Compositional Typed Analysis of ARBAC Policies

Stefano Calzavara, Alvise Rabitti and Michele Bugliesi
Dipartimento di Scienze Ambientali, Informatica e Statistica (DAIS)
Università Ca’ Foscari Venezia

Abstract—Model-checking is a popular approach to the security analysis of ARBAC policies, but its effectiveness is hindered by the exponential explosion of the ways in which different users can be assigned to different role combinations. In this paper we propose a paradigm shift, based on the observation that, while verifying ARBAC by exhaustive state search is complex, realistic policies often have rather simple security proofs, and we propose to use types as an effective tool to leverage this simplicity. Concretely, we present a static type system to verify the security of ARBAC policies, along with a sound and complete type inference algorithm used to automate the verification process. We then introduce compositionality results, which identify sufficient conditions to preserve the security guarantees obtained by the verification of different sub-policies when these sub-policies are combined together: this compositional reasoning is crucial when policy administration is highly distributed and naturally supports the security analysis of evolving ARBAC policies. We evaluate our approach by implementing TAPA, a static analyser for ARBAC policies based on our theory, which we test on a number of relatively large, publicly available policies from the literature.

I. INTRODUCTION

Role-based access control (RBAC) is one of the most widespread authorization models deployed nowadays [1]. In RBAC, permissions are collected into abstractions known as roles, which in turn are assigned to individual users (subjects). User privileges only depend on the assigned roles: since the number of roles is typically static and much smaller than the number of users, RBAC greatly simplifies the deployment of authorization policies for large organizations. Administrative RBAC (ARBAC) promotes the role-based abstraction into role administration itself, by allowing policy writers to specify which roles are entitled to assign or revoke other roles to users. On the one hand, ARBAC is particularly intuitive and effective, especially when role administration is highly distributed. On the other hand, however, writing correct ARBAC policies is notoriously hard, since the privileges assigned to individual users dynamically change due to administrative actions, potentially leading to privilege escalations by untrusted users or to conflict of interest scenarios, where two roles which are intended as mutually exclusive are instead assigned to the same user of the system [2], [3]. An automated analysis of ARBAC policies which captures their inherently dynamic nature is thus highly desirable and substantial research work has focused on this problem in the last decade, see, e.g., [4], [5], [6], [7], [8]. Though with notable differences in their technical development, advantages and limitations, the very large majority of these papers relies on model-checking techniques: the common idea is to explore (an abstraction of) the state-transition system representing the different user-role combinations permitted by the ARBAC policy, looking for possible security violations, such as the possession of a security-critical role by an untrusted user of the system. The specific state exploration process may be geared towards error finding or policy verification, i.e., proving the absence of security flaws.

Unfortunately, it is extremely hard to verify the security of ARBAC by an exhaustive exploration of the state space, due to the exponential explosion of the ways in which different users can be assigned to different role combinations [9], [6]. Researchers have tackled this problem in many orthogonal ways, from the identification of fragments of ARBAC which enable more efficient security analyses [4], [2], [10] to the design of abstraction/pruning techniques used to reduce the size of the state space to explore [11], [12], [13], [14]. In the end, however, when it comes to ARBAC verification, all these approaches rely on sophisticated (abstract) model-checking frameworks. In this paper we propose a paradigm shift, based on the observation that, while verification by exhaustive state search is complex, ARBAC policies most often have rather simple security proofs, and we propose to use types and type systems as effective tools to leverage this simplicity and develop a novel framework for static verification.

Compared to other approaches, type systems have several distinctive advantages. Being syntactic in nature, they enable a rather direct inspection of the ARBAC policies to analyse, which in turn capture the role assignment invariants which constitute the security proof intended by the policy writer. Since most of these proofs are easy, typing saves the need for awkward syntactic restrictions on the ARBAC policy to verify, and similarly for the complex policy transformations required to make model-checking reasonably efficient. The syntactic nature of typing is also helpful when a policy is not secure, since the inability to type-check one of the policy rules likely suggests the “culprit” of the security violation, providing the policy writer with useful information on how to fix the flaw.

But the major advantage of types is their ability to support compositional security proofs. Under appropriate conditions, different sub-policies can be verified independently and combined together into a larger policy, without weakening any of the security guarantees provided by verification. This is crucial when policy administration is highly distributed and different (possibly mutually distrusted) administrators state different security requirements. This compositional reasoning holds promise for ensuring the scalability of verification by design, and naturally supports the security analysis of evolving ARBAC policies, where new policy rules are added by the administrators to accommodate late security needs [15], [16].

All in all, we argue that types have a number of desirable features which make them a promising tool for the verification of ARBAC policies and we set forth a first investigation of their benefits from both a theoretical and a practical perspective. Clearly, typing cannot entirely subsume model-checking in ARBAC verification, as type systems necessarily provide an
over-approximated security analysis when more precise solutions may be sometimes desirable in practice. In our view, however, types constitute an additional analysis layer with clear practical benefits (and an interesting theory).

A. Contributions

Our contributions can be summarized as follows:

1) we present a static type system to verify the security of ARBAC policies by proving the absence of undesired privilege escalations and conflict of interest scenarios;
2) we describe a sound and complete type inference algorithm, to automatically build a security proof from an ARBAC policy by constraint solving;
3) we discuss useful compositionality results, which identify sufficient conditions to preserve the security guarantees obtained by the verification of different sub-policies when these sub-policies are combined together;
4) we present simple syntactic transformations for ARBAC policies, which ease the construction of a type-based security proof and boost the expressiveness of the analysis on realistic examples;
5) we implement TAPA (Typed Arbac Policy Analyser), a static analyser for ARBAC policies based on our theory; we evaluate both the efficiency and the expressiveness of our approach by type-checking a number of relatively large, publicly available policies from the literature.

B. Related Work

Security Analysis of ARBAC: many research papers have studied the computational complexity of the security analysis of (restricted fragments of) the ARBAC model [2], [3], [6], [9], [4], [17]. Model-checking has been first proposed as a valuable tool for ARBAC verification in [6] and largely adopted since then [7], [18], [19], [8], [20], [10]. Many of these works consider strong syntactic restrictions on the format of the ARBAC policies to simplify the security analysis, e.g., the absence of negative preconditions in the role assignment rules or the separate administration assumption [4]. In this work, we do not assume any of these restrictions, which severely limit the expressiveness of the ARBAC model.

Abstraction techniques for the analysis of ARBAC systems have been first independently proposed by Bugliesi et al. [12] and Ferrara et al. [11]. More specifically, the first paper advocates the use of abstract model-checking techniques to overcome the state space explosion problem affecting plain model-checking solutions, while the second one proposes a program verification approach based on abstract interpretation. This abstract interpretation approach shares a similar mindset with our present proposal, but lacks many of the advantages enabled by typing, most notably a compositionality result. In more recent work, the authors of [11] proposed an aggressive pruning technique to significantly simplify the security analysis of large ARBAC policies [13]. The proposal is very effective at shrinking even complicated policies and constitutes a useful preprocessing step for many different analyses, but it is based on convoluted syntactic transformations, which significantly complicate error reporting for insecure policies [21]. The pruning algorithm is non-compositional and goal-based: if new rules are added to the original policy or additional security properties have to be verified, the administrator has to run the algorithm again and reassess the results of verification. This limits the approach to a largely centralized setting.

Compositionality in Access Control: policy composition is an important topic in access control and several formal frameworks have been proposed to study its foundations, most notably Belnap logic [22] and policy algebras [23], [24], [25]. The main goal of these works is to isolate meaningful combinatorial structures for policy composition and/or to identify conditions which ensure that some composition of policies presents desirable features, e.g., lack of conflicts on authorization decisions. Policy composition in ARBAC is naturally interpreted just as the union of different policies and no conflict may arise upon such composition, since the current ARBAC standard does not include negative permissions [1]. To the best of our knowledge, compositional verification results for ARBAC like the ones we present in this paper have never been proposed before.

One of the advantages of compositionality is the ability to reuse previous verification results: in the context of ARBAC, some incremental analysis algorithms which support the verification of evolving policies have been proposed [15], [16]. Both these works describe special model-checking algorithms which exploit the information computed from previous analyses to update the verification results when the original policy changes. In both cases, the evolution of the policy is represented by the addition/deletion of a single rule to/from the original policy, hence it is not obvious whether these algorithms are efficient when a relatively large sub-policy is included into the original one.

Types for Access Control: types have been extensively used to prove that programs or distributed processes comply with an underlying access control policy, see, e.g., [26], [27], [28], [29], [30]. In the realm of ARBAC, types have been studied by Braghin et al. [31] and Jagadeesan et al. [32]. The first paper proposes a type system which over-approximates the minimum set of roles that guarantees that all the executions of a distributed process (in an extension of the π-calculus) are successful. The second paper develops a similar analysis for a λ-calculus and a dual analysis which deduces the minimal set of roles activated in all the execution paths. These papers use types to verify programs or processes subject to an access control policy, rather than the policy itself, hence they bear only limited similarities to the present approach.

C. Structure of the Paper

Section II describes the operational semantics of ARBAC and defines a formal notion of safety. Section III presents the type system and discusses the security guarantees it provides. Section IV introduces the type inference algorithm. Section V presents the compositionality results. Section VI proposes an intuitive policy transformation which simplifies the construction of type-based security proofs. Section VII describes TAPA and presents the experimental results. Section VIII concludes. For space reasons, most of the proofs of the formal results are only available in the online long version [33].

II. THE ARBAC MODEL

We consider a simple operational semantics for the ARBAC model [1]. For the sake of simplicity, we do not represent
sessions and the role hierarchy. Notice that any hierarchical policy can be transformed into a non-hierarchical one using a standard transformation [17]. A similar model with the same simplifications has been adopted by Ferrara et al. [11].

A. ARBAC Systems

We presuppose an unbounded set of users $\tilde{U}$ and a finite set of roles $\tilde{R}$.

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

A can-assign rule $(r_u, R_p, R_o, r_t) \in CA$ states that a user with role $r_u$ 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 $R_o$ (the negative preconditions). A can-revoke rule $(r_u, r_t) \in CR$, instead, states that a user with role $r_u$ can unconditionally revoke role $r_t$ from any user.

We say that a role $r_t$ is administrative if it appears as the first element of a can-assign or a can-revoke rule. In this paper we identify administrative roles with regular roles, i.e., we do not assume a separate administration policy [4]. Separate administration requires that administrative roles are never assigned or revoked, which greatly simplifies the security analysis, but it is unrealistic in many settings [5].

A configuration consists of a finite set of users, along with the roles assigned to them.

Definition 2 (Configuration): An ARBAC configuration is a pair $\sigma = (U, UR)$, where:
- $U \subseteq \tilde{U}$ is a finite set of users;
- $UR \subseteq U \times \tilde{R}$ is the user-to-role assignment relation.

In the following, we let $UR(u) = \{r \mid (u, r) \in UR\}$.

The operational semantics of the ARBAC model is defined in terms of the changes which can be performed to an initial configuration, based on the administrative actions enabled by the underlying policy. The evolution of a configuration $\sigma$ under a policy $\mathcal{P}$ is defined by the reduction relation $\mathcal{P} \triangleright \sigma \longrightarrow^* \sigma'$ given in Table I.

Rules (R-ASSIGN) and (R-REVOKE) formalize the intuitive semantics of the can-assign and can-revoke relations respectively, while rules (R-JOIN) and (R-LEAVE) account for the dynamic joining and leaving of users. Since users joining a configuration are drawn from an unbounded set $\tilde{U}$, the semantics gives rise to an infinite-state transition system. We let $\longrightarrow^*$ stand for the reflexive-transitive closure of $\longrightarrow$.

Definition 3 (System): An ARBAC system is a pair $\mathcal{S} = (\mathcal{P}, \sigma)$ including a policy $\mathcal{P}$ and an initial configuration $\sigma$.

B. Security Analysis

Many existing works study the security of ARBAC systems in terms of the so-called role reachability problem [17]: given a system $\mathcal{S} = (\mathcal{P}, \sigma)$, a user $u$ and a target role $goal$, does there exist a configuration $\sigma' = (U, UR)$ such that $\mathcal{P} \triangleright \sigma \longrightarrow^* \sigma'$ and $(u, goal) \in UR$?

This is a very useful property, since many classic security queries can be encoded in terms of it [34], [11]. Here, however, we propose a more structured approach to ARBAC verification, which enables a more powerful reasoning and fosters compositionality, without sacrificing expressiveness (see below).

We presuppose the existence of a security lattice $(\tilde{L}, \sqsubseteq)$ with a bottom element. In our examples we always consider a two-point lattice $\tilde{L} = \{L, H\}$ with $L \sqsubseteq H$, but the theory is defined for an arbitrary lattice with bottom.

Definition 4 (Security Labelling): A security labelling is a pair $\mathcal{L} = (\gamma, \delta)$, where:
- $\gamma : \tilde{U} \rightarrow \tilde{L}$ is a mapping from users to security labels, representing their level of trust;
- $\delta : 2^{\tilde{R}} \rightarrow \tilde{L}$ is a monotonic mapping from sets of roles to security labels, representing the least level of trust needed for the possession of a given role combination.

We always presuppose that $\delta(\emptyset) = \bot$.

Intuitively, a security labelling $\mathcal{L} = (\gamma, \delta)$ represents a meta-policy dictating constraints over role assignment. By stipulating that $\delta(R) = H$ for a given set of roles $R$, we specify that the role combination $R$ must never be granted to any user $u$ such that $\gamma(u) = L$. The definition of an appropriate security labelling depends on the application scenario, e.g., the enterprise/organization modelled by the ARBAC system, which defines the identity of the users and the semantics of the permissions granted by the individual roles.

We define the safety of an ARBAC system in terms of an underlying security labelling.

Definition 5 (Admissible Configuration): A configuration $\sigma = (U, UR)$ is admitted by the security labelling $\mathcal{L} = (\gamma, \delta)$, written $\mathcal{L} \models \sigma$, iff for every $u \in U$ we have $\delta(UR(u)) \subseteq \gamma(u)$.

Definition 6 (Safety): An ARBAC system $\mathcal{S} = (\mathcal{P}, \sigma)$ is safe with respect to the security labelling $\mathcal{L}$ iff, for any $\sigma'$ such that $\mathcal{P} \triangleright \sigma \longrightarrow^* \sigma'$, we have $\mathcal{L} \models \sigma'$.

It is easy to show that proving safety is enough to guarantee that a given role is unreachable. Specifically, given a security labelling $\mathcal{L} = (\gamma, \delta)$ such that $\gamma(u) = L$ for some user $u$ and $\delta(\{goal\}) = H$ for some goal role, then proving safety with respect to $\mathcal{L}$ ensures that role $goal$ is not reachable by $u$. Notice also that this safety notion is expressive enough to directly encode static separation of duty constraints, expressing for instance that no user should be able to acquire the roles Doctor and Patient at the same time [1].

We conclude this section by observing that our notion of security labelling is reminiscent of previous proposals which combine ARBAC with an underlying security ordering, most notably the SHRBAC model by Crampton [35]. In SHRBAC, a seniority function assigns to each role and user of the system a level of seniority from an underlying poset, to enforce security properties like “role activation”, which ensures that each user is at least as senior as the roles she activates. However, the enforcement of this property is purely dynamic in SHRBAC, where a policy monitor applies order-based policies like role
activation; conversely, the security labelling we propose here does not alter the semantics of the original ARBAC model and just represents an encoding of the security goals one wants to ensure by static typing, without any need of additional runtime checks by a custom reference monitor.

C. Examples

Let \( \tilde{R} = \{r_a, r_1, r_2, r_3\} \) and \( \tilde{U} = \{u_1, u_2\} \). We assume a security labelling \((\gamma, \delta)\) such that \(\gamma(u_1) = H, \gamma(u_2) = L\) and:

\[
\forall i \in \{1, 2, 3\} : \quad \delta(\{r_i\}) = L,
\delta(\{r_a\}) = H,
\delta(\{r_1, r_2\}) = H.
\]

In words, user \(u_1\) is trusted, while user \(u_2\) is not; role \(r_a\) and the combination of roles \(\{r_1, r_2\}\) should only be given to trusted users, while no restriction is put on the possession of the individual roles \(r_1, r_2, r_3\). In all the examples, we assume the same initial configuration \(\sigma = (\tilde{U}, \{(u_1, r_a)\})\).

Example 1 (Mutual Exclusion): Let:

\[ CA_1 = \{(r_a, \emptyset, \{r_1\}, r_2), (r_a, \emptyset, \{r_2\}, r_1)\}. \]

The system \(\{(CA_1, CR)\}, \sigma\) is safe for any \(CR\), since \(r_a\) is only given to the high user \(u_1\), while roles \(r_1, r_2\) are mutually exclusive, hence they cannot be assigned to any user at the same time.

Example 2 (Secure Flow): Let:

\[ CA_2 = \{(r_a, \{r_3\}, \emptyset, r_2), (r_a, \emptyset, \emptyset, r_1)\}. \]

The system \(\{(CA_2, CR)\}, \sigma\) is safe for any \(CR\), since \(r_2\) can only be assigned to users who are granted \(r_a\), i.e., it can only be assigned to high users. This means in particular that only high users can possess \(r_1\) and \(r_2\) at the same time.

Example 3 (Irrevocable Guard): Let:

\[ CA_3 = \{(r_a, \{r_3\}, \emptyset, r_1), (r_a, \emptyset, \{r_1\}, r_2), (r_a, \emptyset, \{r_2\}, r_3)\}. \]

The system \(\{(CA_3, CR)\}, \sigma\) is safe for any \(CR\) which does not allow the revocation of \(r_3\), i.e., for any \(CR\) such that there does not exist \(r\) such that \((r, r_3) \in CR\). The observation here is that \(r_2\) and \(r_3\) are mutually exclusive, and to grant \(r_1\) we must first assign \(r_3\) (which excludes \(r_2\)).

Notice that the system would not be safe if \(r_3\) could be revoked, since the low user \(u_2\) with no role assigned could first acquire \(r_3\), then acquire \(r_1\), revoke \(r_3\) and finally acquire \(r_2\), thus getting the high combination of roles \(\{r_1, r_2\}\).

III. TYPES FOR ARBAC VERIFICATION

The core idea behind any type-based analysis is to identify useful invariants which hold true for selected resources of interest. These invariants are distilled into types assigned to the resources and enforced by a set of typing rules, dictating syntactic checks which preserve the invariants. In a sound type system, this is enough to entail a semantic property of the system under study: in this paper, we assign types to roles to prove the safety of ARBAC systems.

A. Types and Typing Environments

In our setting, a type \(\tau \in \tilde{T}\) is a triple \([\tau^+, \tau^-]\). If a role \(r\) is given this type, then \(r\) can be assigned only to users with a label \(\ell' \sqsubseteq \ell\), and users who are granted \(r\) must always have all the roles in the set \(\tau^+\) and none of the roles in the set \(\tau^-\).

Given a type \(\tau = [\ell^+, \ell^-]\), we let \(lab(\tau) = \ell\). We say that a type \(\tau = [\ell^+, \ell^-]\) is consistent, written \(cons(\tau)\), iff \(\ell^+ \cap \ell^- = \emptyset\). If a role is given an inconsistent type, it can never be assigned to any user in a well-typed system, since the intended invariant predicated by the type would lead to a contradiction.

We let \(\Gamma : \tilde{R} \rightarrow \tilde{T}\) stand for a typing environment, assigning to each role a corresponding type. We write \(\Gamma^+(r) = \tau^+\) and \(\Gamma^-(r) = \tau^-\) whenever \(\Gamma(r) = \tau = [\ell^+, \ell^-]\) for some label \(\ell\). For a set of roles \(R\), let \(\Gamma^+(R) = \bigcup_{r \in R} \Gamma^+(r)\) and \(\Gamma^-(R) = \bigcup_{r \in R} \Gamma^-(r)\).

B. Closures

Given a typing environment, we can exploit the syntactic information contained therein to soundly infer additional invariants over role assignment. For instance, pick three roles \(r_1, r_2, r_3\) and assume that \(\Gamma^+(r_2) = \{r_1\}\) and \(\Gamma^+(r_3) = \{r_2\}\): then any user who is assigned \(r_2\) must have \(r_1\) and any user who is assigned \(r_3\) must have \(r_2\). It is then natural to conclude that any user who is assigned \(r_3\) must be assigned also \(r_1\), even though this is not explicit in the typing environment.

We formalize and generalize this reasoning by defining a closure operation \((R^+_1, R^-_1) \triangleright_1 (R^+_2, R^-_2, R^+_3, R^-_3)\). The idea is that, given a set of assigned roles \(R^+_1\) and a set of unassigned roles \(R^-_1\), we exploit the type information in \(\Gamma\) to build a new set of assigned roles \(R^+_2 \supseteq R^+_1\) and a new set of unassigned roles \(R^-_2 \supseteq R^-_1\). The closure operation plays a central role in the typing rules in the next section.

Definition 7 (Closure): A pair \((R^+, R^-)\) is closed under \(\Gamma\) iff all the following clauses hold true:
We now explain the typing rules for role revocation. Let \( cr = (r_a, r_t) \) be a can-revoke rule: if either \( r_a \) or \( r_t \) is inconsistent, then \( cr \) will never be fired and we can trivially accept it by rule (CR-SINGLE). Otherwise, the typing rule has to ensure that \( r_t \) is not assumed to be always given in combination with some other role \( r \neq r_t \) according to the information in the typing environment, i.e., \( r_t \notin \Gamma^+(r) \). Indeed, if it was the case, the revocation of \( r_t \) in a configuration where a user is also assigned the role \( r \) would break the expected invariant that the possession of \( r \) implies the possession of \( r_t \). Typing multiple can-revoke rules just amounts to typing the individual rules by rule (CR-UNION).

Having discussed the typing rules for policies, we now present the typing rules for configurations, deriving the second kind of judgements. These rules are simpler than the other ones, since they just ensure that any well-typed configuration is compliant with the information stored in the typing environment, according with the informal reading of types we introduced in Section III-A.

**Definition 8 (Well-typed Configuration):** Let \( \sigma = (U, UR) \) and \( \mathcal{L} = (\gamma, \delta) \), we write \( \Gamma; \mathcal{L} \vdash \sigma \) iff for all \( u \in U \) and all \( r \in UR(u) \) we have:

1. \( \text{lab}(\Gamma(r)) \subseteq \gamma(u) \);
2. \( \Gamma^-(r) \cap UR(u) = \emptyset \);
3. \( \Gamma^+(r) \subseteq UR(u) \).

Given a system \( S = (\mathcal{P}, \sigma) \), we write \( \Gamma; \mathcal{L} \vdash S \) as a shortcut for \( \Gamma; \mathcal{P} \vdash \sigma \).

**D. Safety by Typing**

We now discuss how the typing rules can be used to prove the safety of an ARBAC system (Theorem 2 below).

A crucial result is the following subject reduction theorem, which ensures that the invariants collected by the typing environment are preserved in any reachable configuration.

**Theorem 1 (Subject Reduction):** Let \( \Gamma; \mathcal{P} \vdash \sigma \). If \( \mathcal{P} \vdash \sigma \rightarrow \sigma' \), then \( \Gamma; \mathcal{L} \vdash \sigma' \).

The proof of the theorem uses the following lemma, providing a formal characterization of the intuitive semantics of the closure operation (Definition 7).

**Lemma 1 (Implied and Conflicting Roles):** Let \( R, R' \) be two sets of roles. If \( \Gamma; \mathcal{L} \vdash (U, UR) \) and for some \( u \in U \) we have \( R \subseteq UR(u) \) and \( R' \cap UR(u) = \emptyset \), then \( (R, R') \vdash \mathcal{L} \) implies \( R^+ \subseteq UR(u) \) and \( R^- \cap UR(u) = \emptyset \).

**Proof:** By induction on the number of operations involved in the closure construction. If \( (R, R') \) is closed under \( \Gamma \), then \( R^+ = R \) and \( R^- = R' \), hence we are done. Otherwise, we use the induction hypothesis and the assumption \( \Gamma; \mathcal{L} \vdash (U, UR) \) to prove the conclusion. Recall in fact that \( \Gamma; \mathcal{L} \vdash (U, UR) \) ensures that for all \( u \in U \) and all \( r \in UR(u) \) we have \( \Gamma^+(r) \subseteq UR(u) \) and \( \Gamma^-(r) \cap UR(u) = \emptyset \).

In contrast with many type-based analyses, safety is not an immediate consequence of the subject reduction result. In our case, we have to combine together the type information for the different roles and exploit the invariants collected by the typing environment to ensure compliance with respect to the security labelling used for defining safety, thus obtaining a correct security proof. We reuse a closure operation to identify sufficient conditions on the typing environment which allow to prove safety under the desired security labelling.
Definition of $\Gamma \vdash_{rev} CR$:

\[
\begin{align*}
\text{(CR-SINGLE)} & \\
\Gamma \vdash_{rev} (r_a, R) & \text{ if }\forall r \in R^+ \setminus \{r_t\} : r_t \in \Gamma^+(r)\\
\end{align*}
\]

Definition of $\Gamma \vdash_{\text{arg}} CA$:

\[
\begin{align*}
\text{(CA-SINGLE)} & \\
\Gamma \vdash_{\text{arg}} CA & \text{ if }\forall r \in R^+ \setminus \{r_t\} : r_t \in \Gamma^+(r)\\
\end{align*}
\]

**Convention:** in rule (CA-SINGLE) we let $\sqcup_{r \in R^+ \setminus \{r_t\}} \text{lab}(\Gamma(r)) = \bot$ whenever $R^+ = \emptyset$

**Definition 9 (Enforceability):** A security labelling $L = (\gamma, \delta)$ is enforceable by a typing environment $\Gamma$, written $\Gamma \vdash L$, iff for all $R \in \mathcal{R}$, whenever $(R, \emptyset) \not\vdash (R^+, R^-)$, either of the following conditions holds true:

1) $\exists r \in R^+ : \delta(R) \subseteq \text{lab}(\Gamma(r))$

2) $R^+ \cap R^- \neq \emptyset$

In words, for every possible combinations of roles $R$, we require that either (1) one of the roles implied by $R$ is at least as sensitive as $R$ itself, which is enough to ensure that $R$ is not assigned to untrusted users in a well-typed configuration by the first condition of Definition 8; or (2) $R$ is not assignable at all, since there exists at least one role which is both implied by and conflicting with $R$.

Using the subject reduction theorem and Lemma 1, we can prove the main safety result as follows.

**Lemma 2 (Static Safety):** Let $\Gamma \vdash L$. If $\Gamma; L \vdash \sigma$, then $L \models \sigma$.

**Proof:** Let $\sigma = (U, UR)$ and let $L = (\gamma, \delta)$, we pick any user $u \in U$ and we show that $\delta(UR(u)) \subseteq \gamma(u)$. Let $\Gamma = (U(u), \emptyset) \not\vdash (R^+, R^-)$. Since $\Gamma \vdash L$, we have two cases:

1) either $\exists r \in R^+ : \delta(UR(u)) \subseteq \text{lab}(\Gamma(r))$

2) or $R^+ \cap R^- \neq \emptyset$.

By Lemma 1, we have $R^+ \subseteq UR(u)$ and $R^- \cap UR(u) = \emptyset$, hence $R^+ \cap R^- = \emptyset$ and we must be in the first of the two cases. Let then $r \in R^+$ be a role s.t. $\delta(UR(u)) \subseteq \text{lab}(\Gamma(r))$. Since $\Gamma; L \vdash \sigma$ and $r \in R^+ \subseteq UR(u)$, we have $\text{lab}(\Gamma(r)) \subseteq \gamma(u)$. By transitivity we get $\delta(UR(u)) \subseteq \gamma(u)$.

**Theorem 2 (Safety by Typing):** Let $\Gamma \vdash L$. If $\Gamma; L \vdash S$, then $S$ is safe with respect to $L$.

**Proof:** Let $\mathcal{P} \triangleright \sigma \rightarrow^* \sigma'$, by Theorem 1 we know that $\Gamma; L \vdash \sigma'$. Hence, we have $L \models \sigma'$ by Lemma 2.

**E. Examples**

Here, we use the typing rules and the main theorem given above to build type-based security proofs of the three examples given in Section II-C. Recall that in all cases we want to prove safety with respect to a security labelling $L$ dictating that the dangerous combination of roles $\{r_1, r_2\}$ is not assigned to low users. These are toy examples, which we use to present the type system: an experimental evaluation on realistic policies is given in Section VII-B.

**Example 1 (Mutual Exclusion):** Let:

$\Gamma = r_a : H[\emptyset, \emptyset], r_1 : L[\emptyset, \{r_2\}], r_2 : L[\emptyset, \{r_1\}]$,

we have:

\[
\begin{align*}
R^+ = \{r_a\} & \quad R^- = \{r_1, r_2\} \\
\text{lab}(\Gamma(r_2)) = \emptyset \subseteq L & \quad \forall r \in \{r_a, r_1\} : r_2 \notin \Gamma^-(r) \\
\Gamma^-(r_2) = \{r_1\} \subseteq R^- \setminus \{r_2\} & \quad \Gamma^+(r_2) = \emptyset \subseteq R^+ \cup \{r_2\} \\
\Gamma \vdash (r_a, \emptyset, \{r_2\}, r_1)
\end{align*}
\]

The proof of $\Gamma \vdash (r_a, \emptyset, \{r_2\}, r_1)$ is analogous.

We then observe that $(\{r_1, r_2\}, \emptyset) \not\vdash (\{r_1, r_2\})$, hence we have $\Gamma \vdash L$ by condition (2) of Definition 9 and the system is safe.

**Example 2 (Secure Flow):** Let:

$\Gamma = r_a : H[\emptyset, \emptyset], r_1 : L[\emptyset, \emptyset], r_2 : L[\{r_a\}, \emptyset]$,

we have:

\[
\begin{align*}
R^+ = \{r_a\} & \quad R^- = \{r_2\} \\
\text{lab}(\Gamma(r_2)) = \emptyset \subseteq L & \quad \forall r \in \{r_a, r_1\} : r_2 \notin \Gamma^-(r) \\
\Gamma^-(r_2) = \emptyset \subseteq R^- \setminus \{r_2\} & \quad \Gamma^+(r_2) = \emptyset \subseteq R^+ \cup \{r_2\} \\
\Gamma \vdash (r_a, \emptyset, \{r_2\}, r_2)
\end{align*}
\]

We then prove:

\[
\begin{align*}
R^+ = \emptyset & \quad R^- = \{r_1, r_2\} \\
\text{lab}(\Gamma(r_1)) = \emptyset \subseteq L & \quad \forall r \in \{r_a, r_1\} : r_1 \notin \Gamma^-(r) \\
\Gamma^-(r_1) = \emptyset \subseteq R^- \setminus \{r_1\} & \quad \Gamma^+(r_1) = \emptyset \subseteq R^+ \cup \{r_1\} \\
\Gamma \vdash (r_a, \emptyset, \{r_2\}, r_1)
\end{align*}
\]
we think that most of the useful invariants intended by policy are much less efficient than the one we present here. Moreover, agnostic to users. Since real-world configurations can be huge, labels, and the corresponding proof of type safety would be assigned the role. For instance, we could have types which are parametric with respect to the label of the user who is sive type system, where the invariants about role assignment F .

Discussion

To conclude the proof, we observe that \((\Gamma, r) \models \emptyset \cup \{r_1, r_2, r_3\}\). The fact that \(r_3\) is not revocable, we could still construct a security proof for the policy by \(\Gamma\). Since \(r_2\) was revocable, \(L\) is not secure.

\[\text{Example 3 (Irrevocable Guard): Let:} \]

\[\Gamma = r_0 : H[\emptyset] , r_1 : L[\{r_3\}] , r_2 : L [\emptyset , \{r_2\}], r_3 : L \{\emptyset , \{r_2\}\}, \]

we have:

\[\begin{align*}
R^+ &= \{r_3\} & R^- &= \{r_1, r_2\} \\
lab(\Gamma(r_1)) &= L \subseteq lab(\Gamma(r_3)) & \forall r \in \{r_1, r_3\} : r_1 \notin \Gamma^-(r) \\
\Gamma^-(r_2) &= \emptyset \subseteq R^- \setminus \{r_2\} & \Gamma^+(r_1) &= \{r_3\} \subseteq R^+ \cup \{r_1\}
\end{align*}\]

\[\Gamma \vdash (r_3, r_1, r_2, r_1)\]

Then, we observe that:

\[\begin{align*}
R^+ &= \emptyset & R^- &= \{r_1, r_2, r_3\} \\
lab(\Gamma(r_2)) &= L \subseteq L & \forall r \in \{r_2\} : r_2 \notin \Gamma^-(r) \\
\Gamma^-(r_2) &= \{r_3\} \subseteq R^- \setminus \{r_2\} & \Gamma^+(r_2) &= \emptyset \subseteq R^+ \cup \{r_2\}
\end{align*}\]

Finally, we have:

\[\begin{align*}
R^+ &= \emptyset & R^- &= \{r_1, r_2, r_3\} \\
lab(\Gamma(r_3)) &= L \subseteq L & \forall r \in \{r_3\} : r_3 \notin \Gamma^-(r) \\
\Gamma^-(r_3) &= \{r_2\} \subseteq R^- \setminus \{r_3\} & \Gamma^+(r_3) &= \emptyset \subseteq R^+ \cup \{r_3\}
\end{align*}\]

To conclude the proof, we observe that \(\{\{r_1, r_2\}, \emptyset\} \vdash_{\Gamma} \{\{r_1, r_2, r_3\}, \{r_2, r_3\}\}\), hence we have \(\Gamma \vdash_{\emptyset} L\) by condition (2) of Definition 9 and the system is safe.

Notice that to construct this security proof it is required that \(r_3\) cannot be revoked, as observed when discussing the safety of the example. Indeed, since \(r_3 \in \Gamma^+(r_1)\), any can-revoke rule enabling the revocation of \(r_3\) would not satisfy the premise of rule (CR-SINGLE). The fact that \(r_3 \in \Gamma^+(r_1)\), in turn, is crucial to prove \(\Gamma \vdash_{\emptyset} L\).

F. Discussion

In principle, it would be possible to design a more expressive type system, where the invariants about role assignment are parametric with respect to the label of the user who is assigned the role. For instance, we could have types which guarantee that low users are never assigned two given roles in combination, without imposing this restriction on high users: technically, it would be enough to enrich the syntax of types, so as to keep track of different invariants for different security labels, and the corresponding proof of type safety would be a straightforward extension of the current one. However, this kind of guarantee exclusively depends on the initial configuration and not on the policy itself, since the latter is deliberately agnostic to users. Since real-world configurations can be huge and they are typically much bigger than the corresponding policies, the resulting type-based analysis would likely be much less efficient than the one we present here. Moreover, we think that most of the useful invariants intended by policy administrators are hidden in the policy itself rather than in the initial configuration, thus we privilege the former in our investigation.

We also notice that it would be possible to extend the type system to include negative information, capturing invariants which are true when a given role is not assigned. From a theoretical perspective, the resulting formalism would be more expressive than the present one, but the additional value in the real world is unclear. We think that most of the useful invariants for real policies predicate on role assignment, something which seems confirmed by our experiments (see Section VII-B). That said, we do not foresee any significant challenge in extending the type system to include this additional information, if future research highlighted its value.

IV. Type Inference

The typing rules in Section III allow to check whether a given ARBAC system complies with the information stored in the typing environment \(\Gamma\), but they do not provide any constructive way to automatically build a security proof from a system \(S\) and a security labelling \(L\). To fully automate this process and make the analysis practically useful, we then devise a type inference algorithm.

A. Specification

The core idea behind the type inference algorithm is to generate a set of constraints from a system \(S\) and a security labelling \(L\) in a syntax-directed way: any solution to the constraints is a typing environment \(\Gamma\) such that \(\Gamma; L \vdash S\) and \(\Gamma \vdash_{\emptyset} L\). The main challenge here is ensuring the efficiency of constraint solving, most notably by dealing with the closure operations used to type-check the can-assign rules and to ensure the enforceability (Definition 9) of the security labelling, without sacrificing the completeness of the type inference.

Formally, we define a function \(\llbracket \cdot \rrbracket^L\) to generate a set of constraints from the system to analyse with respect to \(L\). The definition relies on three auxiliary functions \(\llbracket \cdot \rrbracket_{\text{asg}}, \llbracket \cdot \rrbracket_{\text{rev}}\) and \(\llbracket \cdot \rrbracket_{\text{conf}}\) generating constraints for the can-assign relation, the can-revoke relation and the initial configuration respectively.

The formal definition of the four functions is given in Table III and commented below.

For the function \(\llbracket \cdot \rrbracket_{\text{asg}}\) the generated constraints just correspond to a bottom-up reading of the typing rules for the can-assign relation in Table II, so that a solution to the constraints will be an environment \(\Gamma\) which type-checks the can-assign relation by construction. Similarly, \(\llbracket \cdot \rrbracket_{\text{conf}}\) mimics the conditions dictated by Definition 8, to guarantee that the initial configuration will be well-typed. The most interesting definitions are those of \(\llbracket \cdot \rrbracket_{\text{asg}}\) and \(\llbracket \cdot \rrbracket^L\).

The definition of \(\llbracket \cdot \rrbracket_{\text{asg}}\) reminds a bottom-up reading of the typing rules for the can-assign relation in Table II, but with the important difference that, given \(ca = (r_3, R_\text{asg}, R_\text{in}, r_1)\), the closure operation \((R_\text{asg}, R_\text{in}) \cup \{r_1\}\) used to type-check \(ca\) under \(\Gamma\) is subsumed by a stronger syntactic requirement on \(\Gamma\) itself. Specifically, let \(\tilde{\Gamma} : \tilde{R} \rightarrow 2^H\) be defined as follows:

\[\forall r \in \tilde{R} : \tilde{\Gamma}(r) = \{r\} \cup \{r' \in \tilde{R} | r \in \Gamma^+(r')\},\]
TABLE III Constraint generation for type inference, where $\mathcal{P} = (CA, CR)$ and $\mathcal{L} = (\gamma, \delta)$

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[(r_a, r_t)]_{\text{rev}}$</td>
<td>${ \text{cons}(\Gamma(r_a)) \land \text{cons}(\Gamma(r_t)) \Rightarrow \neg \exists r \in R \setminus { r_1 } : r_t \in \Gamma^+(r) }$</td>
</tr>
<tr>
<td>$[CR \cup CR']_{\text{rev}}$</td>
<td>$[CR]<em>{\text{rev}} \cup [CR']</em>{\text{rev}}$</td>
</tr>
<tr>
<td>$[\emptyset]_{\text{rev}}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$[(r_a, R_p, R_n, r_t)]_{\text{arg}}$</td>
<td>${ \text{cons}(\Gamma(r_a)) \land (\Gamma^+(R_p) \cap (\Gamma(R_n \cup { r_1 }) \cup \Gamma^-(R_p)) = \emptyset) \Rightarrow \text{lab}(\Gamma(r_a)) \subseteq \bigcup_{r \in \Gamma^+(R_p)} \text{lab}(\Gamma(r)) \land (\forall r \in R : r \notin \Gamma(R_n \cup { r_1 }) \cup \Gamma^-(R_p) \Rightarrow r_t \notin \Gamma^+(r) \land \Gamma^-(r_t) \subseteq (\Gamma(R_n \cup { r_1 }) \cup \Gamma^-(R_p)) \setminus { r_1 } \land \Gamma^+(r_t) \subseteq \Gamma^+(R_p) \cup { r_1 } }$</td>
</tr>
<tr>
<td>$[CA \cup CA']_{\text{arg}}$</td>
<td>${ \text{CA} }<em>{\text{arg}} \cup { \text{CA} }</em>{\text{arg}}$</td>
</tr>
<tr>
<td>$[\emptyset]_{\text{arg}}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$[(U, UR)]_{\text{conf}}$</td>
<td>${ \forall u \in U : \forall r \in R : r \in UR(u) \Rightarrow \text{lab}(\Gamma(r)) \subseteq \gamma(u) \land \Gamma^-(r) \cap UR(u) = \emptyset \land \Gamma^+(r) \subseteq UR(u) }$</td>
</tr>
<tr>
<td>$[(\mathcal{P}, \sigma)]_{\text{conf}}$</td>
<td>${ \forall R \in \mathcal{D}_L : \exists r \in \Gamma^+(R) : \delta(r) \subseteq \text{lab}(\Gamma(r)) \lor (\Gamma^+(R) \cap \Gamma^-(R) = \emptyset) }$</td>
</tr>
</tbody>
</table>

Definitions: we let:
- $\mathcal{D}_L = \{ R \in 2^R : \forall R' \in R : \delta(R') \subseteq \delta(R) \}$
- $\Gamma(r) = \{ r \} \cup \{ r' \in R : r \in \Gamma^+(r') \}$ for all $r \in R$
- $\Gamma(R) = \bigcup_{r \in R} \Gamma(r)$ for all $R \in 2^R$

and let $\Gamma(R) = \bigcup_{r \in R} \Gamma(r)$; intuitively, $\Gamma(R)$ tracks the roles which are not assigned when none of the roles in $R$ is assigned.

Now, when analysing rule $ca$, we replace the set of assigned roles $R^+$ with $\Gamma^+(R_p)$, i.e., the set of roles implied by the positive preconditions $R_p$, and the set of unassigned roles $R^-$ with $\Gamma(R_n \cup \{ r_1 \}) \cup \Gamma^-(R_p)$, i.e., the set of roles excluded by the negative preconditions $R_n$ (and the target $r_t$) and the set of roles conflicting with the positive preconditions $R_p$. Hence, we replace the closure of the pair $(R_p, R_n \cup \{ r_1 \})$ with some information which is entirely local to the can-assign rule and readily available in the typing environment. Perhaps surprisingly, as we discuss below, this formulation does not entail any loss of precision for the analysis, but it leads to a constraint system which is much easier to solve.

The definition of $\llbracket \square \rrbracket_{\text{conf}}$ exploits the same intuition when generating the constraints dictating that the security labelling $\mathcal{L}$ must be enforceable by the solution. We also notice that the definition of enforceability includes a universal quantification over all the possible sets of roles, but the monotonicity requirement on the security labelling (Definition 4) allows to tame this source of complexity. Indeed, given a security labelling $\mathcal{L} = (\gamma, \delta)$, it is enough for security to focus on the set of the dangerous role combinations:

$\mathcal{D}_L = \{ R \in 2^R : \forall R' \in R : \delta(R') \subseteq \delta(R) \}$,

i.e., on the set of roles where there is a steep in the security labelling. The intuition is that each role combination $R$ such that $\delta(R) \neq \bot$ must contain some $R' \in \mathcal{D}_L$: if we constrain the assignment of $R'$, we implicitly constrain also the assignment of its superset $R$. Notice that, for any realistic security labelling $\mathcal{L}$, we have that $\mathcal{D}_L$ is much smaller than $2^R$.

B. Formal Results

It is straightforward to prove the following (syntactic) soundness result for the type inference algorithm.

**Theorem 3 (Sound Inference):** If $\Gamma$ is a solution for $\llbracket \mathcal{S} \rrbracket_{\text{conf}}$, then $\Gamma ; \mathcal{L} \vdash \mathcal{S}$ and $\Gamma \vdash \mathcal{L}$.

Notice that the converse of the previous result does not hold true in general, since different typing environments may compute the same closures and thus type-check the same policies, but not all these environments are a solution to the constraints generated by the type inference. To exemplify, consider a security labelling $\mathcal{L} = (\gamma, \delta)$ such that $\gamma(u) = \mathcal{L}$ for every user $u \in \hat{U}$ and $\delta\{r_1, r_4\} = \emptyset$. Consider then the following typing environment:

$\Gamma = \begin{cases} r_1 : L \{ \{r_2, \emptyset\}, r_2 : L \{ \{r_3, \emptyset\}, r_3 : L \{ \emptyset, \{r_4\} \}, r_4 : L \{ \emptyset, \{r_1\} \} \end{cases}$

Let $\mathcal{P} = (CA, \emptyset)$ with $CA = \{ \{r_3, \{r_2\}, \emptyset, \{r_1\} \}$ and let:

$\sigma = (\{u_1\}, \{u_1, r_1\}, \{u_1, r_2\}, \{u_1, r_3\})$.

We have that $\Gamma \vdash \mathcal{P}$ as follows:

$R^+ = \{ r_2, r_3 \} \quad R^- = \{ r_1, r_4 \}$

$\text{lab}(\Gamma(r_1)) = L \subseteq L \quad \forall r \in \{ r_2, r_3, r_4 \} : r \notin \Gamma^-(r)$

$\Gamma^-(r_1) = \emptyset \subseteq R^- \setminus \{ r_1 \} \quad \Gamma^+(r_1) = \{ r_2 \} \subseteq R^+ \cup \{ r_1 \}$

Moreover, $\Gamma ; \mathcal{L} \vdash \sigma$, since $u_1$ has $r_1, r_2, r_3$, but not $r_4$. Since $(\{r_1, r_4\}, \emptyset) \vdash \sigma\{\{r_1, r_2, r_3, r_4\}, \{r_1, r_4\}\}$, we have $\Gamma \vdash \sigma' \mathcal{L}$. However, $\Gamma$ is not a solution to the constraints in $\llbracket (\mathcal{P}, \sigma) \rrbracket_{\text{conf}}$, since it does not satisfy the constraints generated by the can-assign rule. For instance, we have to satisfy:

$\forall r \in R : r \notin \{ r_1 \} \cup \Gamma^-(r) \Rightarrow r_1 \notin \Gamma^-(r)$,

which is false, since $r_4 \notin \{ r_1 \} \cup \Gamma^-(r)$, but $r_1 \in \Gamma^-(r_4)$.

This counter-example highlights that there exist typing environments which would allow to prove the security of a given system, but will never be returned by the type inference algorithm. Luckily, however, for any of these environments there exists another typing environment which is a solution to the
constraint system generated by the type inference algorithm, hence the type inference is still complete. To exemplify the intuition, we construct from the previous $\Gamma$ a new environment $\Gamma_p$ by propagating the positive and negative information in $\Gamma$ throughout the types assigned to the individual roles, much in the same spirit of a closure operation, thus getting:

$$
\Gamma_p = r_1 : L \{r_1, r_2, r_3, \{r_1\}\}, r_2 : L \{r_2, r_3, \{r_1\}\},
$$

$$
r_3 : L \{r_3, \{r_1\}\}, r_4 : L \{r_4, \{r_1\}\}.
$$

It is possible to show that $\Gamma_p \vdash L \vdash S$ and $\Gamma_p \vdash \sigma$, and that $\Gamma_p$ is a solution to the constraints in $\mathcal{C}$. Notice in particular that the previously violated constraint is satisfied by $\Gamma_p$.

By generalizing over the example above, we let $\Gamma \vdash$ be the typing environment such that for all $r \in R$:

1. $\text{lab}(\Gamma \downarrow (r)) = \text{lab}(\Gamma(r))$;
2. $(\Gamma \downarrow)^+ (r) = \{r\} \cup \Gamma^-(r) \cup \Gamma^+(\Gamma^+(r))$;
3. $(\Gamma \downarrow)^- (r) = \Gamma^-(r) \cup \Gamma^-(\Gamma^+(r)) \cup \{r' \in \bar{R} \mid \Gamma^+(r') \cap \Gamma^-(r') \neq \emptyset\}$.

Let then $\Gamma_0 = \Gamma$ and $\Gamma_n = (\Gamma_{n-1}) \downarrow$ for each natural $n$, we denote by $\Gamma_m$ the environment $\Gamma_m$ such that $m$ is the largest natural such that $\Gamma_m \vdash$. We can prove the following key technical lemma, which ensures that $\Gamma_m$ captures the same information of $\Gamma$ without any need to compute closures.

**Lemma 3 (Removing Closures):** For any typing environment $\Gamma$ and sets of roles $R, R'$, we have:

1. $(R, R') \downarrow \Gamma (R', R^\ast) \iff (R, R') \downarrow \Gamma (R', R^\ast)$;
2. whenever $(R, R') \downarrow \Gamma (R', R^\ast)$, we have $R^\ast \subseteq \Gamma_\Gamma(R) \cup \Gamma_{\downarrow}(R')$.

The first point of the lemma is used in the proof of the next result, which in turn is important to prove the completeness of the type inference algorithm.

**Lemma 4:** All the following statements hold true:

1. If $\Gamma \vdash P$, then $\Gamma_m \vdash P$;
2. if $\Gamma; L \vdash \sigma$, then $\Gamma_m; L \vdash \sigma$;
3. if $\Gamma \vdash_0 L$, then $\Gamma_m \vdash_0 L$.

Based on this, we can prove the completeness of the type inference algorithm.

**Theorem 4 (Complete Inference):** If $\Gamma; L \vdash S$ and $\Gamma \vdash_0 L$, then $\Gamma_m$ is a solution for $\mathcal{S}_\Gamma$.

**Proof:** Let $S = (P, \sigma)$, by hypothesis we have $\Gamma \vdash P$ and $\Gamma; L \vdash \sigma$ and $\Gamma \vdash_0 L$. By Lemma 4, we have $\Gamma_m \vdash P$ and $\Gamma_m; L \vdash \sigma$ and $\Gamma_m \vdash_0 L$. Recall that, for $P = (CA, CR)$, the judgement $\Gamma_m \vdash P$ means $\Gamma_m \vdash_{CA} CA$ and $\Gamma_m \vdash_{CR} CR$.

Now we observe that $\Gamma_m \vdash_{CR} CR$ implies that $\Gamma_m$ is a solution for $[\mathcal{C}A]_{\mathcal{S}_\Gamma}$, since the constraints in the latter set correspond to a bottom-up reading of the typing rules. For the very same reason, $\Gamma_m; L \vdash \sigma$ implies that $\Gamma_m$ is a solution for $[\mathcal{A}S]_{\mathcal{S}_\Gamma}$. To show that $\Gamma_m \vdash_{CA} CA$ implies that $\Gamma_m$ is a solution for $[\mathcal{A}S]_{\mathcal{S}_\Gamma}$, we observe that the gap between a bottom-up reading of the typing rules and the constraints in $[\mathcal{A}S]_{\mathcal{S}_\Gamma}$ is filled in by the second point of Lemma 3, i.e., the closures under $\Gamma_m$ computed by the typing rules can be replaced by the information locally available in the typing environment. We similarly proceed for the enforceability constraints.

V. COMPOSITIONAL SECURITY PROOFS

One of the biggest advantages of type-based verification is the compositional nature of its security proofs, i.e., the results of the verification of small sub-policies can be combined together to obtain a security proof of a larger policy including them. This is important to ensure the scalability of the type inference algorithm to huge ARBAC policies and to support scenarios where policy administration is strongly distributed.

In the following, we write $P_1 \cup P_2$ and $\sigma_1 \cup \sigma_2$ respectively to stand for the policy/configuration which is obtained by performing the pointwise union of the components of the two policies/configurations.

A. Preliminaries

In the most general setting, the problem of compositional verification can be stated as follows: given $n$ ARBAC systems $S_i = (P_i, \sigma_i)$ which are proved safe against $n$ different security labellings $L_i = (\gamma_i, \delta_i)$, can we prove the security of the system $S = (\bigcup_{i=1}^n P_i, \bigcup_{i=1}^n \sigma_i)$ against each $L_i$?

Our compositionality results only apply to the restricted case where, for each $i, j \in \{1, n\}$, $\sigma_i = \sigma_j$ and $\gamma_i = \gamma_j$. This means that the different safe sub-systems $S_i$ all share the same initial configuration and that the different security labellings $L_i$ assign the same trust level to each user. This is a useful and realistic assumption, since this information is inherent to the nature of the enterprise/organization which is modelled by the ARBAC system. If the verification of different sub-systems is performed by different administrators, these administrators may be entitled to write different sub-policies and to have different views of the dangers connected to the possession of some role combinations, but they must all know the underlying configuration and agree on the trust level of each user. Similar assumptions are reasonable also when dealing with evolving ARBAC policies, where new policy rules are added to accommodate changing security needs.

We now present a number of formal results, which combined together give rise to the main compositionality principle we present in this paper (Theorem 5 below).

B. Compositional Verification of Policies

Assume that $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$, we show how to combine $\Gamma_1$ and $\Gamma_2$ to type-check $P_1 \cup P_2$.

**Definition 10 (Joining Environments):** Let $\Gamma_1$ and $\Gamma_2$ be two typing environments, their join is the typing environment $\Gamma_1 + \Gamma_2$ such that for all $r \in \bar{R}$:

- $\text{lab}(\Gamma_1 + \Gamma_2)(r) = \text{lab}(\Gamma_1(r)) \cup \text{lab}(\Gamma_2(r))$;
- $(\Gamma_1 + \Gamma_2)^+(r) = \Gamma_1^+(r) \cup \Gamma_2^+(r)$;
- $(\Gamma_1 + \Gamma_2)^-(r) = \Gamma_1^-(r) \cup \Gamma_2^-(r)$.

Based on this definition, we prove the following formal result, enabling a compositionality principle for policies.

**Lemma 5:** If $\Gamma_1 \vdash P$ and $\Gamma_2 \vdash P$, then $\Gamma_1 + \Gamma_2 \vdash P$.

Assume then that $\Gamma_1 \vdash P_1$ and $\Gamma_2 \vdash P_2$. To type-check $P_1 \cup P_2$, it is enough to check whether $\Gamma_1 \vdash P_2$ and $\Gamma_2 \vdash P_1$. Indeed, if this is the case, then by the previous theorem we have $\Gamma_1 + \Gamma_2 \vdash P_1$ and $\Gamma_1 + \Gamma_2 \vdash P_2$; by the definition of the typing rules, this allows one to conclude $\Gamma_1 + \Gamma_2 \vdash P_1 \cup P_2$. 
C. Compositional Verification of Configurations

We start by defining a pre-order on security labellings.

**Definition 11 (Security Ordering):** Given two security labellings \( L_1 = (\gamma_1, \delta_1) \) and \( L_2 = (\gamma_2, \delta_2) \), we say that \( L_1 \) is no more restrictive than \( L_2 \), written \( L_1 <: L_2 \), if and only if:

- for all \( u \in \tilde{U} \), \( \gamma_2(u) \subseteq \gamma_1(u) \);
- for all \( R \in \gamma(R) \), \( \delta_1(R) \subseteq \delta_2(R) \).

Let \( L_1 <: L_2 \), then any system which is proved safe with respect to \( L_2 \) is safe also with respect to \( L_1 \) by definition.

**Definition 12 (Joining Labellings):** The join of two security labellings \( L_1 = (\gamma_1, \delta_1) \) and \( L_2 = (\gamma_2, \delta_2) \) is the security labelling \( L_1 \sqcup L_2 = (\gamma, \delta) \) such that, for all \( R \in \gamma(R) \), \( \delta(R) = \delta_1(R) \cup \delta_2(R) \).

Notice that \( L_i <: L_1 \sqcup L_2 \) for any \( i \in \{1, 2\} \), hence proving security with respect to \( L_1 \sqcup L_2 \) is enough to prove security with respect to any of the two security labellings.

**Lemma 6:** If \( \Gamma_1; L_1 \vdash \sigma \) and \( \Gamma_2; L_2 \vdash \sigma \), then we have \( \Gamma_1 + \Gamma_2; L_1 \sqcup L_2 \vdash \sigma \).

The lemma is important since it allows to compose different results obtained by type-checking the initial configuration, without weakening their security guarantees.

D. Compositional Verification of Systems

We first introduce a pre-order on typing environments.

**Definition 13 (Environment Ordering):** Given two typing environments \( \Gamma_1 \) and \( \Gamma_2 \), we say that \( \Gamma_1 \) is no more restrictive than \( \Gamma_2 \), written \( \Gamma_1 <: \Gamma_2 \), iff for all \( r \in R \) we have:

- \( \text{lab}(\Gamma_1(r)) \subseteq \text{lab}(\Gamma_2(r)) \);
- \( \Gamma_1^+(r) \subseteq \Gamma_2^+(r) \);
- \( \Gamma_1^-(r) \subseteq \Gamma_2^-(r) \).

Let \( \Gamma <: \Gamma' \), then any security labelling which can be enforced by \( \Gamma \) can also be enforced by the more restrictive \( \Gamma' \). This is formalized by the next lemma.

**Lemma 7:** If \( \Gamma \vdash_0 \mathcal{L} \) and \( \Gamma <: \Gamma' \), then \( \Gamma' \vdash_0 \mathcal{L} \).

To state the next compositional result, we need a further definition, which is needed to compose different security labellings so as to preserve their security guarantees.

**Definition 14 (Compatibility):** Two security labellings \( L_1 = (\gamma_1, \delta_1) \) and \( L_2 = (\gamma_2, \delta_2) \) are compatible iff for all \( R \in \gamma(R) \) we have \( \delta_1(R) \cup \delta_2(R) = \delta(R) \) for some \( i \in \{1, 2\} \).

**Lemma 8:** Let \( L_1 \) and \( L_2 \) be two compatible security labellings. If \( \Gamma \vdash_0 L_1 \) and \( \Gamma \vdash_0 L_2 \), then \( \Gamma \vdash_0 L_1 \sqcup L_2 \).

Assume then that \( \Gamma_1 \vdash_0 L_1 \) and \( \Gamma_2 \vdash_0 L_2 \). Since \( \Gamma_i <: \Gamma_1 + \Gamma_2 \) for any \( i \in \{1, 2\} \), we have \( \Gamma_1 + \Gamma_2 \vdash_0 L_1 \) and \( \Gamma_1 + \Gamma_2 \vdash_0 L_2 \) by Lemma 7. If \( L_1 \) and \( L_2 \) are compatible, we then conclude \( \Gamma_1 + \Gamma_2 \vdash_0 L_1 \sqcup L_2 \) by Lemma 8.

By combining all the results and the observations presented in this section, we finally get our main theorem.

**Theorem 5 (Compositionality):** Let \( \Gamma_1; L_1 \vdash_0 (P_1, \sigma) \) and \( \Gamma_2; L_2 \vdash_0 (P_2, \sigma) \) with \( \Gamma_1 \vdash_0 L_1 \) and \( \Gamma_2 \vdash_0 L_2 \). If \( \Gamma_2 \vdash_0 P_1 \) and \( \Gamma_1 \vdash_0 P_2 \) and \( L_1, L_2 \) are compatible, then the system \( (P_1 \cup P_2, \sigma) \) is safe with respect to \( L_1 \sqcup L_2 \).

The theorem ensures that the union of two policies preserves the security guarantees enforced by the individual policies, provided that the first policy is typeable in the typing environment used to verify the second policy and vice-versa. The result allows one to verify the security of a large ARBAC policy by decomposing it into smaller sub-policies where the security analysis is much more efficient.

VI. Boosting the Expressiveness

Though already quite expressive, the type system presented in Section III fails at proving the safety of many correct ARBAC systems. In particular, we observe two potential expressiveness problems for the type system:

1) it is crucial for many security proofs, including that of Example 3, that some roles are never revoked. However, many real ARBAC policies allow the revocation of a significant number of roles or even all the roles, which limits the power of reasoning by typing;
2) even though the ARBAC policy ensures a number of useful invariants, the initial configuration may violate some of them, hence Theorem 2 could not be applied to construct a security proof.

While it is certainly possible to devise more sophisticated type disciplines to deal with these problems, doing so would complicate the formal framework and likely make the analysis less efficient and more difficult to understand. Our choice then is keeping the analysis simple, while proposing syntactic transformations of the ARBAC system which preserve the soundness of the analysis, while simplifying the construction of a security proof.

A. Safety by Rewriting and Typing

We motivate our approach with a very practical example. Consider a policy \( P = (CA, CR) \) where a role \( r \) does not occur in the negative preconditions of any can-assign rule: for this policy, the revocation of \( r \) does not enable any new administrative action. Hence, building a security proof for the policy \( P' = (CA, \{ (r_a, r') \in CR \mid r' \neq r \} ) \) is enough to prove the safety of \( P' \), even though \( P \) may not admit a type-based security proof.

By generalizing over the observations above, we presuppose a rewriting relation \( S \leadsto S' \) to support the analysis, with the idea that:

1) no security violation is lost upon rewriting
2) the existence of a type-based security proof is not lost upon rewriting. Hence, proving the safety of \( S' \) is enough to prove the safety of \( S \), but it is no harder. This is formalized by the following two definitions.

**Definition 15 (Sound Rewriting):** The rewriting \( S \leadsto S' \) is sound for the security labelling \( \mathcal{L} \) iff, whenever \( S' \) is safe with respect to \( \mathcal{L} \), also \( S \) is safe.

**Definition 16 (Precise Rewriting):** The rewriting \( S \leadsto S' \) is precise for the security labelling \( \mathcal{L} \) iff, for all \( \Gamma \) such that
\[\Gamma \vdash \mathcal{L} \quad \text{and} \quad \Gamma; \mathcal{L} \vdash \mathcal{S}, \quad \text{there exists} \quad \Gamma' \quad \text{such that} \quad \Gamma' \vdash_{\mathcal{L}} \mathcal{L} \quad \text{and} \quad \Gamma'; \mathcal{L} \vdash \mathcal{S}'.\]

While the soundness of rewriting is clearly a strict requirement to preserve the correctness of the analysis, one may occasionally choose to sacrifice precision, which may be harder to prove. Notice however that, due to the nature of the typing rules, any rewriting which just drops can-assign or can-revoke rules from the original policy is necessarily precise.

B. A Sensible Rewriting

We now discuss one possible rewriting which is a good candidate to support our type-based analysis, since it allows one to eliminate many can-assign rules from the original policy and significantly reduce its size.

We start by defining when a role is irrelevant for security.

Definition 17 (Irrelevant Role): Let \( \mathcal{P} = (CA, CR) \). A role \( r \) is irrelevant for \( \mathcal{P} \) under \( \mathcal{L} = (\gamma, \delta) \) iff both the following conditions hold:

1) \( \forall (r_a, R_p, R_n, r_t) \in CA : r_a \neq r \land r \notin R_p; \)
2) \( \forall R \in 2^{\hat{R}} : \delta(R) = \delta(R \setminus \{r\}). \)

In words, a role \( r \) is irrelevant whenever (1) it cannot be used to assign other roles and it is not needed to acquire new roles, (2) its presence in a set of roles is not important to detect a security violation. Notice that the second condition implies that \( \delta(\{r\}) = \bot \), since \( \delta(\emptyset) = \bot \) by definition.

Let \( \mathcal{S} = (\mathcal{P}, \sigma) \) with \( \mathcal{P} = (CA, CR) \) and \( \sigma = (U, UR) \), we let \( \langle S \rangle^i_{fix} \) be the system obtained by applying the \( i \)-th rule from the following list:

1) if \( (u, r) \in UR \) and \( r \) is irrelevant for \( \mathcal{P} \) under \( \mathcal{L} \), then remove \( (u, r) \);
2) if \( ca = (r_a, R_p, R_n, r_t) \in CA \) and \( r_t \) is irrelevant for \( \mathcal{P} \) under \( \mathcal{L} \), then remove \( ca \);
3) if \( cr = (r_a, r_t) \in CR \) and there does not exist any \( (r'_{a}, R_p, R_n, r'_{t}) \in CA \) with \( r_t \in R_n \), then remove \( cr \).

We let \( \langle S \rangle^i_{fix} \) be the system obtained by repeatedly applying the rules above (in any order) up to a fix point: notice that there must exist one, since any rule application shrinks the size of the system. Moreover, observe that the fix point is unique, since different rule applications commute with each other.

Lemma 9: The rewriting \( S \leadsto \langle S \rangle^i_{fix} \) is both sound and precise for any \( \mathcal{L} \).

VII. IMPLEMENTATION

We implemented TAPA, a static type-checker for ARBAC policies based on our theory. TAPA consists of around 1100 lines of Python code.

A. Overview

Given an ARBAC system \( \mathcal{S} \) encoded in a standard format [11] and a security labelling \( \mathcal{L} \), TAPA first performs the rewriting described in Section VI-B to simplify the security problem and then generates the constraints required by the type inference algorithm in Section IV. The constraints are then solved by the open-source SMT solver CVC4 [36].

If the constraints are solvable, the system \( \mathcal{S} \) is safe with respect to \( \mathcal{L} \) and a corresponding typing environment is returned as an output, otherwise \( \mathcal{S} \) is not typeable and it may be safe or not (since the type system is sound, but not complete). If a can-assign rule is type-checked by rule (CA-Trivial1) or by rule (CA-Trivial2), or if a can-revoke rule is type-checked by a vacuous application of rule (CR-Single), TAPA warns the policy administrator about the presence of a policy rule which can never be triggered.

B. Experiments

We evaluate TAPA against several ARBAC policies from the literature, representing a hospital, a university and a bank with sixteen identical branches [10], [4], [20]. The security analysis considers both privilege escalation (PE) problems and separation of duty (SoD) constraints, including both safe and unsafe examples. For each example, we keep track of its size before and after the rewriting, the verification time required by the type inference algorithm and the output of the type-checker; moreover, we report whether the ill-typed examples are actually unsafe or not, to understand the expressiveness of the analysis. The experimental results collected by running TAPA on an Intel i7-4600U quad-core 2.1 GHz with 6 GB RAM are given in Table IV and commented below.

For the hospital and the university policies the tool was extremely effective, finding security proofs for all the safe examples in less than 2 seconds and failing in less than 1 second on the unsafe instances. The bank policy is much larger and involved, but still TAPA performed quite well in our experiments. From a performance point of view, we observe that most of the examples are solved in less than 10 seconds, though the verification is particularly challenging for examples B4 and B5. However, we are able to deal with these cases by appealing to our compositionality results to dramatically improve the verification time. For instance, B5 checks whether the role combination \( \{Obi, Fai, Sti, Sei\} \) for some \( i \in [1,16] \) can be assigned to the same user, i.e., if there exists one branch of the bank where four given roles can be assigned together. Since the sixteen branches of the bank are identical, the example is a good candidate to apply our compositionality results: example B5c just checks if the role combination \( \{Obi, Fai, Sti, Sei\} \) for some \( i \in [1,16] \) can be assigned to the same user, i.e., if the first branch of the bank admits a security violation. As it turns out, verifying B5 requires 7m45s, while checking B5c just takes 4.2s. By replicating B5c on the other branches, we are able to verify security in 16 x 4.2s = 67.2s, with a 7x speed-up with respect to B5. This performance boost is mostly due to the fact that the rewriting performs poorly on B5, since the number of irrelevant roles is much smaller than in B5c, thus leading to a significantly harder security problem. The improvement in performances is even more apparent for examples B3 and B4, where verification takes more than 20 minutes, but by solving example B4c and using compositionality we are able to verify security in approximately 2m30s.

From the expressiveness point of view, we observe that the tool reported only one false positive, namely example B1, since it was not able to prove by typing that the role combination \( R = \{OBasst1, OBJunior1, OBSenior1, OBClerk1\} \) is never assigned to the same user. The reason is that each of the roles in \( R \) can be assigned together with any other of the roles in
TABLE IV Experimental results

<table>
<thead>
<tr>
<th>Input</th>
<th>Original</th>
<th>Rewritten</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy</td>
<td>#roles</td>
<td>#CA</td>
<td>#CR</td>
</tr>
<tr>
<td>H1</td>
<td>15</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>H2</td>
<td>15</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>H3</td>
<td>15</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>H4</td>
<td>15</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>H5</td>
<td>15</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>U1</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>U2</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>U3</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>U4</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>U5</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>U6</td>
<td>34</td>
<td>374</td>
<td>75</td>
</tr>
<tr>
<td>B1</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B2</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B3</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B4</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B4c</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B5</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
<tr>
<td>B5c</td>
<td>531</td>
<td>4625</td>
<td>516</td>
</tr>
</tbody>
</table>

H1 = SoD {Doctor, Receptionist} U1 = SoD {DepartmentChair, Dean} B1 = SoD {OBAst1, OBJunior1, OBSenior1, OBClerk1}
H2 = SoD {PrimDoctor, Patient} U2 = PE {President, Provost, Dean} B2 = SoD {OBI, FA1}
H3 = SoD {Doctor, Nurse} U3 = SoD {UnderGrad, Grad} B3 = SoD {OBI, FASpecial1}
H4 = PE {PatientWithTPC} U4 = SoD {GACommittee, AOfficer} B4 = SoD {OBI, FASpecial1} \in [1,16]
H5 = H1 + H2 U5 = SoD {President, Dean} B5 = SoD {OBI, FASpecial1, ST1, SE1}
U6 = U1 + U3 B5c = SoD {OBI, FA1, ST1, SE1}

R, even though they cannot all be assigned at the same time, hence the invariants on the individual role assignments are too weak to build a security proof. Improving the expressiveness of the type system to deal with this complex example would be an interesting avenue for future work.

VIII. Conclusion

In this paper, we performed a first investigation on the benefits of types as an effective tool for the static verification of ARBAC policies. Our type system comes with a sound and complete type inference algorithm, and novel compositionality results which make verification scale by design to extremely large policies. We evaluated our theory by implementing TAPA, a static analyser for ARBAC policies, and by testing it on a number of publicly available examples from the literature. Our experimental results confirm both the efficiency and the expressiveness of the analysis we proposed.

As future work, we plan to further improve the expressiveness of the type system and to apply it to the verification of parametric ARBAC policies [37]. In these policies, parametrized roles are built from a set of role templates to limit the scope of access control permissions: for instance, a role template Professor[\cdot] with read permission can be instantiated to the parametrized role Professor[Math] to provide read access only to the resources related to the Math degree. A limited form of parametrized roles can be encoded into ARBAC by creating a different role for each different instantiation of the role templates, but at the cost of significantly increasing the size of the system to analyse. We think that types for role templates look like a promising approach to the scalable verification of parametric ARBAC policies.

Acknowledgements

We would like to thank Morgan Deters and Kshitij Bansal for their guidance in the use of CVC4. This work was supported by the MIUR projects ADAPT and CINA.

References
