, 53–58 (2014 (published 2015))

[61] Politz, J.G., Carroll, M.J., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. In: DLS. pp. 1–16 (2012)

[62] Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: Adsafety: Type-based verification of JavaScript sandboxing. In: USENIX Security Symposium (2011)

[63] Porter Felt, A., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)

[64] Ranise, S., Truong, A.T., Armando, A.: Boosting model checking to analyse large ARBAC policies. In: Security and Trust Management - 8th International Workshop, STM 2012, Pisa, Italy, September 13-14, 2012, Revised Selected Papers. pp. 273–288 (2012)

[65] Ranise, S., Truong, A.T., Viganò, L.: Automated analysis of RBAC policies with temporal constraints and static role hierarchies. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 2177–2184 (2015)

[66] Roesner, F., Kohno, T., Wetherall, D.: Detecting and defending against third-party tracking on the web. In: NSDI. pp. 155–168 (2012)

[67] Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)

[68] Sandhu, R., Bhamidipati, V., Munawer, Q.: The arbac97 model for role-based administration of roles. ACM Trans. Inf. Syst. Secur. 2(1) (Feb 1999)

[69] Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.R.: Policy analysis for administrative role-based access control. Theor. Comput. Sci. 412(44), 6208–6234 (2011)

[70] Stoller, S.D., Yang, P., Gofman, M.I., Ramakrishnan, C.R.: Symbolic reachability analysis for parameterized administrative role-based access control. Computers & Security 30(2-3), 148–164 (2011)

[71] Stoller, S.D., Yang, P., Ramakrishnan, C.R., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 445–455 (2007)

[72] Tan, K., Crampton, J., Gunter, C.A.: The consistency of task-based authorization constraints in workflow systems. In: 17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA. p. 155 (2004)

[73] Uzun, E., Atluri, V., Vaidya, J., Sural, S., Ferrara, A.L., Parlato, G., Madhusudan, P.: Security analysis for temporal role based access control. Journal of Computer Security 22(6), 961–996 (2014)

[74] Wang, Q., Li, N.: Satisfiability and resiliency in workflow authorization systems. ACM Trans. Inf. Syst. Secur. 13(4), 40 (2010)

[75] Wang, Q., Li, N., Chen, H.: On the security of delegation in access control systems. In: Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings. pp. 317–332 (2008)

[76] Winskel, G.: Event structures. In: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986. pp. 325–392 (1986)

[77] Winskel, G.: An introduction to event structures. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings. pp. 364–397 (1988)

[78] Yang, P., Xie, X., Ray, I., Lu, S.: Satisfiability analysis of workflows with control-flow patterns and authorization constraints. IEEE Trans. Services Computing 7(2), 237–251 (2014)