MOHAWK: Abstraction-Refinement and Bound-Estimation for Verifying Access Control Policies

KARTHICK JAYARAMAN, Microsoft
MAHESH TRIPUNITARA, University of Waterloo
VIJAY GANESH and MARTIN RINARD, MIT
STEVE CHAPIN, Syracuse University

Verifying that access-control systems maintain desired security properties is recognized as an important problem in security. Enterprise access-control systems have grown to protect tens of thousands of resources, and there is a need for verification to scale commensurately. We present techniques for abstraction-refinement and bound-estimation for bounded model checkers to automatically find errors in Administrative Role-Based Access Control (ARBAC) security policies. ARBAC is the first and most comprehensive administrative scheme for Role-Based Access Control (RBAC) systems. In the abstraction-refinement portion of our approach, we identify and discard roles that are unlikely to be relevant to the verification question (the abstraction step). We then restore such abstracted roles incrementally (the refinement steps). In the bound-estimation portion of our approach, we lower the estimate of the diameter of the reachability graph from the worst-case by recognizing relationships between roles and state-change rules. Our techniques complement one another, and are used with conventional bounded model checking. Our approach is sound and complete: an error is found if and only if it exists. We have implemented our technique in an access-control policy analysis tool called MOHAWK. We show empirically that MOHAWK scales well to realistic policies, and provide a comparison with prior tools.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.4.6 [**Operating System**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Algorithms, Security, Verification

ACM Reference Format:
Jayaraman, K., Tripunitara, M., Ganesh, V., Rinard, M., and Chapin, S. 2013. MOHAWK: Abstraction-refinement and bound-estimation for verifying access control policies. *ACM Trans. Inf. Syst. Secur.* 15, 4, Article 18 (April 2013), 28 pages.
DOI: http://dx.doi.org/10.1145/2445566.2445570

1. INTRODUCTION

We present a technique and a tool for verifying access-control policies. Specifying and managing access-control policies is a problem of critical importance in system security. Researchers have proposed access-control frameworks (e.g., Administrative Role Based Access Control — ARBAC [Sandhu et al. 1999]) that have considerable expressive

A preliminary version of this article appears in *Proceedings of the ACM Conference on Computer and Communications Security (CCS'11)* [Jayaraman et al. 2011].
Authors' addresses: K. Jayaraman, Microsoft, Windows Azure, Microsoft, Redmond, WA; email: karjay@microsoft.com; M. Tripunitara, University of Waterloo, 200 University Ave., W, Waterloo, ON N2L 3G1, Canada; email: tripunit@uwaterloo.ca; V. Ganesh and M. Rinard, Massachusetts Institute of Technology, 77 Massachusetts Ave., Cambridge, MA 02139; email: {vagnesh, rinard}@csail.mit.edu; S. Chapin, Syracuse University, 900 South Crouse Ave., Syracuse, NY 13244; email: chapin@syr.edu.

DOI: http://dx.doi.org/10.1145/2445566.2445570

power, and can be used to specify complex policies. However, we do not have adequate tools for analyzing such complex policies. Without tools for analyzing these policies, administrators cannot determine the correctness of policies. As a consequence, several of these sophisticated frameworks are not deployed in practice.

An access-control policy contains an error if it allows an unauthorized user access to a resource. This is considered an error because a security property that an enterprise wants to (or even is legally required to) maintain, such as separation of privilege [Saltzer and Schroeder 1975], may be violated by the user's access to the resource. In RBAC, for example, if a user is already a member of a sensitive role, we may want to ensure that there exists no reachable state in which he is authorized to another sensitive role.

Administrators require efficient tools for identifying such errors in policies prior to deployment. Access-control policies for large systems feature several sources of complexity that make it difficult to find errors in them. An access-control policy is essentially a state machine that accepts valid requests. Depending on the framework, the policy may comprise states only or comprise both states and state changes. The nature of the states and state changes are sources of complexity (We explain the sources of complexity in Section 4.) We need tools that are effective irrespective of the complexity of the access-control policies.

Automated analysis and verification of access-control policies is both an area of active research [Ferrara et al. 2012; Fisler et al. 2005; Gofman et al. 2009; Harrison et al. 1975; Hu and Ahn 2008; Hu et al. 2008; Hughes and Bultan 2008; Jha et al. 2008; Kolovski et al. 2007; Li and Tripunitara 2006; Li et al. 2005; Sasturkar et al. 2011; Sohr et al. 2008; Stoller et al. 2007; Zhang et al. 2008; Zhao et al. 2005] and practical interest [Aveksa 2012; SailPoint 2012]. Model checking [Clarke et al. 1999] has emerged as a promising, automated approach to the verification problem [Gofman et al. 2009; Jha et al. 2008; Stoller et al. 2007]. In this approach, a model checker takes as input an access-control policy and a security property, and declares whether or not the policy adheres to the input security property. The idea is similar to verifying computer programs; the access-control policy is analogous to a computer program, and the security property is analogous to a program property. Ideally, the model checker checks whether the property always holds for all possible authorizations that the policy allows. However, the model-checking problem for the class of access-control policies that we consider is intractable in general (PSPACE-complete [Jha et al. 2008; Sasturkar et al. 2011]), and scalability for practical verification tools remains a significant issue despite considerable progress (see Section 5).

We present a new abstraction refinement [Clarke et al. 2003] approach for analyzing complex security policies. Abstraction refinement is a paradigm for efficiently verifying a property on a system's abstraction that is iteratively constructed to contain only the relevant parts; the techniques vary depending on the specific system and property [Ball and Rajamani 2002; Clarke et al. 2003; Ganesh and Dill 2007]. The use of a sophisticated abstraction-refinement algorithm makes our technique more efficient than previous approaches for several classes of problem instances. In realistic policies, our algorithm is often able to abstract most of the access-control policy while preserving the presence of any errors. This abstraction enables the underlying bounded model checker to find the error in much less time.

The insight behind the abstraction we use is that the dependency graph of the roles for many real-world access-control policies often contains many tightly knit sets of roles that are loosely-connected to other tightly knit sets of roles. This structure reflects the organization of large institutions (such as corporations, hospitals, and universities) as a conglomeration of loosely coupled tightly knit departments or groups. Abstracting the policy produces a much smaller policy because it eliminates a large

number of loosely coupled roles that do not directly interact with the target set of tightly coupled roles. If some of the loosely coupled roles are indirectly involved in the error, the refinement step will incrementally add them back to the abstraction, enabling the analysis to find the error. Our results show that even when these roles are required to find the error, the analysis can usually find the error after applying a few refinement steps.

Complementarily, we present a new bound-estimation technique for ARBAC policies. In this portion of our approach, we begin with an upper-bound for the diameter of the state-reachability graph within which an error, if it exists, is guaranteed to lie. We then decrease this upper-bound while ensuring that it remains an upper-bound for the diameter. In practice, we expect our bound-estimation technique to significantly decrease the upper-bound, which, in the worst-case can be super-polynomial in the size of the input (assuming $\mathbf{P} \neq \mathbf{PSPACE}$).

We have implemented these techniques in the MOHAWK access-control policy analysis tool [Jayaramam 2012]. MOHAWK accepts as input an access-control policy and a safety query.[1] If MOHAWK finds an error in the input policy, it terminates and produces as output a sequence of actions that cause the error. We show that MOHAWK scales well as the complexity of the input policies increases. The MOHAWK tool is available as open source [Jayaramam 2012].

Contributions

We make the following contributions in this article.

(1) We describe an *abstraction-refinement*-based approach for verifying access-control policies (specifically, ARBAC policies). The resulting technique, implemented on top of a bounded model checker, scales well as the size and complexity of the input policies increase. Our technique effectively tackles the sources of complexity (Section 4) in large, complex, real-world policies. Our technique can also be used for verification in frameworks other than ARBAC as the sources of complexity that we identify in Section 4 are not unique to ARBAC. They exist in other access-control schemes as well, such as administrative scope [Crampton and Loizou 2003] and even the original access matrix scheme due to Harrison et al. [1975].

(2) We describe a method for *bound-estimation*; computing an approximation of the diameter for an ARBAC policy within which any error is guaranteed to lie. We provide this estimate to a bounded model checker. A bounded model checker searches for counter examples for a property ϕ with a bounded length k. If k is set to at least the diameter of the ARBAC policy and the bounded model checker does not find any counterexamples, then no counterexamples exist for the specified property ϕ [Clarke et al. 2005; Kroening 2006]. MOHAWK, when used in the bounded model checking mode, estimates the diameter and uses that as the bound for k.

(3) An implementation of our technique. MOHAWK accepts as input an access-control policy and a safety question, and outputs whether or not it found an error. Following similar techniques from software verification, our technique constructs an approximation (and successive refinements, if necessary) of the input policy, and checks for errors. It terminates when an error has been found or the underlying bounded model checker has reached a predetermined bound.

[1] A safety query is a pair $\langle u, r \rangle$ where u is a user and r is a role – see Section 2.2. Our implementation takes as input a slightly more general query of the form $\langle u, s \rangle$, where s is a set of roles. The instance is deemed to be unsafe if and only if there is a reachable state in which u is a member of all roles in s. We have adopted the $\langle u, s \rangle$ form for queries in our implementation as some prior tools accept such queries. There is a straightforward linear-time reduction from the problem with a query of the form $\langle u, s \rangle$ to the problem with a query of the form $\langle u, r \rangle$. Therefore, in this article, we discuss $\langle u, r \rangle$ queries only.

(4) We provide a detailed experimental comparison of MOHAWK against NuSMV [NuSMV 2012], a well-known model checking and bounded model checking tool, and RBAC-PAT [Gofman et al. 2009; Stoller et al. 2007], a tool specifically designed for analyzing ARBAC policies. In comparison to the existing approaches, MOHAWK scales well with the size and complexity of several classes of input policies.
Our experimental evaluation uses a benchmark that includes a realistic case study for banking (Section 5.1). The case study has helped us understand the aspects of the ARBAC policy language that are likely to be used in creating real-world policies. In particular, we have discovered that the sources of complexity in ARBAC policies that we discuss in Section 4 can arise in realistic settings. Our benchmark is also publicly available along with the tool [Jayaramam 2012].

Organization. The remainder of our article is organized as follows. In Section 2, we discuss access-control models and schemes. In Section 3, we describe the architecture of MOHAWK, and how it performs abstraction-refinement and bound-estimation. In Section 4, we describe the sources of complexity and how MOHAWK deals with them. In Section 5, we present empirical results that demonstrate the efficacy of our approach. We discuss related work in Section 6 and conclude with Section 7.

2. PRELIMINARIES

In this section, we provide basic definitions and concepts relating to access-control policies, in particular the ARBAC framework. We also introduce the verification problem for access-control systems.

An access-control policy is a state-change system, $\langle \gamma, \psi \rangle$, where $\gamma \in \Gamma$ is the start or current state, and $\psi \in \Psi$ is a state-change rule. The pair $\langle \Gamma, \Psi \rangle$ is an access control model or framework. The state, γ, specifies the resources to which a principal has a particular kind of access. For example, γ may specify that the principal Alice is allowed to read a particular file. Several different specifications have been proposed for the syntax for a state. Two well-known ones are the access matrix [Graham and Denning 1972; Harrison et al. 1975] and Role-Based Access Control (RBAC) [Ferraiolo et al. 2003; Sandhu et al. 1996]. In this article, we focus on the latter to make our contributions concrete.

In RBAC, users are not assigned permissions directly, but via a role. Users are assigned to roles, as are permissions, and a user gains those permissions that are assigned to the same roles to which he is assigned. Consequently, given the set U of users, P of permissions and R of roles, a state γ in RBAC is a pair $\langle UA, PA \rangle$ where $UA \subseteq U \times R$ is the user-role assignment relation, and $PA \subseteq P \times R$ is the permission-role assignment relation.

RBAC also allows for roles to be related to one another in a partial order called a role hierarchy. However, as we point out in Section 2.2 under "The role hierarchy," in the context of this article, we can reduce the verification problems of interest to us to those for which the RBAC state has no role hierarchy. Figure 1 contains a example of an RBAC state for a hypothetical company with 7 roles and 2 users, namely Alice and Bob. Alice is assigned to the Admin role. Bob is assigned to the Acct and Audit roles. For the sake of illustration, we have only a limited number of roles in the example. We explain how to interpret the state-change rules in the next section.

2.1. ARBAC

The need for a state-change rule, ψ, arises from a tension between security and scalability in access control systems. Realistic access control systems may comprise tens of thousands of users and resources. Allowing only a few trusted administrators to handle changes to the state (e.g., remove read access from Alice) does not scale. A

	RBAC STATE
Roles	{BudgetCommitee, Finance, Acct, Audit, TechSupport, IT, Admin}
Users	{Alice, Bob}
UA	{⟨Bob, Acct⟩, ⟨Bob, Audit⟩, ⟨Alice, Admin⟩}
	STATE-CHANGE RULES
can_assign	{⟨Admin, Finance, BudgetCommittee⟩, ⟨Admin, Acct ∧ ¬Audit, Finance⟩, ⟨Admin, TRUE, Acct⟩, ⟨Admin, TRUE, Audit⟩ ⟨Admin, TechSupport, IT⟩, ⟨Admin, TRUE, TechSupport⟩}
can_revoke	{⟨Admin, Acct⟩, ⟨Admin, Audit⟩, ⟨Admin, TechSupport⟩}

Fig. 1. State and state-change rules for an ARBAC policy.

state-change rule allows for the delegation of some state changes to users that may not be fully trusted.

ARBAC [Sandhu et al. 1999] and administrative scope [Crampton and Loizou 2003] are examples of such schemes for RBAC. To our knowledge, ARBAC is the first and most comprehensive state-change scheme to have been proposed for RBAC. This is one of the reasons that research on policy verification in RBAC [Gofman et al. 2009; Li and Tripunitara 2006; Stoller et al. 2007], including this article, focuses on ARBAC.

An ARBAC specification comprises three components, *URA*, *PRA*, and *RRA*. *URA* is the administrative component for the user-role assignment relation, *UA*, *PRA* is the administrative component for the permission-role assignment relation, *PA*, and *RRA* is the administrative component for the role hierarchy.

Of these, *URA* is of most practical interest from the standpoint of verification. The reason is that in practice, user-role relationships are the most volatile [Kern 2002]. Permission-role relationships change less frequently, and role-role relationships change rarely. Furthermore, as role-role relationships are particularly sensitive to the security of an organization, we can assume that only trusted administrators are allowed to make such changes.

PRA is syntactically identical to *URA* except that the rules apply to permissions and not users. Consequently, all our results in this paper for *URA* apply to *PRA* as well. We do not consider analysis problems that relate to changes in role-role relationships for the reasons we cited above. In the remainder of this article, when we refer to ARBAC, we mean the *URA* component that is used to manage user-role relationships.

URA. A *URA* specification comprises two relations, *can_assign* and *can_revoke*. The relation *can_assign* is used to specify under what conditions a user may be assigned to a role, and *can_revoke* is used to specify the roles from which users' memberships may be revoked. We call a member of *can_assign* or *can_revoke* a *rule*.

A rule in *can_assign* is of the form $\langle r_a, c, r_t \rangle$, where r_a is an administrative role, c is a precondition and r_t is the target role. An administrative role is a special kind of role associated with users that may administer (make changes to) the RBAC policy. The first component of a *can_assign* rule identifies the particular administrative role whose users may employ that rule as a state change.

A precondition is a propositional logic formula of roles in which the only operators are negations and conjunctions. Figure 1 contains the *can_assign* and *can_revoke* rules

for our example RBAC state. An example of c is Acct $\wedge\neg$Audit in the *can_assign* rule that has Finance as the target role. For an administrator, Alice, to exercise the rule to assign a user *Bob* to Finance, Alice must be a member of Admin, *Bob* must be a member of Acct and must not be a member of Audit.

A *can_revoke* rule is of the form $\langle r_a, r_t \rangle$. The existence of such a rule indicates that users may be revoked from the role r_t by an administrator that is a member of r_a. For example, roles Acct, Audit, and TechSupport can be revoked from a user by the administrator Alice.

We point out that so long as r_a has at least one user, the rule can potentially fire as a state change. If r_a has no users, we remove the corresponding *can_assign* rule from the system as it has no effect on verification. One of the consequences of this relates to what has been called the separate administration restriction. We discuss this in Section 2.2.

We have omitted some other details that are in the original specification for *URA* [Sandhu et al. 1999] because those details are inconsequential to the verification problem we address in this paper. For example, the original specification allows for the target role to be specified as a set or range of roles. We assume that it is a single role, r_t. A rule that has a set or range as its component for the target role can be rewritten as a rule for every role in that set or range. We know that roles in a range do not change, as we assume that changes to roles may be effected only by trusted administrators. Policy verification is not used for such changes.

2.2. The Verification Problem

The verification problem in the context of access-control policies arises because state changes may be effected by users that are not fully trusted, but an organization still wants to ensure that desirable security properties are met. The reason such problems can be challenging is that state-change rules are specified procedurally, but security properties of interest (e.g., Alice should never be able to read a particular file) are declarative [Jones et al. 1976].

A basic verification problem is safety analysis. In the context of RBAC and ARBAC, a basic safety question is: can a user u become a member of a role r? We call such a situation (in which the safety question is true), an *error*.

There are two reasons that the basic safety question such as the one from above has received considerable attention in the literature [Jha et al. 2008; Li and Tripunitara 2006; Sasturkar et al. 2011; Stoller et al. 2007]. One is that it is natural in its own right. The reason for asking such a question is that u should not be authorized to r. If the analysis reveals that he may be, by some sequence of state changes, then we know that there is a problem with the security policy. Another reason is that several other questions of interest, such as those related to separation of privilege, can be reduced efficiently to the basic safety question. This observation has been made before [Jha et al. 2008; Stoller et al. 2007]. Consequently, even though MOHAWK's input interface supports problem instances that ask whether a user can be a member of a set of roles, in this article, we assume that an instance pertains to a user u and a role r only.

An instance of a verification problem, in the context of this article, specifies an ARBAC access control system, a user, and a role. It is of the form $\langle \gamma, \psi, u, r \rangle$. We ask whether u may become a member of r given the initial state $\gamma = \langle U, R, UA \rangle$, and state-change rule $\psi = \langle can\_assign, can\_revoke \rangle$.

The Separate Administration Restriction. In the context of the verification problem, the separate administration restriction excludes administrative roles from the verification problem instance, that is, administrative roles are not administered by the same rules as "regular" roles. We adopt the separate administration restriction in this

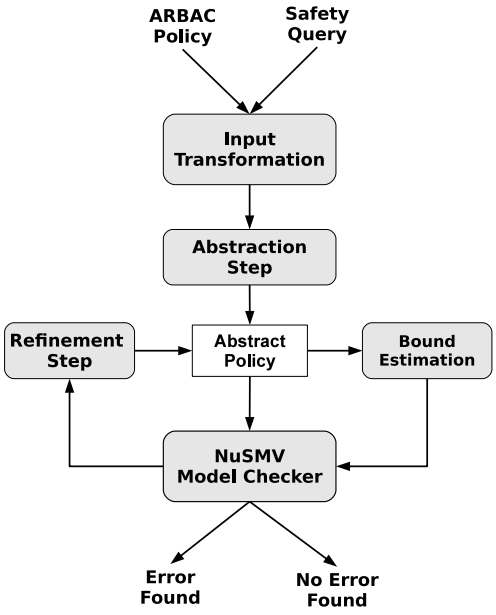

Fig. 2. MOHAWK architecture.

paper. Some previous work [Stoller et al. 2007] has considered subcases of when this restriction is lifted. We do not consider those subcases in this article. We adopt the restriction because the original specification on ARBAC [Sandhu et al. 1999] adopts this restriction. Moreover, the separate administration restriction does not affect the complexity of the verification problem [Sasturkar et al. 2011].

The Role Hierarchy. We assume that an RBAC state has no role hierarchy. The reason is that there is a straightforward, efficient reduction that has been presented in prior work from an verification instance that has a role hierarchy to a verification instance with no role hierarchy [Sasturkar et al. 2006, 2011].

3. ARCHITECTURE OF MOHAWK

In this section, we describe MOHAWK's architecture (Sections 3.1–3.2), and illustrate our approach using an example (Section 3.1.5). Figure 2 illustrates the architecture of MOHAWK. MOHAWK accepts an ARBAC Policy $\langle U, R, UA, can\_assign, can\_revoke\rangle$ and a safety query $\langle u, r\rangle$ as input. MOHAWK reports an error if it finds one. Otherwise, MOHAWK terminates and reports that it could not find any errors. In the following two sections, we discuss the various components of MOHAWK that are shown in Figure 2.

3.1. Abstraction Refinement in Mohawk

The general idea in an abstraction-refinement [Clarke et al. 2003] technique is to first abstract (over or under approximate) a system and check if a desired property holds in the abstraction, and iteratively refine as necessary. If the abstraction is an overapproximation, that is, the abstraction has more behaviors compared to its original, then the refinements restrict the abstraction's behavior on each iteration, so that it has fewer behaviors, until in the limit the abstraction equals the original. If the abstraction is an under approximation, that is, the abstraction has fewer behaviors compared to its original, then the refinements relax the abstraction's behavior on each iteration, so

that it has more behaviors, until in the limit the abstraction equals the original. Both techniques keep the errors one sided. In the case of overapproximation, correctness of the abstraction corresponds with the original. In the case of underapproximation, an error detected in the abstraction corresponds with the original.

MOHAWK uses an under approximation strategy [Clarke et al. 2003]. Therefore, any error detected in an abstract policy exists in the original policy. In the following, we refer to the role in the safety query as the query role. MOHAWK works on the input as follows.

—*Input Transformation* (Section 3.1.1). MOHAWK transforms the policy and safety query into an intermediate representation (IR). The IR maintains a priority queue of roles based on how they are related to the query role, and uses this stratification to incrementally add roles, *can_assign*, and *can_revoke* rules in the refinement steps as necessary.
—*Abstraction Step* (Section 3.1.2). MOHAWK performs an initial abstraction step to produce an abstract policy.
—*Verification* (Section 3.1.3). In this step, MOHAWK invokes the underlying NuSMV model checker on the finite-state machine representation of the current abstract policy. MOHAWK can be configured to choose either the symbolic model checker or the bounded model checker. When used in the bounded model-checking mode, MOHAWK additionally calculates an overapproximation of diameter of the policy and uses that as the bound.
If a counter-example is produced by NuSMV, MOHAWK terminates and reports the error found. For MOHAWK, the counter-example reported corresponds to an error in the policy and is essentially a sequence of actions that enable the unauthorized user referred in the safety query to reach the query role.
—*Refinement Step* (Section 3.1.4). If no counter example is found in the previous step, MOHAWK refines the abstract policy. If no further refinements are possible, MOHAWK terminates and reports that it could not find any errors. MOHAWK may execute the verify-refine loop multiple times, until either MOHAWK identifies an error or no further refinements are possible.

Configurability of Abstraction-Refinement. MOHAWK's abstraction-refinement strategy is configurable. In abstraction-refinement techniques, there is an interplay between three factors, namely aggressiveness of the abstraction step, verification efficiency, and number of refinement steps. Aggressive abstraction-refinement makes the verification efficient at the cost of increasing the number of refinements. A less aggressive abstraction-refinement may reduce the number of refinements at the cost of making the verification harder. A key aspect of our approach is that, MOHAWK enables the user to control the aggressiveness of the abstraction and refinement steps. A configurable parameter a determines the number of queues of roles from the priority queues that are added to the abstract policy at each refinement step. The default value for a is one (the most aggressive setting for a).

3.1.1. Input Transformation. Figure 3 illustrates how a policy is specified in MOHAWK's input language. The "Roles", "Users", "UA", "CA", and "CR" keywords identify the lists of roles, users, user-role assignments, *can_assign* rules, and *can_revoke* rules respectively. The "ADMIN" key word identifies the list of admin users. In the example, Alice is the admin user and is assigned to Admin, which is the administrative role assumed in all the *can_assign* and *can_revoke* rules. The SPEC keyword identifies the safety query. In the example, the safety query is asking whether user Bob can be assigned to BudgetCommittee. In the intended policy, Bob cannot be assigned to BudgetCommittee. However, he can be assigned to the role in the policy specified in Figure 3 because

```
Roles BudgetCommitee Finance Acct Audit
TechSupport IT Admin;

Users Alice Bob;

UA <Alice, Admin> <Bob, Acct> <Bob, Audit>;

CR <Admin, Acct> <Admin, Audit>
<Admin, TechSupport>;

CA <Admin, Finance, BudgetCommittee>
<Admin, Acct&Audit, Finance> <Admin, TRUE, Acct>
<Admin, TRUE, Audit> <Admin, TechSupport, IT>
<Admin, True, TechSupport>;

ADMIN Alice;

SPEC Bob BudgetCommittee;
```

Fig. 3. An ARBAC policy in the MOHAWK's input language.

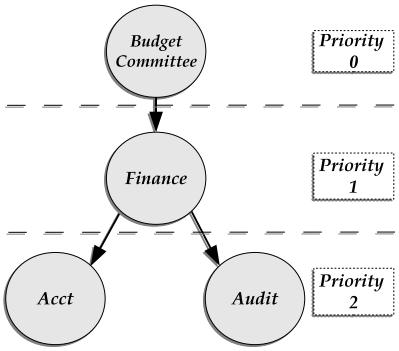

Fig. 4. Related-by-assignment (RBA) relationship between roles with respect to BudgetCommittee.

of the error we introduced in the *can_assign* rule. In Section 3.1.5, we show how MOHAWK identifies the error.

MOHAWK transforms the policy in the input into an intermediate representation, which enables efficient querying of the policy to facilitate the abstraction and refinement steps.

Related-by-Assignment. A role r_1 is said to be *Related-by-Assignment* to a role r_2, if $r_2 = r_1$ or r_2 appears in the precondition of at least one of the *can_assign* rules that have r_1 as their target role. *Related-by-Assignment* does not distinguish between positive and negative preconditions. The *Related-by-Assignment* relationship describes whether a user's membership to one role can affect the membership to another role. The *Related-by-Assignment* relationship between roles can be represented using a tree as shown in Figure 4, in which all nodes correspond to roles and a role r_2 appears as a child node of role r_1, if and only if $r_2 \neq r1$ and r_2 is *Related-by-Assignment* to r_1.

MOHAWK identifies all the roles *Related-by-Assignment* to the query role by performing a breadth-first search of the associated *can_assign* rules. The algorithm first assigns the highest priority to the query role and adds it to a work queue. While the work queue is not empty, the algorithm picks the next role in the work queue, and considers the *can_assign* rules that have the role being analyzed as their target role. All the roles in the preconditions in the *can_assign* rules are added to the work queue, and also added to the priority queue at the next lower priority compared to the role being analyzed. At the end of the analysis, we have a priority queue, in which all the roles *Related-by-Assignment* to the query role are inserted in the queues based on their priorities. Roles that directly affect the membership to the query role have the highest priority, while roles affecting the membership to roles that are *Related-by-Assignment* to the query role have a lesser priority.

3.1.2. Abstraction Step. In the initial abstraction step, MOHAWK constructs an abstract policy $\langle U', R', UA', can\_assign', can\_revoke' \rangle$, where

— U' contains the user in the safety query and admin users;
— R' contains the administrative roles, and roles from the first a queues in the priority queue;
— $UA' = \{(u,r) \mid (u,r) \in UA \wedge u \in U' \wedge r \in R'\}$;

— can_assign' contains only *can_assign* rules in the input policy whose precondition roles and target roles are members of R';
— can_revoke' contains only *can_revoke* rules from the input policy whose target roles are members of R'.

3.1.3. Verification. In each verification step, MOHAWK translates the current abstract policy to a finite-state-machine specification in the SMV language and the safety query to a LTL (Linear Temporal Logic) formula. If MOHAWK is used in the bounded model checking mode, then MOHAWK also calculates an overapproximation of the diameter for the abstract policy and uses that as the bound. Following that, MOHAWK feeds these as input to the underlying model checker. If the model checker identifies a state in which the user is assigned to the role, then it provides a counterexample. The counterexample corresponds to a sequence of assignment and revocation actions from initial authorization state that will result in the user being assigned to the role. On identifying a counterexample, MOHAWK reports the error and terminates. Otherwise, MOHAWK refines the abstract policy in the refinement step.

Any counterexample that MOHAWK identifies in an abstract policy also exists in the original policy. Recall that MOHAWK uses an under approximation strategy. Therefore, each abstract policy permits only a subset of the administrative action sequences accepted in the full policy; each administrative action in the counterexample corresponds to either a *can_assign* or *can_revoke* rule that exists in both the abstract and original policies. Therefore, the counterexample is true in the original policy also.

3.1.4. Refinement Step. An abstract policy verified in the previous step is refined as follows. (We use "$\leftarrow$" to represent instantiation.)

— $R' \leftarrow R' \cup R''$, where R'' is the set of roles from the next a queues from the priority queue.
— $UA' \leftarrow UA' \cup UA''$, where UA'' is the user's membership for the roles in R'', if there are any.
— $can\_assign' \leftarrow can\_assign' \cup can\_assign''$, where can_assign'' is the additional set of *can_assign* rules from the input policy whose preconditions and target roles are members of the updated R'.
— $can\_revoke' \leftarrow can\_revoke' \cup can\_revoke''$, where can_revoke'' is the additional set of *can_revoke* rules from the input policy whose target roles are members of R''.

If no additional refinements are possible, MOHAWK reports that no error was found.

3.1.5. Example. To illustrate MOHAWK's operations, we introduce an error in the policy of our running example in Figure 1. In the *can_assign* rule with target role Finance, we change c from Acct$\wedge\neg$Audit to Acct$\wedge$Audit. The intent of the original policy is to assign the role Finance only to users who are in the Acct role and not in the Audit role. The error we introduced in the example by changing the *can_assign* rule will enable users who are in both Acct and Audit roles to be assigned to Finance. Figure 3 contains the erroneous policy in MOHAWK's input language.

Table I contains the abstraction-refinement steps for the example policy in Figure 3. Figure 4 contains the tree for the roles *Related-by-Assignment* with respect to the BudgetCommittee, which is the query role. In the priority queue, BudgetCommittee has priority 0, Finance has priority 1, and finally Acct and Audit have priority 2 (lower numbers indicate better priorities).

In the abstraction step, MOHAWK adds the users Alice and Bob, and roles BudgetCommittee and Admin. Bob is the user in the query, and Alice is the admin user. BudgetCommittee is the role from the queue with priority 0 and Admin is the admin role. The UA membership, (Alice, Admin), is added to the abstract policy. No *can_assign* or

Table I. Illustrating Abstraction-Refinement Steps for the Running Example in Figure 3

Steps	Users	Roles	UA	*can_assign*	*can_revoke*	Bound
Abstraction step	Alice, Bob	BudgetCommittee, Admin	(Alice, Admin)			2
Refinement 1	Alice, Bob	BudgetCommittee, Admin, Finance	(Alice, Admin)	⟨Admin, Finance, BudgetCommittee, ⟩		3
Refinement 2	Alice, Bob	BudgetCommittee,	(Alice, Admin)	⟨Admin, Finance, BudgetCommittee⟩	⟨Admin, Acct⟩	5
		Admin, Finance,	(Bob, Acct)	⟨Admin, Acct∧Audit, Finance⟩	⟨Admin, Audit⟩	**Error found!**
		Acct, Audit	(Bob, Audit)	⟨Admin, TRUE, Acct⟩		
				⟨Admin, TRUE, Audit⟩		

can_revoke rules are added because all of them require roles not added to the abstract policy. NuSMV does not identify a counterexample for the abstract policy. Therefore, MOHAWK refines the policy.

In the first refinement step, MOHAWK adds Finance from the queue with priority 1. There are no changes to the users, UA, and *can_revoke*. The *can_assign* rule ⟨Admin, Finance, BudgetCommittee⟩ is added to the abstract policy. NuSMV still does not identify a counterexample. Therefore, MOHAWK further refines the abstract policy.

In the second refinement step, MOHAWK adds roles Audit and Acct, 2 UA memberships for Bob, 3 *can_assign* rules, and 2 *can_revoke* rules. Bob's membership to roles Acct and Audit are added to the abstract policy. The three additional *can_assign* rules added are ⟨Admin, Acct∧Audit, Finance⟩, ⟨Admin, TRUE, Audit⟩, and ⟨Admin, TRUE, Acct⟩. The additional *can_revoke* rules added are ⟨Admin, Acct⟩ and ⟨Admin, Audit⟩. NuSMV identifies a counterexample that has the following sequence of administrative actions.

(1) *Alice Assigns Bob to Finance*. This action is allowed because of the *can_assign* rule ⟨Admin, Acct∧Audit, Finance⟩.
(2) *Alice Assigns Bob to BudgetCommittee*. This action is allowed because of the *can_assign* rule ⟨Admin, Finance, BudgetCommittee⟩.

As a result of seeing this counterexample, the administrator can fix the erroneous *can_assign* rule to enforce the correct policy.

As we have illustrated in the example, the abstract policy verified in each step is more constrained compared to the original policy. For example, the initial abstract policy does not allow any assignment and revocation actions. The subsequent two refinement steps add additional roles, *can_assign*, and *can_revoke* rules from the original policy. In effect, the abstraction step aggressively constrains the policy and the subsequent refinement steps relax the constraints to make the policy more precise compared to the earlier step, illustrating our under-approximation strategy.

3.1.6. Correctness of Abstraction-Refinement. The correctness of our approach to abstraction-refinement in MOHAWK is based on the following property.

LEMMA 3.1. *Let* $\langle U, R, UA, can\_assign, can\_revoke\rangle$ *be an input instance to* MOHAWK. *The instance is unsafe for a particular* $\langle u, r\rangle$ *if and only if there exist* $R' \subseteq R$, $UA' \subseteq UA$, $can\_assign' \subseteq can\_assign$ *and* $can\_revoke' \subseteq can\_revoke$ *such that the instance* $\langle U, R', UA', can\_assign', can\_revoke'\rangle$ *is unsafe for* $\langle u, r\rangle$.

The intuition behind the lemma is that adding roles, user-role assignments, *can_assign* or *can_revoke* rules can only add paths to the state-change graph; paths cannot be removed by adding entries to those sets. Consequently, if an intermediate

instance generated by our abstraction-refinement is unsafe, then so is the input instance. Similarly, no intermediate instance generated by our abstraction-refinement is unsafe when the full instance is safe. Therefore, if the underlying model checker is correct, then so is model checking augmented with our approach to abstraction-refinement.

3.2. Bound Estimation in Mohawk

The diameter of a state-change system is the longest shortest path between any two reachable states. Let S be the set of all states and let $s_0 \in S$ be the start-state. Let δ_s be the shortest distance (number of state-changes) from s_0 to $s \in S$. If s is not reachable from s_0, then $\delta_s = 0$. Then,

$$diameter = \max_{s \in S} \{\delta_s\}.$$

A consequence of estimating the diameter is that we can provide it as input to a model checker as an upper-bound on the number of state-changes it has to consider. Given a state-change system M, property ϕ, and a bound k, a bounded model checker verifies ϕ by constructing a Boolean formula that is satisfiable if and only if M contains a state, in which $\neg\phi$ is true, and is reachable within k transitions [Clarke et al. 2001]. A bounded model checker verifies the satisfiability of this formula by using a SAT solver.

In model-checking parlance, the state in which $\neg\phi$ is true is called the error state, and a sequence of state transitions from the initial state to an error state of the input finite-state machine is called a counterexample. If no counterexample of length k is found, then the bounded model checker increases k. The bounded model checker continues this process until either a counterexample is found, or it exhausts its resources. If k is set to the diameter of M and no counterexample is found, then it implies that no such counterexample exists.

THEOREM 3.2. *The diameter of an ARBAC safety instance in which the set of roles is R is $\leq 2^{|R|-1} + 1$.*

The intuition behind this theorem is the following. Let R be the set of roles in the instance, and $\langle u, r \rangle$ be the safety query. Then, we need at most one state-change that pertains to r; one in which we assign u to r. This accounts for the "+1" in the upper-bound for d in Theorem 3.2. For the remainder of the roles, consider the role assignments of u as a bit-vector; that is, each component of the vector is a bit that represents whether u is a member of a particular role. Given any simple path in the state-change graph, each state corresponds to a different value of this bit-vector. Therefore, the longest possible simple path has $2^{|R|-1}$ state-changes.

Whether the upper-bound in Theorem 3.2 is tight is related to foundational open questions in computing. The upper-bound accounts for the possibility that **PSPACE** = **EXPTIME**. If **PSPACE** $\neq$ **EXPTIME**, an open question but widely believed to be true, then the bound is not tight. We can, however, assert that there is a tight upper-bound for d that is superpolynomial in the size of the ARBAC instance if **P** $\neq$ **PSPACE**, another open question that is widely believed to be true.

In practice, the diameter tends to be much smaller than what is suggested by Theorem 3.2. This is because, the *can_assign* and *can_revoke* rules in an ARBAC policy significantly constrain the number of reachable states and the paths between them. For example, let us consider the ARBAC policy in Figure 1. The policy contains 7 roles, and so the upper-bound for the diameter from Theorem 3.2 is 65. However, the actual diameter is 4. In this case, the path between the state wherein a user is assigned to the role "Audit" and the state wherein a user is assigned to roles "BudgetCommittee", "Finance", and "Acct" is the longest amongst all shortest paths. In this path, the

administrator should first revoke the user from "Audit", then assign the user to the roles "Acct", "Finance", and "BudgetCommittee" in order.

In the following, we describe several "tightenings" for calculating a tighter upper-bound for the diameter of the ARBAC policy based on the nature of the *can_revoke* and *can_assign* rule. Tightening 1 is applied first; then the order of the other tightenings does not matter. MOHAWK uses these tightenings for computing the diameter when used in bounded model-checking mode. Our empirical results (see Section 5) demonstrate that these are highly effective. Our general strategy is to trade-off additive terms for exponential terms in the upper-bound in Theorem 3.2. We maintain the invariant that the tightened instance is safe if and only if the original instance is safe. We first discuss static pruning of the ARBAC instance, and then Tightening $1, \ldots, 5$. For each Tightening, we compute a set of roles. We then assert that the size of the union of those sets of roles can be removed from the exponent in the upper-bound in Theorem 3.2. We assert also that every tightening can be carried out efficiently; in time polynomial in the size of the ARBAC instance.

3.2.1. Static Pruning. Jha et al. [2008] discuss two kinds of techniques for pruning the size of an ARBAC safety instance. Both sets of techniques can be called static in that the pruning is done before we input an instance to the model checker. Two kinds of such static pruning are discussed, forward and backward. In forward pruning, we remove *can_assign* rules that cannot possibly be executed. In backward pruning, we remove roles that are irrelevant to deciding the safety query. We remove also *can_assign* and *can_revoke* rules in which any of those roles appears as the target. We refer the reader to Jha et al. [2008] for a description of forward and backward pruning. We point out, however, that while forward and backward pruning are sound, they are not complete. That is, there exist ARBAC instances for which the two prunings do not necessarily remove all possible irrelevant rules/roles.

We have adopted and applied both forward and backward pruning to all the policies we consider. In addition, we adopt and apply the following pruning.

> Let the safety query be $\langle u, r\rangle$. If r appears as a positive precondition in a *can_assign* rule, we discard that rule. If r appears as a negative precondition in a *can_assign* rule, we remove that part (i.e., $\neg r$) from every such precondition.

The rationale for this additional pruning is the following. A rule in which r appears as a positive precondition can fire only if u is already a member of r. If that is the case, we know that the instance is unsafe and need not explore any more reachable states. As for a precondition that contains $\neg r$, consider any simple path in the state-change graph that either does not contain an unsafe state (in which u is a member of r) or terminates in an unsafe state. If the path has at least one state-change, then we know that in all but the final unsafe state, $\neg r$ is true.

The only exception is a path that has no state-changes, that is, the case that u is a member of r in the start-state. We can easily check for this case and eliminate it before input to the model checker. A consequence of static pruning is expressed in the following lemma.

LEMMA 3.3. *Let the safety query be $\langle u, r\rangle$ and the set of roles in the safety instance be R. Then every role in $R - \{r\}$ appears in at least one precondition in can_assign.*

The intuition behind this lemma is the following. We have three possibilities for any role $r' \in R - \{r\}$ for which the assertion is not true. These possibilities are not mutually exclusive. (a) r' appears in the RBAC policy, (b) r' appears as a target role in a *can_assign* rule, and, (c) r' appears as a target role in a *can_revoke* rule. In any

1 $A \leftarrow \{a \in \mathit{can\_assign} \mid \mathit{target\text{-}role}[a] = r\}$
2 **foreach** $a \in A$ **do**
3 $S_a \leftarrow$ smallest set that satisfies $S_a = \{a\} \cup \{a' \in \mathit{can\_assign} \mid \exists\, a'' \in S_a$ with $\mathit{target\text{-}role}[a'] \in \mathit{positive\text{-}precond\text{-}roles}[a'']\}$
4 $R_a \leftarrow \bigcup_{s \in S_a} (\mathit{precond\text{-}roles}[s] \cup \{\mathit{target\text{-}role}[s]\})$
5 $m \leftarrow \max_{a \in A} |R_a|$
6 $R_1 \leftarrow$ some R_a such that $|R_a| = m$

Fig. 5. The Algorithm for Tightening 1. We assume that the role in the safery query is r. We use the following auxiliary routines for $a \in \mathit{can\_assign}$. (1) $\mathit{target\text{-}role}[a]$ returns the target role of a, (2) $\mathit{precond\text{-}roles}[a]$ returns the set of (positive and negative) precondition roles of a, and, (3) $\mathit{positive\text{-}precond\text{-}roles}[a]$ returns the set of positive precondition roles of a.

combination of these cases, r' and any *can_assign* and *can_revoke* rules with it as target would be removed by backward pruning [Jha et al. 2008].

3.2.2. Tightening 1. In Tightening 1, we identify a set of roles R_1 that is the output of the algorithm in Figure 5. In the algorithm, in Line 1, we first identify a set A of all *can_assign* rules that can cause u to be assigned to the role r, where the safety query is $\langle u, r\rangle$. For each such rule a, we identify a set of rules S_a, and a set of roles R_a that appear in the rules of S_a. The intuition is that at most one amongst the rules of A needs to fire. For $a \in A$, we can identify the roles that are needed to cause it to fire, which is what R_a is. We point out that the algorithm runs in time polynomial in the size of the ARBAC instance.

LEMMA 3.4. *The diameter* $\leq 2^{|R_1|-1} + 1$.

PROOF. For any A that the algorithm of Figure 5 computes in Line 1, at most one $a \in A$ fires in a path to an unsafe state. The only state-changes that matter in enabling a to fire are those in R_a, and $|R_1| \geq |R_a|$. The "-1" (in the exponent) and the "$+1$" terms appear because $r \in R_a$ for every a, and there is at most one state-change that involves r — the assignment of u to it. □

3.2.3. Tightening 2. We identify the set of roles R_2 that appear negated only in a precondition of *can_assign*. That is, $r \in R_2$ if and only if r appears in at least one precondition in *can_assign* as $\neg r$, and appears in no precondition as r. We can identify R_2 in time linear in the number of *can_assign* rules, and therefore the ARBAC instance.

LEMMA 3.5. *The diameter* $\leq 2^{|R_1 - R_2|-1} + |R_2| + 1$.

PROOF. $r \in R_2$ needs to be revoked at most once in any state-change path that does not contain an unsafe state or terminates in an unsafe state. □

3.2.4. Tightening 3. We identify the set of roles R_3 that appear non-negated only in preconditions of *can_assign*. That is, $r \in R_3$ if and only if r appears in at least one precondition in *can_assign*, and $\neg r$ appears in no precondition. We can identify R_3 in time linear in the number of *can_assign* rules, and therefore the ARBAC instance.

LEMMA 3.6. *The diameter* $\leq 2^{|R_1 - (R_2 \cup R_3)|-1} + |R_2 \cup R_3| + 1$.

PROOF. u needs to be assigned to $r \in R_2$ at most once in any state-change path that does not contain an unsafe state or terminates in an unsafe state. □

3.2.5. Tightening 4. We identify the set of roles R_4, none of which appears as the target role in any rule in *can_revoke*. As with R_2 and R_3 above, at most one state-change

1 $Pos \leftarrow$ smallest set that satisfies $Pos = \{r\} \cup \bigcup_{r' \in Pos} positive\text{-}precond\text{-}roles[r']$
2 **foreach** $r' \in Pos$ **do**
3 $\quad R_{neg}(r') \leftarrow \{target\text{-}role[a] \mid a \in can\_assign \wedge r' \in negative\text{-}precond\text{-}roles[a]\}$
4 $R_5 \leftarrow Pos - \{r' \mid R_{neg}(r') \cap Pos \neq \emptyset\}$

Fig. 6. The Algorithm for Tightening 5. After executing it, *Pos* contains those roles on which the role in the safety query, r, depends positively only. We use the following auxiliary routines for $a \in can\_assign$. (1) $target\text{-}role[a]$ returns the target role of a, (2) $negative\text{-}precond\text{-}roles[a]$ returns the set of negative precondition roles of a, and, (3) $positive\text{-}precond\text{-}roles[a]$ returns the set of positive precondition roles of a.

involving each such role occurs in any simple path to an unsafe state — the assignment of u to that role.

LEMMA 3.7. *The diameter* $\leq 2^{|R_1 - (R_2 \cup R_3 \cup R_4)| - 1} + |R_2 \cup R_3 \cup R_4| + 1$.

3.2.6. Tightening 5. We identify a set of roles R_5, which we characterize as the set of roles on which the role r in the safety query depends positively only. Note that static pruning [Jha et al. 2008] identifies a set of roles on which r depends positively. Our set R_5 is different; the extra qualifier "only" characterizes the difference. Indeed, R_5 is a subset of the set identified in static pruning.

LEMMA 3.8. *The diameter* $\leq 2^{|R_1 - (R_2 \cup R_3 \cup R_4 \cup R_5)| - 1} + |R_2 \cup R_3 \cup R_4 \cup R_5| + 1$.

PROOF. The intuition is that as the role in the safety query r depends positively only on every $r' \in R_5$, we do not need to revoke u from r'. Therefore, in any simple state-change path to an unsafe state, we have at most one state-change that involves r' — the assignment of u to it.

To show that the algorithm in Figure 6 is sound, we observe the following. In Line 1, we compute as *Pos*, the set of roles on which r depends positively. That is, we may have to assign u to $r' \in Pos$ computed in Line 1 so we may ultimately assign him to r. We may then need to remove some roles from *Pos* to meet the "only" qualification. For this, in Lines 2–3, we identify as $R_{neg}(r')$, those roles that depend negatively on r', for every $r' \in Pos$. That is, to assign u to any of those roles, we may need to revoke u from r'. Finally, in Line 4, we remove r' from *Pos* if any of the roles we may need to assign depends negatively on r' as determined by its membership in $R_{neg}(r')$. □

Given some $r'' \in R_{neg}(r')$, one may wonder why we do not further consider all r''' that depend on r'' positively. That is, a role r''' that needs u to be assigned to $r'' \in R_{neg}(r')$ for u to be assigned to r'''. The reason is that if $r''' \in Pos$ computed in Line 1, then $r'' \in Pos$ computed in Line 1, and r' will be removed from *Pos* in Line 4 as desired.

3.3. Correctness of MOHAWK

We now assert the correctness of MOHAWK.

THEOREM 3.9. MOHAWK *is sound and complete.*

Soundness means that if MOHAWK returns a "safe" or "unsafe" verdict on an input instance, then it is the correct verdict for that instance. Completeness means that MOHAWK returns some verdict on every input instance. A premise behind the theorem is that model checking is correct. The theorem is a direct consequence of Lemmas 3.1 and 3.8. Those two lemmas capture the correctness of the two techniques we employ in MOHAWK.

It may seem somewhat counterintuitive that MOHAWK achieves soundness, completeness and efficiency simultaneously. MOHAWK performs well for certain classes of inputs on which prior approaches do not. However, there are classes of inputs for which MOHAWK may perform worse. The reason is the overhead that abstraction-refinement

and bound-estimation add. We refer the reader to the discussions under "Where MOHAWK performs better" and "Where other tools may perform better" in Section 5. Our main observation regarding efficiency is that there exist classes of realistic instances for which MOHAWK outperforms prior approaches.

3.4. Implementation

We have implemented MOHAWK using Java. In addition to the abstraction-refinement approach, we have implemented several supporting tools for MOHAWK. We have a tool for automatically converting a policy in the MOHAWK language to NuSMV specification. Also, we have implemented a tool for creating complex ARBAC policies. We have incorporated prior static pruning techniques [Jha et al. 2008; Stoller et al. 2007], and our bound estimation techniques.

4. SOURCES OF COMPLEXITY

In this section, we describe the aspects that make verification in the context of access control systems a difficult problem. We call these "sources of complexity." For each source of complexity, we give some intuition as to why it is a source of complexity. Previous work [Jha et al. 2008; Sasturkar et al. 2011; Stoller et al. 2007] on the verification problem alludes to some of these sources of complexity. Finally, we explain how MOHAWK's abstraction refinement and bound estimation strategies deal with these sources of complexity.

4.1. The Sources

Three aspects of access-control systems can bring complexity to the verification: (1) the syntax for the state, (2) the state-change rules, and (3) the verification question of interest. In the context of (1), the size of the state is a source of complexity. In our case, this size is quantified by the number of roles in the RBAC component of the ARBAC policy. The rationale is that as the number of roles in the RBAC component increases, a verifier has to maintain a larger vector of user-role assignments, and also deal with more possible combinations of such assignments in considering state-changes.

The potential source (3) is not a source of complexity in our context. We study the basic question of safety. Given a state, checking whether the question is true or false is equivalent to an access-check. It has been observed in previous work [Jha et al. 2008] that more complex questions can be reduced to this rather basic notion of safety. Consequently, it appears that even more complex questions will not make the verification problem any more difficult.

The other sources of complexity are in the state-change rules. The component within the state-change rules that is relevant is the precondition. The specific aspects of preconditions that are complexity sources are (1) disjunctions — these are introduced by multiple *can_assign* rules with the same target role, (2) irrevocable roles — these are roles for which there is no *can_revoke* rule with any such role as the target, (3) mixed roles — these are roles that appear with and without negation in *can_assign* rules, and, (4) positive precondition roles — these are roles that appear without negation in *can_assign* rules.

Disjunctions. Given a safety instance, $\langle \gamma, \psi, u, r \rangle$ (see Section 2.2) that contains disjunctions, we observe that determining whether the answer is true or not can be equivalent to determining the satisfiability of a boolean expression in Conjunctive Normal Form (CNF). This problem is known to be NP-complete.

Consider the following example. We have as target roles in *can_assign*, $r_1, \ldots, r_n$. The rule that corresponds to r_i in *can_assign* is $\langle r_a, c_i \wedge r_{i-1}, r_i \rangle$ where c_i is a

disjunction of roles or their negations,[2] and contains no roles from among $r_1, \ldots, r_n$. The only *can_assign* rule with r as the target role is $\langle r_a, r_n, r\rangle$ and u is assigned to r_0 in the start state.

In our example, the verification instance $\langle \gamma, \psi, u, r\rangle$ is true if and only if the Boolean expression $c_1 \wedge \cdots \wedge c_n$ is satisfiable via the firing of *can_assign* and *can_revoke* rules. Indeed, this construction that we use as an example is similar to an NP-hardness reduction in previous work [Jha et al. 2008].

Irrevocable Roles. A role $\widehat{r}$ is *irrevocable* if it is not the target role in any *can_revoke* rule. Once u is assigned to $\widehat{r}$, u's membership in $\widehat{r}$ cannot be revoked. Consider the case that an irrevocable role $\widehat{r}$ appears as a negated role in some *can_assign* rules. The challenge for a "forward-search" algorithm that decides the verification question $\langle u, r\rangle$ is that it is not obvious when u should be assigned to $\widehat{r}$.

In a path in the state-transition graph, if u is assigned to $\widehat{r}$ quite close to the start state, then it is possible that that action causes u to never be authorized to r on that path. Given a set of roles I, all of which appear negated in preconditions of *can_assign* rules and are irrevocable, such an algorithm must consider paths that correspond to every subset of I.

Stoller et al. [2007] capture this requirement in what they call Stage 2 ("forward analysis") of their backward-search algorithm. The algorithm maintains a subset of I as an annotation in the state-reachability graph (or "plan," as they call it). They observe that their algorithm is doubly exponential in the size of I.

Mixed Roles. A mixed role is one that appears with negation and without in preconditions of *can_assign* rules.[3] Stoller et al. [2007] show that the verification problem is fixed parameter tractable in the number of mixed roles. To see why the number of mixed roles is a source of complexity, consider the case that no role is mixed.

An algorithm can simply adopt the greedy approach of maximally assigning u to every role r_p that appears without negation, and revoking u from every role r_n that appears negated. Such an approach will not work for a mixed role. Given a mixed role r_m, it is possible that we may need to repeatedly assign u to it, and revoke u from it on a path to a state in which u is assigned to r.

A search algorithm must decide whether to revoke u from r_m in every state in which he is assigned to r_m, and whether to assign u to r_m in every state in which he is not assigned to r_m. In the worst case, every such combination must be tried for every mixed role.

Positive Precondition Roles. A positive precondition role is a role that appears without negation in a precondition. The number of positive precondition roles is a source of complexity. Sasturkar et al. [2011] and Stoller et al. [2007] observe that if we restrict each *can_assign* rule to only one positive precondition role, then the verification problem becomes fixed parameter tractable in the number of irrevocable roles.

An intuition behind this is that if there is at most one positive precondition role in every precondition of the *can_assign* rules, then the resultant CNF expression for which the model checker checks satisfiability comprises only of Horn clauses. We know

[2]As we discuss in Section 2.1, disjunctions are disallowed in an individual *can_assign* rule. However, multiple rules with the same target role results in a disjunction of the preconditions of those rules. In our example, if $c_i = r_{i,1} \vee \cdots \vee r_{i,m}$, then we assume that we have the following *can_assign* rules with r_i as the target role: $\langle r_a, r_{i,1} \wedge r_{i-1}, r_i\rangle, \ldots, \langle r_a, r_{i,m} \wedge r_{i-1}, r_i\rangle$.

[3]We point out that a role does not appear with and without negation in the same *can_assign* rule. This is because conjunction and negation are the only operators in a rule (see Section 2.1), and therefore such a precondition is always false.

that Horn Satisfiability is in **P**. If this restriction is lifted, then the corresponding satisfiability problem is NP-complete, as we discuss previously under "Disjunctions."

We point out that these sources of complexity are not unique to ARBAC. The access matrix scheme due to Harrison et al. [1975], for example, has preconditions in its state-change rules as well. Similarly, in the context of RBAC, the work of Crampton and Loizou [2003] on the scoped administration of RBAC, has what they call conditions on state-changes that are very similar to the preconditions of ARBAC.

4.2. Mohawk and the Sources of Complexity

An aspect from our approach that assuages the complexity is that we are goal-oriented in our abstraction-refinement algorithm (see Section 3). Recall that we create a priority queue of roles that are *Related-by-Assignment* to the query role, which is the role in the safety instance. This stratification of roles helps us eliminate roles that cannot affect the membership to the query role. A consequence of this is that a number of paths from the start-state that do not lead to the error-state are removed.

Another aspect is that we optimistically look for short paths that lead from the start state to the error state, while not burdening the model checker with a lot of extraneous input. We first check whether we can reach the error state in zero transitions. In doing so, we ensure that the model checker is provided no state-change rules. We then check whether we can reach it in only a few transitions. In doing so, we provide the model checker with only those state-change rules that may be used for those few transitions. And so on.

Every source of complexity is associated with an intractable problem. For example, disjunctions are associated with satisfiability of Boolean expressions in Conjunctive Normal Form (CNF). For a model-checker to check whether there is an error requires it to check whether a Boolean expression in CNF that is embedded in the broader problem instance is satisfiable. The two aspects we discuss previously result in fewer clauses in the corresponding Boolean expression in the abstract policy and its refinements.

The numbers of irrevocable, mixed and positive precondition roles are fewer in the abstract policy and its refinements as well. Also, they pertain to fewer target roles. Consequently, the corresponding instances of intractable problems are smaller, and there are fewer possible paths for the model checker to explore than if such roles are strewn across rules for several target roles.

In Bound Estimation, we look for portions of the input instance that are not associated with these sources of complexity, and we tighten our estimate of the diameter of the state-reachability graph as a consequence. For example, in Tightening 5 in Section 3.2, we look for roles in preconditions on which the role in the safety query depends positively only. A consequence of such a property is that that part of the input instance does not lend to intractability. That is, even though safety analysis in ARBAC is intractable in general, if we can identify portions of the input instance that are not sources of complexity, we can exploit that and provide a corresponding input (in this case a bound) to the model checker so it can be more efficient. Our empirical assessment that we discuss in the following section bears out these discussions.

5. RESULTS

Our experimental evaluation compares MOHAWK to current state-of-the-art verification tools for ARBAC policies, namely symbolic model checking, bounded model checking, and RBAC-PAT [Stoller et al. 2007]. We chose NuSMV [NuSMV 2012] as the reference implementation for both symbolic and bounded model checking. In the following, we use the terms MC and BMC to refer to NuSMV's symbolic model checker

and bounded model checker respectively. Our evaluation focused on ascertaining the efficiency and the scalability of the tools for verification.

We measure the efficiency based on the time taken to find an error. We measure the scalability with respect to the sources of complexity (Section 4), namely number of roles, and four aspects of preconditions (number of disjunctions, number of irrevocable roles, number of mixed roles, and number of positive preconditions) in total in the input policies. Our case study (see Section 5.1) establishes that such features are required for creating realistic ARBAC policies. Our results can be summarized as follows.

Where MOHAWK *Performs Better.* The mindset behind MOHAWK is that policies are likely to contain errors, and most of these errors can be found in only a few refinements. In other words, we are optimistic about finding errors. Consequently, we undergo the additional overhead of abstraction-refinement. For complex policies in which it is likely that there are several errors at various levels of refinement, it is likely that MOHAWK will outperform conventional approaches. MOHAWK is also likely to perform better on policies that contain several sources of complexity (see Section 4).

Where Other Tools May Perform Better. Other tools may perform better than MOHAWK in two particular cases in which abstraction-refinement is unnecessary overhead. One is for the subcases for which the problem is in **P**. In this case, conventional model checking is likely to perform better than MOHAWK. Another is the case that there are only a few errors, and these errors require several refinement steps. It is not necessarily true that MOHAWK will perform worse in this case, as the fact that we use bounded model checking may in itself mitigate the effects of the overhead from abstraction-refinement. However, it is possible that in such a case, abstraction-refinement adds overhead which may cause it to perform worse than other approaches.

In the following sections, we describe our case study, benchmarks, experimental methodology and provide a summary of results.

5.1. Case Study

We conducted a case study for banking that has been vetted by a major financial institution [Jayaraman et al. 2012]. Our case-study has a number of similarities to an earlier case-study [Schaad et al. 2001]. We discussed our case study with the representatives of a leading bank, who agreed that enforcing separation of privilege is crucial for their operations and that our example is realistic.[4]

In our case study, we consider a bank that has several branches. Each branch replicates a role-hierarchy design. The bank has a separation of privilege constraint that they would like enforced in every branch. Specifically, each branch has four sets of five non-managerial roles each. The sets correspond to business divisions (e.g., financial analysis and e-commerce) in a branch. The separation of duty requirement is that an employee is a member of at most three of those five roles at any given time. This separation of duty requirement applies to every branch. We refer the reader to our technical report [Jayaraman et al. 2012] for a more detailed discussion; here, we assume that the five roles in a given set are $r_1, \ldots, r_5$. There is also a role, say r_0, of which anyone that intends to be a member of any of $r_1, \ldots, r_5$ must be a member.

It is possible to capture this separation of privilege requirement in ARBAC. We assume that in the start (or current) authorization state, the requirement is not violated in any division of any branch. Then, we can encode the requirement using preconditions to state-changes. For each role $r_i \in \{r_1, \ldots, r_5\}$, we associate six *can_assign* rules

[4]Personal communication with the security architect of a leading bank.

with that role as the target. For example, with r_1 as the target role, we would have the following *can_assign* rules (r_a is an administrative role):

$$\langle r_a, r_0 \wedge \neg r_2 \wedge \neg r_3, r_1 \rangle, \langle r_a, r_0 \wedge \neg r_2 \wedge \neg r_4, r_1 \rangle, \langle r_a, r_0 \wedge \neg r_2 \wedge \neg r_5, r_1 \rangle$$
$$\langle r_a, r_0 \wedge \neg r_3 \wedge \neg r_4, r_1 \rangle, \langle r_a, r_0 \wedge \neg r_3 \wedge \neg r_5, r_1 \rangle, \langle r_a, r_0 \wedge \neg r_4 \wedge \neg r_5, r_1 \rangle.$$

We replicate this for each role $r_1, \ldots, r_5$ as the target role for each business division in each branch of the bank. We ran our case study for 10, 20, 30, and 40 branches after introducing an error. An error comprises additional policy elements that cause the above requirement, that a user is a member of at most three of $r_1, \ldots, r_5$, to be violated.

As the error, we introduced *can_assign* rules of the form $\langle r_a, \text{true}, r_1 \rangle, \ldots, \langle r_a, \text{true}, r_4 \rangle$ for roles $r_1, \ldots, r_4$ in a particular division of a particular branch. To encode our property of interest, we extended the query interface of MOHAWK. With the extension, MOHAWK accepts queries of the form $\langle u, R \rangle$, where $R = \left(r_{1,1} \wedge r_{1,2} \wedge \cdots \wedge r_{1,k_1}\right) \vee \left(r_{2,1} \wedge \cdots \wedge r_{2,k_2}\right) \vee \cdots \vee \left(r_{m,1} \wedge \cdots \wedge r_{m,k_m}\right)$, and each $r_{i,j}$ is a role in the policy. That is, R is a disjunction of clauses, each of which is a conjunction of roles. There is a straightforward efficient reduction from the verification problem with such generalized queries to the problem that allows queries of the form $\langle u, r \rangle$ only, where r is a single role. Nonetheless, the generalized query interface may be more usable, and allows one to more clearly isolate the property of interest from the policy.

For our case study, a query is a disjunction of five clauses, each of which is the conjunction of four roles from amongst $r_1, \ldots, r_5$. This expresses the query we seek to ask: whether u could become a member of any of the four roles simultaneously.

We ran our case study against all the tools; we present our results in Table II. As the table indicates, MOHAWK and BMC performed reasonably well. The results clearly show the power of providing the model checker with a bound. MOHAWK fares worse than BMC owing to its additional overhead of abstraction-refinement.

Our experience with the case study provides two additional insights about realistic policies. First, our case study affirms that the sources of complexity that we discuss in Section 4 occur in realistic policies. Second, realistic ARBAC policies are not amenable to static pruning techniques such as those proposed in prior work [Jha et al. 2008; Stoller et al. 2007]. Static pruning techniques are effective in cases where the role in question is dependent in particular ways only on a small set of roles, irrespective of the size of the policy (see Section 3.4). Such a dependency does not exist for the encoding of the separation of privilege property on which Table II is based, and for other properties we have investigated in our case-study. The following table quantitatively indicates the effect of static pruning (the ones from [Jha et al. 2008; Stoller et al. 2007] plus the one we introduce in Section 3.2.1). The "Unpruned" and "Pruned" columns each contains a 3-tuple, $\langle$# roles, # *can_assign* rules, # *can_revoke* rules$\rangle$.

# Branches	Unpruned	Pruned
10	$\langle 343, 1904, 321 \rangle$	$\langle 252, 1454, 240 \rangle$
20	$\langle 683, 3804, 641 \rangle$	$\langle 502, 2904, 480 \rangle$
30	$\langle 1023, 5704, 961 \rangle$	$\langle 752, 4354, 720 \rangle$
40	$\langle 1363, 7604, 1281 \rangle$	$\langle 1002, 5804, 960 \rangle$

5.2. Benchmarks Used

We used two sets of policies in our evaluation. Both are based on prior work [Gofman et al. 2009; Jha et al. 2008; Schaad et al. 2001; Stoller et al. 2007]. The first set has not been used previously; we built it based on our case study, and the sources of complexity, given that they do indeed occur in practice. Our second set of policies has been used in

Table II. Evaluation of Model-Checking, Bounded Model-Checking, RBAC-PAT, and MOHAWK on Various Benchmarks

		Num. of Roles, Rules	MC	BMC	RBAC-PAT		MOHAWK w/ MC	MOHAWK w/ BMC
					Forward reachability	Backward reachability		
Case Study	1.	343, 2226	M/O	19s	T/O	M/O	M/O	48s
	2.	683, 4446	M/O	41s	T/O	M/O	M/O	4m 26s
	3.	1023, 6666	M/O	1m 13s	T/O	M/O	M/O	14m 36s
	4.	1363, 8885	M/O	M/O	T/O	M/O	M/O	M/O
Test suite 1 *Poly-time verifiable*	1.	3, 15	0.097s	0.016s	0.625s	0.240s	0.542s	0.456s
	2.	5, 25	0.050s	0.025s	0.695s	0.281s	0.554s	0.484s
	3.	20, 100	M/O	0.103s	0.806s	M/O	T/O	1.051s
	4.	40, 200	M/O	0.110s	0.780s	M/O	0.556s	0.514s
	5.	200, 1000	M/O	0.624s	1.471s	M/O	1.012s	0.645s
	6.	500, 2500	M/O	3.2s	2.177s	M/O	1.607s	0.768s
	7.	4000, 20000	M/O	414s	7.658s	M/O	3.185s	1.905s
	8.	20000, 80000	M/O	M/O	110s	M/O	28.992s	22.436s
	9.	30000, 120000	M/O	M/O	210s	M/O	1m 10s	1m 2s
	10.	40000, 200000	M/O	M/O	6m 16s	M/O	4m 28s	2m 35s
Test suite 2 *NP-Complete*	1.	3, 15	0.022s	0.021s	0.513s	0.241s	0.429s	0.435s
	2.	5, 25	0.064s	0.026s	0.519s	0.252s	0.485s	0.472s
	3.	20, 100	M/O	0.048s	0.512s	M/O	0.441s	0.437s
	4.	40, 200	M/O	0.122s	0.534s	M/O	T/O	1.118s
	5.	200, 1000	M/O	0.472s	0.699s	M/O	0.607s	0.615s
	6.	500, 2500	M/O	1.819s	2.414s	M/O	0.705s	0.717s
	7.	4000, 20000	M/O	109s	311s	M/O	1.722s	1.713s
	8.	20000, 80000	M/O	M/O	T/O	M/O	17.7s	17.708s
	9.	30000, 130000	M/O	M/O	T/O	M/O	41.334s	40.690s
	10.	40000, 200000	M/O	M/O	T/O	M/O	3m 6s	2m 9s
Test suite 3 *PSPACE-Complete*	1.	3, 15	0.030s	0.102s	1.452s	0.665s	0.423s	0.432s
	2.	5, 25	0.044s	0.033s	1.666s	0.881s	0.497s	0.486s
	3.	20, 100	M/O	0.056s	1.364s	M/O	0.452	0.448s
	4.	40, 200	M/O	0.169s	1.476s	M/O	T/O	1.161s
	5.	200, 1000	M/O	0.972s	2.258s	M/O	0.615s	0.613s
	6.	500, 2500	M/O	2.422s	7.350s	M/O	0.716s	0.730s
	7.	4000, 20000	M/O	109s	511s	M/O	0.722s	1.717s
	8.	20000, 80000	M/O	M/O	T/O	M/O	17.666s	18.406s
	9.	30000, 130000	M/O	M/O	T/O	M/O	47s	45s
	10.	40000, 200000	M/O	M/O	T/O	M/O	2m 16s	2m 45s
Simple Policies	1.	12, 19	0.025s	0.022s	0.531s	0.238s	0.553s	0.429s
	2.	20, 266	0.023s	0.026s	0.559s	0.247s	0.426s	0.415s
	3.	32, 162	M/O	0.182s	1.556s	0.568s	40s	0.735s

m - minutes
s - seconds
ms - milliseconds
MC - NuSMV symbolic model checking
BMC - NuSMV bounded model checking
RBAC-PAT - Tool from [Gofman et al. 2009; Stoller et al. 2007]
M/O - Memory out
MOHAWK w/ MC - MOHAWK symbolic model checking mode
MOHAWK w/ BMC - MOHAWK bounded model checking mode
T/O - Time out after 60 mins

the context of verification of ARBAC policies in prior work [Gofman et al. 2009; Stoller et al. 2007]. These policies are simpler than the policies in the first set.

Complex Policies (Set 1). We have created a test suite based on the case study, and the sources of complexity. The number of roles (or size of the policy) and type of state-change rules are sources of complexity in ARBAC policies. Depending on the type of state-change rules, the safety analysis problem for ARBAC is PSPACE-Complete, NP-Complete, or solvable in polynomial time [Jha et al. 2008]. Accordingly, we have created three sets of complex test suites with varying gradations of roles.

— *Test Suite 1.* Policies with positive conjunctive *can_assign* rules and a nonempty set of *can_revoke* rules. Verification is polynomial-time for these policies.

```
1 while |can_revoke| < no-of-cr do
2     target-role ← a random role ∈ roles that is ∉ all-target-roles[can_revoke]
3     can_revoke ← can_revoke ∪ {⟨adminrole, target-role⟩}
```

Fig. 7. The algorithm for generating *can_revoke* rules. *no-of-cr* specifies the number of desired *can_revoke* rules, *roles* is the set of roles, and *adminrole* is the administrative role . We use the auxiliary routine *all-target-roles[*can_revoke *]* which returns the set of all target roles (so far) in *can_revoke*.

```
1 foreach role ∈ roles do
2     target-role ← role
3     foreach i = 1 → rules-per-role do
4         preconditions ← ∅
5         foreach j = 1 → no-of-preconditions do
6             precondition-role ← a random role ∈ roles that is ≠ target-role
7             preconditions ← preconditions ∪ {+precondition-role}
8         can_assign ← can_assign ∪ {⟨adminrole, preconditions, target-role⟩}
```

Fig. 8. The algorithm for generating positive *can_assign* rules. *rules-per-role* specifies the desired number of *can_assign* rules per role, *no-of-preconditions* specifies the desired number of preconditions in a *can_assign* rule, *roles* is the set of roles, and *adminrole* is the administrative role.

— *Test Suite 2.* Policies with mixed conjunctive *can_assign* rules and no *can_revoke* rules. Verification is NP-Complete for these policies.
— *Test Suite 3.* Policies with mixed conjunctive *can_assign* rules and a nonempty set of *can_revoke* rules. Verification is PSPACE-Complete for these policies.

Each of the complex policies were created as follows. First, we create a policy with the desired number of roles. Then, we add only two users, namely an admin user and a normal user who would be used in the safety question. This is because number of users is not a source of complexity. Following that we generate the desired type of *can_assign* and *can_revoke* rules based on the test suite.

Figures 7, 9, and 8 describe the algorithms we use for generating *can_revoke* rules, positive conjunctive *can_assign* rules, and mixed conjunctive *can_assign* rules. For test suite 1, we used the algorithms described in Figures 7 and 8. For test suite 2, we used the algorithm described in Figure 9. For test suite 3, we used the algorithms described in Figures 7 and 9.

For each complex policy, we identified a user-role pair at random such that the role is reachable by an unauthorized user. Our results are for verifying this question. Recall that the basic safety question is determining whether a unauthorized user u can reach a role r. In our complex policies, a majority of the roles in the policy are related to the target role in the safety query. This is realistic, as it is similar to the policy in our case study.

Simple Policies (Set 2). This set comprises three ARBAC policies that have been used in previous work for the evaluation of RBAC-PAT [Gofman et al. 2009; Sasturkar et al. 2011; Stoller et al. 2007]. The first policy is for a hypothetical hospital, and the second policy is for a hypothetical university. The third policy from Gofman et al. [2009] is a test case that has mixed preconditions. The first two policies were used in Stoller et al. [2007] for case studies. The third policy was used in Gofman et al. [2009], and a complete state-space exploration is reported to have taken 8.6 hours in RBAC-PAT. An important restriction in these policies is that they have at most one positive precondition per *can_assign* rule. As we explain in Section 5.4.2, answering the safety question for these policies was fairly easy for all the tools.

```
1  foreach role ∈ roles do
2      target-role ← role
3      foreach i = 1 → rules-per-role do
4          preconditions ← ∅
5          mixed-role ← a random role ∈ roles that is ≠ target-role
6          toggle ← true
7          foreach j = 1 → no-of-preconditions do
8              precondition-role ← a random role ∈ roles that is ∉
               {target-role, mixed-role}
9              preconditions ← preconditions ∪ {+precondition-role}
10             if toggle then
11                 preconditions ← preconditions ∪ {+mixed-role}
12             else
13                 preconditions ← preconditions ∪ {¬mixed-role}
14             toggle ← ¬toggle
15         can_assign ← can_assign ∪ {⟨adminrole, preconditions, target-role⟩}
```

Fig. 9. The algorithm for generating mixed *can_assign* rules. *rules-per-role* specifies the desired number of *can_assign* rules per role, *no-of-preconditions* specifies the desired number of preconditions in a *can_assign* rule, *roles* is the set of roles, and *adminrole* is the administrative role. *mixed-role* is the role that is picked as the choice for a mixed role in the current group of *can_assign* rules to be generated. In lines 10-13, depending on the value of *toggle*, *mixed-role* is added as either a positive or negative precondition role. In line 11, *mixed-role* is added as a positive precondition role, and in line 13, *mixed-role* is added as a negative precondition role.

5.3. Experimental Methodology

All the experiments were conducted on a Macbook Pro laptop with an Intel Core 2 Duo 2.4 GHz processor and 4GB of RAM.

In all the experiments, the input to the verification tools consisted of an ARBAC policy and a safety question. We applied the static pruning techniques proposed in prior work [Jha et al. 2008; Stoller et al. 2007] on all the policies prior to the experiments. The policies were encoded using the input language of the respective tools. MC and BMC use the SMV finite state machine language, while RBAC-PAT and MOHAWK have their own input language. We implemented a translation tool to convert policies in MOHAWK's input language to both SMV and RBAC-PAT input languages. We expected the tools to conclude that the role is reachable and provide the sequence of administrative actions that lead to the role assignment. In our evaluation, we had two users for each policy, namely the user in the safety question and the administrator. These are the only users required for answering the safety question. Moreover, static pruning techniques remove all users but the one that is relevant to the safety question.

5.4. Results Explained

We explain the results of our experimental evaluation below. Whenever we refer to MOHAWK here, we mean MOHAWK configured with aggressive abstraction-refinement, unless specified otherwise. For the policies in Table II, on average we needed 2 refinements. The worst-case was for the policy from our case study, for which we needed 7 refinements.

5.4.1. Results on Complex Policies. The results of our evaluation on complex policies (Set 1) are contained in Table II. Our results show that MOHAWK scales better than all competing tools, irrespective of the complexity of the input policy.

There are two differences between "BMC" and "MOHAWK w/ BMC". First, "MOHAWK w/ BMC" uses the abstraction-refinement approach. Second, "BMC" uses 10 as the

bound, but "MOHAWK w/ BMC" estimates the diameter for the policy in each iteration and uses that as the bound. Therefore, "MOHAWK w/ BMC" is a verification tool, but "BMC" is an error-finding tool.

MOHAWK improves the scalability of both BMC and MC. Although BMC scales better than MC, both the tools run out of memory for large policies. In contrast, MOHAWK, when used on top of either MC or BMC, scales better for all the three test suites irrespective of their complexity class. This further underlines the effectiveness of the abstraction-refinement based technique in MOHAWK. Also, having a single technique that can perform well on large real-world policies that belong to different complexity classes is also useful from the point of view of maintaining and extending the tool. Furthermore, since our technique is not tied to specific model-checking algorithms, it can be used in conjunction with other algorithms such as RBAC-PAT's forward reachability.

The RBAC-PAT forward reachability algorithm is very effective for test suite 1, whose policies are verifiable in polynomial time. However, for the other two test suites, the RBAC-PAT forward reachability is faster compared to MC and slower compared to both BMC and MOHAWK. RBAC-PAT's backward reachability algorithm was faster compared RBAC-PAT's forward algorithm for a few small policies, but ran out of memory for majority of the policies in all the test suites.

MOHAWK performs worse compared to BMC for small policies in test suite 2. This is because these policies are so small that BMC can analyze them easily, but MOHAWK takes multiple iterations to arrive at the same answer. In other words, the policies are too simple, and the abstraction-refinement step creates unnecessary overhead.

5.4.2. Results on Simple Policies. The results of our evaluation on simple policies are summarized in Table II. The first and second policies did not satisfy separate administration restriction, so we removed *can_assign* rules that have the administrative roles as target and used the modified policies in our evaluation.

BMC, RBAC-PAT, and MOHAWK were effective for all the three policies. The differences in time taken to verify are not very significant because they are less than a second.

MOHAWK with aggressive abstraction-refinement is faster compared to RBAC-PAT's forward reachability, but slower compared to BMC and RBAC-PAT's backward reachability for these policies. However, the slow down in each case is less than a second and is imperceptible to the user. MOHAWK's slowdown when compared to BMC is expected because of the small size of the policies. For the third test case, MC timed out. Both RBAC-PAT and MOHAWK with aggressive abstraction-refinement are slower compared to BMC, and MOHAWK is faster compared to RBAC-PAT's forward reachability algorithm and slightly slower compared to RBAC-PAT's backward reachability.

6. RELATED WORK

This article is closely related to our earlier work [Jayaraman et al. 2011]. Whereas that work proposes the use of abstraction-refinement only, in this work, we complement abstraction-refinement with bound-estimation. The incorporation of bound-estimation affects the architecture of MOHAWK (Figure 2 and Section 3), how MOHAWK reconciles the sources of complexity (Section 4.2) and our empirical results (Section 5). In addition, in this work, we clearly articulate the correctness of abstraction-refinement (Section 3.1.6) and of MOHAWK (Section 3.3).

Counterexample guided abstraction refinement was originally developed in the context of model checking [Clarke et al. 2003]. Since then the basic idea has been adapted in different ways in the context of bounded model-checking for hardware verification [Kroening 2006] and program analysis [Ball and Rajamani 2002] to verify computer

programs. The idea of abstraction refinement has also been adapted in the context of solvers for various theories such as modular and integer linear arithmetic [Ganesh and Dill 2007]. To the best of our knowledge, MOHAWK is the first tool to adapt the paradigm of abstraction-refinement for verifying access-control policies.

More recent work [Ferrara et al. 2012] has adopted an abstract-interpretation approach for the same problem that we address. They apply an abstraction function to the ARBAC policies that converts them into imperative programs. Then they apply standard abstract-interpretation program analysis techniques to prove correctness properties of these policies. It appears that their performance numbers are orders of magnitude worse than MOHAWK for verification of similar-sized policies; however, it is possible that the problem instances were not the same.

We can classify verification problems in the context of access control broadly into two categories: fixed-state, and with state changes.

The work that falls in fixed-state considers only a given state, and verification of properties within that state. Examples of work that fall in this category include those of Jha and Reps [2004], Jha et al. [2006], Hughes and Bultan [2008], Hu and Ahn [2008], Martin and Xie [2007], Rao et al. [2007], Kolovski et al. [2007], Zhao et al. [2005], and Fisler et al. [2005]. It is conceivable that our approach can be used in such fixed-state contexts. For example, the work of Martin and Xie [2007] considers testing of XACML policies for errors by introducing faults in to such policies. They then determine whether a particular test for errors is effective by checking whether it finds the fault that is intentionally introduced. We seek to find errors, and to such an end, we could exploit the modularity that is inherent to XACML policies and use abstraction-refinement. We could abstract most of a policy away, and then gradually refine it by adding components back, while testing for errors against access requests (as proposed by Martin and Xie [2007]) at each step.

Plain model-checking approach has also been proposed for some state-change schemes [Zhang et al. 2008]. As we have shown in Section 5, plain model checking does not scale adequately for verifying policies of very large sizes.

Work on safety analysis dates back to the mid-1970's; the work by Harrison et al. [1975] is considered foundational work in access control. They were the first to provide a characterization of safety. They show also, that safety analysis for an access matrix scheme with state changes specified as commands in a particular syntax is undecidable. Since then, there has been considerable interest and work in safety, and more generally, security analysis in the context of various access control schemes.

Safety analysis in monotonic versions of the HRU scheme has been studied in Harrison and Ruzzo [1978]. Jones et al. [1976] introduced the Take-Grant scheme, in which safety is decidable in linear time. Amman and Sandhu consider safety in the context of the Extended Schematic Protection Model (ESPM) [Ammann and Sandhu 1991] and the Typed Access Matrix model Sandhu [1992], Budd [1983], and Motwani et al. [2000] studied grammatical protection systems. Soshi [2000] studied safety analysis in Dynamic-Typed Access Matrix model. These models all have subcases where safety is decidable. Solworth and Sloan [2004] introduced discretionary access control model in which safety is decidable. This thread of research has proposed many new access control schemes, but has had limited impact on access control systems used in practice. This is potentially because the proposals were either too simplistic or too arcane to be useful. The focus of this article is ARBAC, which was primarily proposed to meet the need of expressive access control schemes required for large-scale real-world deployments.

To our knowledge, Li and Tripunitara [2004] were the first to consider security analysis in the context of ARBAC. Jha et al. [2008] were the first to consider the use of model checking to for the verification problem of ARBAC. That work also identifies that

the verification problem for ARBAC is PSPACE-complete. Subsequently, Stoller et al. [2007] established that user-role reachability analysis is fixed parameter tractable with respect to number of mixed roles, irrevocable roles, positive preconditions, and goal size. Furthermore, they have proposed new model-checking algorithms for similar verification problems and implemented them in a tool called RBAC-PAT [Gofman et al. 2009].

RBAC-PAT contains two algorithms for analyzing ARBAC policies, namely forward reachability and backward reachability. As we have shown in Section 5, forward reachability algorithm scales better compared to plain model checking, is effective for polynomial-time-verifiable policies, but does not scale adequately with complexity of the policies. We could not extensively evaluate the backward reachability algorithm because the implementation gave a segmentation fault for even moderately sized policies. In contrast, MOHAWK scales better and is efficient for identifying errors irrespective of the complexity of the policies. The key reason for MOHAWK's effectiveness is the abstraction-refinement approach that is goal oriented and optimally looks short paths that lead from the start path to the error state (Section 4.2).

Gofman et al. [2010] proposes incremental algorithms for analyzing the impact of changes to ARBAC policies by taking advantage of previous analysis results. Such incremental analysis is outside the scope of this article.

7. CONCLUSION

We presented an abstraction-refinement and bound-estimation based technique, and its implementation, the MOHAWK tool, for verifying ARBAC access-control policies. MOHAWK accepts an access-control policy and a safety question as input, and outputs whether or not an error is found. We extensively evaluated MOHAWK and compared our results to those for current state-of-the-art tools for policy analysis. Our experiments show that MOHAWK scales well with the complexity of realistic policies. Analysis tools such as MOHAWK enable policy administrators to quickly analyze policies prior to deployment, thereby increasing the assurance of the system.

ACKNOWLEDGMENTS

We thank Mikhail Gofman, Scott Stoller, C. R. Ramakrishnan, and Ping Yang for providing access to the RBAC-PAT tool and their experimental data. We thank the anonymous reviewers for their helpful comments that have greatly improved the article, our MOHAWK tool and our case study.

REFERENCES

Ammann, P. and Sandhu, R. 1991. Safety analysis for the extended schematic protection model. In *Proceedings of the IEEE Symposium on Security and Privacy*. 87–97.

Aveksa. 2012. What is business-driven identity and access management? `http://www.aveksa.com/what-we-do/What-Is-Business-Driven-Identity-and-Access-Management.cfm`.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, New York, 1–3.

Budd, T. A. 1983. Safety in grammatical protection systems. *Int. J. Paral. Prog. 12,* 6, 413–431.

Clarke, E., Biere, A., Raimi, R., and Zhu, Y. 2001. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des. 19,* 1, 7–34.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM 50,* 5, 752–794.

Clarke, E., Kroening, D., Ouaknine, J., and Strichman, O. 2005. Computational challenges in bounded model checking. *Softw. Tools Tech. Trans. 7,* 2, 174–183.

Clarke, E. M., Grumberg, O., and Peled, D. A. 1999. *Model Checking*. The MIT Press.

Crampton, J. and Loizou, G. 2003. Administrative scope: A foundation for role-based administrative models. *ACM Trans. Inf. Syst. Secur. 6,* 2, 201–231.

Ferraiolo, D. F., Kuhn, D. R., and Chandramouli, R. 2003. *Role-Based Access Control*. Artech House, Inc., Norwood, MA.

Ferrara, A. L., Madhusudan, P., and Parlato, G. 2012. Security analysis of access control through program verification. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF'12)*. IEEE Computer Society, Cambridge, MA.

Fisler, K., Krishnamurthi, S., Meyerovich, L. A., and Tschantz, M. C. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, 196–205.

Ganesh, V. and Dill, D. L. 2007. A decision procedure for bitvectors and arrays. In *Proceedins of the 19th International Conference on Computer-Aided Verification*. Lecture Notes in Computer Science, Springer, Berlin, 519–531.

Gofman, M. I., Luo, R., Solomon, A. C., Zhang, Y., Yang, P., and Stoller, S. D. 2009. Rbac-pat: A policy analysis tool for role based access control. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 5505, Springer-Verlag, 46–49.

Gofman, M. I., Luo, R., and Yang, P. 2010. User-role reachability analysis of evolving administrative role based access control. In *Proceedings of the 15th European Conference on Research in Computer Security (ESORICS'10)*. Springer-Verlag, Berlin, 455–471.

Graham, G. S. and Denning, P. J. 1972. Protection — Principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*. Vol. 40, AFIPS Press, 417–429.

Harrison, M. A. and Ruzzo, W. L. 1978. Monotonic protection systems. In *Proceedings of the Conference on Foundations of Secure Computation*. 461–471.

Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. 1975. On protection in operating systems. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP'75)*. ACM, New York, 14–24.

Hu, H. and Ahn, G. 2008. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SAC'08)*. ACM, New York, 195–204.

Hu, V. C., Kuhn, D. R., and Xie, T. 2008. Property verification for generic access control models. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE Computer Society, Los Alamitos, CA, 243–250.

Hughes, G. and Bultan, T. 2008. Automated verification of access control policies using a sat solver. *Int. J. Softw. Tools Technol. Transf. 10,* 6, 503–520.

Jayaramam, K. 2012. Mohawk – Automatic Verification of Access-Control Policies. `http://code.google.com/p/mohawk/`.

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

Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M. C., and Chapin, S. J. 2012. Real-world case studies of using ARBAC to enforce separation-of-duty constraints. `http://kjayaram.mysite.syr.edu/mohawk/casestudy.pdf`.

Jha, S. and Reps, T. W. 2004. Model Checking SPKI/SDSI. *J. Comput. Sec. 12,* 3–4, 317–353.

Jha, S., Schwoon, S., Wang, H., and Reps, T. 2006. Weighted Pushdown Systems and Trust-Management Systems. In *Proceedings of TACAS*. Springer-Verlag, Berlin.

Jha, S., Li, N., Tripunitara, M., Wang, Q., and Winsborough, W. 2008. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable Secur. Comput. 5,* 4, 242–255.

Jones, A. K., Lipton, R. J., and Snyder, L. 1976. A linear time algorithm for deciding security. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science (SFCS'76)*. IEEE Computer Society, Washington, DC, 33–41.

Kern, A. 2002. Advanced features for enterprise-wide role-based access control. In *Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)*. IEEE Computer Society, Washington, DC, 333.

Kolovski, V., Hendler, J., and Parsia, B. 2007. Analyzing web access control policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. ACM, New York, 677–686.

Kroening, D. 2006. Computing over-approximations with bounded model checking. *Electron. Notes Theor. Comput. Sci. 144*, 79–92.

Li, N. and Tripunitara, M. V. 2004. Security analysis in role-based access control. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT'04)*. ACM, New York, 126–135.

Li, N. and Tripunitara, M. V. 2006. Security analysis in role-based access control. *ACM Trans. Inf. Syst. Secur. 9,* 4, 391–420.

Li, N., Mitchell, J. C., and Winsborough, W. H. 2005. Beyond proof-of-compliance: Security analysis in trust management. *J. ACM 52,* 3, 474–514.

Martin, E. and Xie, T. 2007. A fault model and mutation testing of access control policies. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. ACM, New York, 667–676.

Motwani, R., Panigrahy, R., Saraswat, V., and Ventkatasubramanian, S. 2000. On the decidability of accessibility problems (extended abstract). In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC'09)*. ACM, New York, 306–315.

NuSMV. 2012. `http://nusmv.irst.itc.it/`.

Rao, P., Lin, D., and Bertino, E. 2007. XACML function annotations. In *Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'07)*. IEEE Computer Society, Washington, DC, 178–182.

SailPoint. 2012. Policy enforcement. http://www.sailpoint.com/products/identity-iq/compliance-manager/policy-enforcement.php.

Saltzer, J. H. and Schroeder, M. D. 1975. The protection of information in computer systems. *Proc. IEEE*.

Sandhu, R., Bhamidipati, V., and Munawer, Q. 1999. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur. 2,* 1, 105–135.

Sandhu, R. S. 1992. The typed access matrix model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*. 122–136.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. 1996. Role-based access control models. *IEEE Computer 29,* 2, 38–47.

Sasturkar, A., Yang, P., Stoller, S. D., and Ramakrishnan, C. 2006. Policy analysis for administrative role based access control. In *Proceedings of the 19th Computer Security Foundations Workshop*. IEEE Computer Society Press.

Sasturkar, A., Yang, P., Stoller, S. D., and Ramakrishnan, C. 2011. Policy analysis for administrative role-based access control. *Theoret. Comput. Sci. 412,* 44, 6208–6234.

Schaad, A., Moffett, J., and Jacob, J. 2001. The role-based access control system of a European bank: A case study and discussion. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*. ACM, New York, 3–9.

Sohr, K., Drouineaud, M., Ahn, G.-J., and Gogolla, M. 2008. Analyzing and managing role-based access control policies. *IEEE Trans. Knowl. Data Eng. 20*, 924–939.

Solworth, J. A. and Sloan, R. H. 2004. A layered design of discretionary access controls with decidable safety properties. In *Proceedings of the IEEE Symposium on Security and Privacy*, 56.

Soshi, M. 2000. Safety analysis of the dynamic-typed access matrix model. In *Proceedings of the Computer Security (ESORICS 2000)*. Lecture Notes in Computer Science, Springer, Berlin, 106–121.

Stoller, S. D., Yang, P., Ramakrishnan, C. R., and Gofman, M. I. 2007. Efficient policy analysis for administrative role based access control. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*. ACM, New York, 445–455.

Zhang, N., Ryan, M., and Guelev, D. P. 2008. Synthesising verified access control systems through model checking. *J. Comput. Secur. 16,* 1, 1–61.

Zhao, C., Heilili, N., Liu, S., and Lin, Z. 2005. Representation and reasoning on RBAC: A description logic approach. In *Proceedings of the 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC'05)*, (Hanoi, Vietnam, October 17–21, 2005). Lecture Notes in Computer Science, Springer, 381–393.

Received April 2012; revised July 2012, September 2012, October 2012; accepted January 2013