Automatic Error Finding in Access-Control Policies

Karthick Jayaraman*
Microsoft
karjay@microsoft.com

Vijay Ganesh
MIT
vganesh@csail.mit.edu

Mahesh Tripunitara
University of Waterloo
tripunit@uwaterloo.ca

Martin Rinard
MIT
rinard@csail.mit.edu

Steve Chapin
Syracuse University
chapin@syr.edu

ABSTRACT

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 a new abstraction-refinement technique for automatically finding 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. Underlying our approach is a change in mindset: we propose that error finding complements verification, can be more scalable, and allows for the use of a wider variety of techniques. In our approach, we use an abstraction-refinement technique to first identify and discard roles that are unlikely to be relevant to the verification question (the abstraction step), and then restore such abstracted roles incrementally (the refinement steps). Errors are one-sided: if there is an error in the abstracted policy, then there is an error in the original policy. If there is an error in a policy whose role-dependency graph diameter is smaller than a certain bound, then we find the error. Our abstraction-refinement technique complements conventional state-space exploration techniques such as model checking. We have implemented our technique in an access-control policy analysis tool. We show empirically that our tool scales well to realistic policies, and is orders of magnitude faster than prior tools.

Categories and Subject Descriptors: K.6.5 [Management and Information Systems]: Security and Protection; D.4.6 [Operating Systems]: Security and Protection; D2.4 [Software Engineering]: Software/Program Verification;

General Terms: Security, verification.

*This work was performed when the author was a graduate student at Syracuse University.

CCS'11, October 17–21, 2011, Chicago, Illinois, USA.

1. INTRODUCTION

This paper presents a technique and a tool for finding errors in 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 [36]) that have considerable expressive 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 to access 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 [35], is 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 reasonably large systems feature several complexity sources that make it difficult to find errors in them. An access-control policy is essentially a finite-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 [11,13,17–21,26,28,30, 39,43,46–48] and practical interest [1,2]. Model checking [8] has emerged as a promising, automated approach to the verification problem [13, 21, 46]. 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 con-

sider is intractable in general (PSPACE-complete [21, 39]), and scalability for practical verification tools remains a significant issue despite considerable progress (see Section 5).

We present a new technique that combines abstraction refinement [6] and bounded model checking 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 [4, 6, 12]. The use of a sophisticated abstraction-refinement algorithm makes our technique significantly more efficient than previous approaches. 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 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.

We have implemented this technique in the MOHAWK access-control policy analysis tool. MOHAWK accepts as input an access-control policy and a safety question. 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 very well as the complexity of the input policies increase, and is orders of magnitude faster than competing tools. The MOHAWK tool is open source and is available from the Google Code website: `http://code.google.com/p/mohawk/`.

Contributions

We make the following contributions in this paper:

1. We describe an *abstraction-refinement* based approach for automatically finding errors in access-control policies (specifically, ARBAC policies). The resulting technique, implemented on top of a bounded model checker, scales very 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. Although we focus on error finding in this paper, our abstraction-refinement technique can be used in conjunction with model checking to perform verification.

   Our technique can also be used for error finding 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 [9] and even the original access matrix scheme due to Harrison et al. [17].

2. An implementation of our technique in a tool called MOHAWK. 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.

3. We provide a detailed experimental comparison of MOHAWK against NuSMV [33], a well known model checking and bounded model checking tool, and RBAC-PAT [13, 46], a tool specifically designed for analyzing ARBAC policies. In comparison to the existing approaches, MOHAWK scales well with the size and complexity of the 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.

Organization: The remainder of our paper is organized as follows. In Section 2, we discuss access-control models and schemes. In Section 3, we describe the architecture of MOHAWK. In Section 4, we describe the sources of complexity from the standpoint of error finding 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 error-finding 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 [15,17] and Role-Based Access Control (RBAC) [10,38]. In this paper, 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

RBAC State	
Roles	{BudgetCommitee, Finance, Acct, Audit, TechSupport, IT, Admin}
Users	{Alice, Bob}
UA	{⟨Bob, Acct⟩, ⟨Bob, Audit⟩, ⟨Alice, Admin⟩}
State-Change Rule	
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⟩}

Figure 1: State and state-change rules for an ARBAC policy

out in Section 2.2 under "The role hierarchy," in the context of this paper, we can reduce the error-finding 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 state-change rule allows for the *delegation* of some state changes to users that may not be fully trusted.

ARBAC [36] and administrative scope [9] 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 [13,30,46], including this paper, 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 error finding. The reason is that in practice, user-role relationships are the most volatile [25]. 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 paper, 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 error finding. One of the consequences of this relates to what has been called the separate administration restriction. We discuss this in Section 2.2 below.

We have omitted some other details that are in the original specification for *URA* [36] because those details are inconsequential to the error-finding 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 error-finding problem

The error-finding 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 [24].

A basic error-finding 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

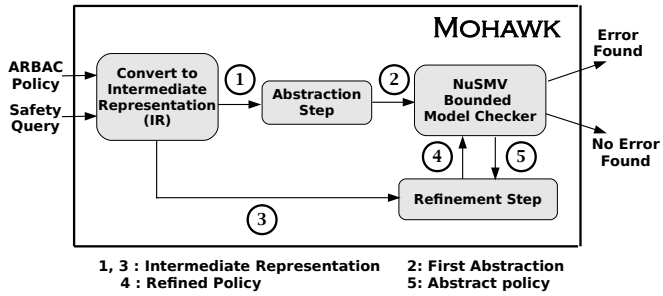

Figure 2: Mohawk Architecture

as the one from above has received considerable attention in the literature [21, 30, 39, 46]. 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 to the basic safety question. This observation has been made before [21, 46].

An instance of an error-finding problem, in the context of this paper, 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 error-finding problem, the separate administration restriction excludes administrative roles from the error-finding problem instance, i.e., administrative roles are not administered by the same rules as "regular" roles. We adopt the separate administration restriction in this paper. Some previous work [46] has considered subcases of when this restriction is lifted. We do not consider those subcases in this paper. We adopt the restriction because the original specification on ARBAC [36] adopts this restriction. Moreover, the separate administration restriction does not affect the complexity of the error-finding problem [40].

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 error finding instance that has a role hierarchy to an error finding instance with no role hierarchy [39].

3. ARCHITECTURE OF MOHAWK

In this section, we describe MOHAWK's architecture (§3.1-§3.6), and illustrate our approach using an example (§3.7).

3.1 Abstraction Refinement in Mohawk

The general idea in an abstraction-refinement [6] technique is to first abstract (over or under approximate) a system represented typically as a logic formula, and check if a desired property holds in the abstraction, and iteratively refine as necessary. If the abstraction is an over approximation, i.e., 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, i.e., 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 over approximation, correctness of the abstraction corresponds with the original. In the case of under approximation, an error detected in the abstraction corresponds with the original.

MOHAWK uses an under approximation strategy [6]. Therefore, any error detected in an abstract policy exists in the original policy. 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, we will refer to the role in the safety query as the query role. Briefly, MOHAWK works on the input as follows:

- **Input Transformation** (§3.2): 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** (§3.3): MOHAWK performs an initial abstraction step to produce an abstract policy.

- **Verification** (§3.4): In this step, MOHAWK invokes the NuSMV bounded model checker on the finite-state machine representation of the current abstract policy. If a counter example is produced by NuSMV, MOHAWK terminates and reports the error found. A counter example, in model-checking parlance, is a sequence of state transitions from the initial state to an error state of the input finite-state machine. 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** (§3.5): 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 k 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 k is one (The most aggressive setting for k).

3.2 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 policy specified in Fig 3 because of the error we introduced in the *can_assign* rule. In Section 3.7, 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 the 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.

3.3 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 k 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'.

3.4 Verification

In the verification step, MOHAWK verifies the safety query in the abstract policy. On each verification step, MOHAWK translates the abstract policy to a finite-state-machine specification in the SMV language and the safety query to a LTL (Linear Temporal Logic) formula. If the model checker identifies a state in which the user is assigned to the role, it provides a counter example. The counter example 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 counter example, MOHAWK reports the error and terminates. Otherwise, MOHAWK refines the abstract policy in the refinement step.

In each step, the abstract policy contains a subset of the roles, UA, *can_assign*, and *can_revoke* rules in the complete policy. Therefore, the abstract policy permits only a subset of the administrative action sequences accepted in the full policy. Each action in the action-sequence identified by the counterexample corresponds to a *can_assign* or *can_revoke* rule that exists in both the abstract and original policies. Therefore, the counter example is true in the original policy also.

3.5 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 k 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.

3.6 On Completeness

Our abstraction-refinement technique is complete — we are no worse than conventional model-checking from the standpoint of completeness. If we do not find any errors at a particular refinement step, we can always continue to refine

```
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;
```

Figure 3: An ARBAC policy in the Mohawk's input language

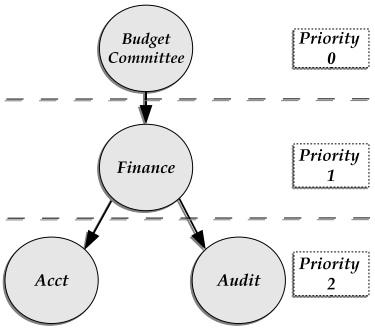

Figure 4: Related-by-assignment (RBA) relationship between roles with respect to BudgetCommittee.

Steps	Users	Roles	UA	*can_assign*	*can_revoke*	Result
Abstraction step	Alice, Bob	BudgetCommittee, Admin	(Alice, Admin)			No counterexample
Refinement 1	Alice, Bob	BudgetCommittee, Admin, Finance	(Alice, Admin)	⟨Admin, Finance, Budget-Committee, ⟩		No counterexample
Refinement 2	Alice, Bob	BudgetCommittee, Admin, Finance, Acct, Audit	(Alice, Admin) (Bob, Acct) (Bob, Audit)	⟨Admin, Finance, BudgetCommittee⟩ ⟨Admin, Acct∧Audit, Finance⟩ ⟨Admin, TRUE, Acct⟩ ⟨Admin, TRUE, Audit⟩	⟨Admin, Acct⟩ ⟨Admin, Audit⟩	**Counterexample found**

Table 1: Illustrating abstraction-refinement steps for the running example in Figure 3

till we check the entire policy. We can improve this approach further by incorporating techniques from the literature on estimating the *completeness threshold* for bounded model checking [7, 27]. Our main point here is that in our technique, we are optimistic about finding errors, and therefore, for a policy that, for example, has no errors, the abstraction-refinement steps that we undergo translate to pure overhead. However, in the case that errors do exist, it is likely that we will find them faster than conventional approaches. Our empirical results (see Section 5) bear this out.

3.7 Example

To illustrate Mohawk's operations, we introduce an error in the policy of our running example in Fig 1. In the *can_assign* rule with target role Finance, we change c from Acct∧¬Audit to Acct∧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 1 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 *can_revoke* rules are added because all of them require roles not added to the abstract policy. NuSMV does not identify a counter example 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 counter example 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 counter example, 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.

3.8 Implementation

We implemented MOHAWK using Java. In addition to the abstraction-refinement approach, we implemented several supporting tools for MOHAWK. We have a tool for automatically converting a policy in the MOHAWK language to NuSMV specification. Also, we implemented a tool for creating complex ARBAC policies.

We also implemented two well-known static slicing techniques for ARBAC policies [21, 46]. The basic idea behind static slicing is to remove users, roles, *can_assign*, and *can_revoke* rules from the policies that are irrelevant to the safety question. There are two types of static slicing techniques, namely *forward pruning* [21] and *backward pruning* [21,46]. We implemented these techniques and analyzed the effect of static slicing on the ARBAC policies used in our experiments. We found that these techniques are not effective for realistic policies (Section 5.1).

4. SOURCES OF COMPLEXITY FOR ERROR FINDING

In this section, we describe the aspects that make error finding 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 [21, 39, 46] on the verification problem alludes to some of these sources of complexity. Finally, we explain how MOHAWK's abstraction refinement strategy deals 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 [21] 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), 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[1], 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 $c_1 \wedge \ldots \wedge c_n$ 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 [21].

Irrevocable roles. A role $\widehat{r}$ is *irrevocable* if it is not a member of *can_revoke*. 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. [46] 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[2]. Stoller et al. [46] show that the verification problem is fixed

[1] 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 $c_i = r_{i,1} \vee \ldots \vee r_{i,m}$, then we assume that we have the following *can_assign* rules with r_i as the target role: $\langle r_a, r_{i,1} \wedge r_{i-1}, r_i \rangle$, ..., $\langle r_a, r_{i,m} \wedge r_{i-1}, r_i \rangle$.

[2] 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.

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. [39] and Stoller et al. [46] 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 that Horn Satisfiability is in **P**. If this restriction is lifted, then the corresponding satisfiability problem is NP-complete, as we discuss above under "Disjunctions."

We point out, however, that these are not unique to ARBAC. The access matrix scheme due to Harrison et al. [17], for example, has preconditions in its state-change rules as well. Similarly, in the context of RBAC, the work of Crampton and Loizou [9] 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 Abstraction Refinement 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 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 above 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. 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 [46]. Note that although all the competing techniques were developed in the context of verification, they can also be used for the purposes of error finding. We chose NuSMV [33] 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 in finding errors.

We measure the efficiency based on the absolute time taken to find an error. We measure the scalability with respect to the complexity sources (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 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 that 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, we acknowledge that 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,

		Num. of Roles, Rules	MC	BMC	RBAC-PAT		Mohawk
					Forward reachability	Backward reachability	
Case Study		612, 6142	M/O	31s	T/O	Err	2s
Test suite 1 *Poly-time verifiable*	1.	3, 15	0.097s	0.016s	0.625s	0.240s	0.382s
	2.	5, 25	0.050s	0.025s	0.695s	0.281s	0.431s
	3.	20, 100	M/O	0.103s	0.806s	Err	0.733s
	4.	40, 200	M/O	0.110s	0.780s	Err	0.379s
	5.	200, 1000	M/O	0.624s	1.471s	Err	0.477s
	6.	500, 2500	M/O	3.2s	2.177s	Err	0.531s
	7.	4000, 20000	M/O	414s	7.658s	Err	3.138s
	8.	20000, 80000	M/O	M/O	110s	Err	53s
	9.	30000, 120000	M/O	M/O	210s	Err	2m 14s
	10.	40000, 200000	M/O	M/O	6m 16s	Err	4m 32s
Test suite 2 *NP-Complete*	1.	3, 15	0.022s	0.021s	0.513s	0.241s	0.442s
	2.	5, 25	0.064s	0.026s	0.519s	0.252s	0.501s
	3.	20, 100	M/O	0.048s	0.512s	Err	0.436s
	4.	40, 200	M/O	0.122s	0.534s	Err	1.303s
	5.	200, 1000	M/O	0.472s	0.699s	Err	0.504s
	6.	500, 2500	M/O	1.819s	2.414s	Err	0.597s
	7.	4000, 20000	M/O	109s	311s	Err	2.753s
	8.	20000, 80000	M/O	M/O	T/O	Err	40s
	9.	30000, 130000	M/O	M/O	T/O	Err	1m 39s
	10.	40000, 200000	M/O	M/O	T/O	Err	4m 12s
Test suite 3 *PSPACE-Complete*	1.	3, 15	0.030s	0.102s	1.452s	0.665s	0.380s
	2.	5, 25	0.044s	0.033s	1.666s	0.881s	0.431s
	3.	20, 100	M/O	0.056s	1.364s	Err	0.381s
	4.	40, 200	M/O	0.169s	1.476s	Err	0.984s
	5.	200, 1000	M/O	0.972s	2.258s	Err	0.486s
	6.	500, 2500	M/O	2.422s	7.350s	Err	0.487s
	7.	4000, 20000	M/O	109s	511s	Err	2.478s
	8.	20000, 80000	M/O	M/O	T/O	Err	41s
	9.	30000, 130000	M/O	M/O	T/O	Err	2m 57s
	10.	40000, 200000	M/O	M/O	T/O	Err	6m 21s
Simple Policies	1.	12, 19	0.025s	0.022s	0.531s	0.238s	0.300s
	2.	20, 266	0.023s	0.026s	0.559s	0.247s	0.400s
	3.	32, 162	M/O	0.182s	1.556s	0.568s	0.830s

m - minutes MC - NuSMV symbolic model checking T/O - Time out after 60 mins
s - seconds BMC - NuSMV bounded model checking M/O - Memory out
ms - milliseconds RBAC-PAT - Tool from Stoller et al. [13, 46] Err - Segmentation fault

Table 2: Evaluation of model-checking, bounded model-checking, RBAC-PAT, and Mohawk on various benchmarks.

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. The case study involves several branches, and is realistic. We modeled several of the job functions at the branches, and the associated state-change rules. Our case-study has a number of similarities to an earlier case-study [41]. Our ARBAC policy has 612 roles and 6142 *can_assign* and *can_revoke* rules. Separation of privilege properties are of importance in banking [41]; commercial tools exist for checking this property for certain state-only schemes [1, 2]. Hence, we focused on checking for errors that encode such properties. The particular separation of privilege property on which our case study is based expresses the constraint that in any branch, a user may be assigned to at most three of a set of five roles that are sensitive. The violation of this constraint can be checked by appropriately encoding a safety question. Our benchmarks (see Section 5.2) use similar safety questions.

We discussed this policy 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 [42]. Lack of such an enforcement can lead to *toxic pairs* of roles being assigned to the same user.

Our experience with the case study provides two 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 slicing techniques such as those proposed in prior work [21, 46]. Static slicing techniques are effective only 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 3.8). Such a dependency does not exist for the encoding of the separation of privilege property on which Table 2 is based, and for other properties we have investigated in our case-study.

We have tested our case study on all the error-finding tools for our safety question (see Table 2). Mohawk identified an error in 2 seconds, BMC identified the same error in 31 seconds, and the other tools ran out of resources.

5.2 Benchmarks Used

We used two set of policies in our evaluation. Both are based on prior work [13, 21, 41, 46]. 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 the context of verification of ARBAC policies in prior work [13, 46]. 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 that we know occur in practice. As noted earlier, both from our case study and analysis we have concluded that the 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 [21]. 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 non-empty *can_revoke* rules. Error-finding problem is solvable in polynomial time for these policies.
- **Test suite 2:** Policies with mixed conjunctive *can_assign* rules and empty *can_revoke* rules. Error-finding problem is NP-Complete for these policies.
- **Test suite 3:** Policies with mixed conjunctive *can_assign* rules and non-empty *can_revoke* rules. The error-finding problem is PSPACE-Complete for these policies.

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 [13, 39, 46]. The first policy is for a hypothetical hospital, and the second policy is for a hypothetical university. The third policy from [13] is a test case that has mixed preconditions. The first two policies were used in [46] for case studies. The third policy was used in [13], 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 pre-condition 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.

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 error-finding tools consisted of an ARBAC policy and a safety question. We applied the static slicing techniques proposed in prior work [21, 46] 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 slicing 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 below, we mean MOHAWK configured with aggressive abstraction-refinement, unless specified otherwise. For the policies in Table 2, 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 2. Our results indicate that MOHAWK scales better than all the competing tools, irrespective of the complexity of the input policy. BMC scales better than MC, but still runs out of memory for large policies. RBAC-PAT's forward reachability, although slower compared to BMC, is effective for test suite 1, whose policies are verifiable in polynomial time. However, RBAC-PAT's forward reachability times out for large policies in test suites 2 and 3. The forward reachability algorithm in RBAC-PAT can be considered as a specialized model checking algorithm for ARBAC policies, and scales better than symbolic model checking. RBAC-PAT's backward reachability algorithm was faster compared RBAC-PAT's forward algorithm for a few small policies, but gave a segmentation fault for majority of the policies in all the test suites. It is unclear whether this is because of a bug in the implementation or if the tool ran out of memory. MOHAWK scales better compared to all the tools, and is orders of magnitude faster than competing tools for the larger and more complex policies.

MOHAWK (with aggressive abstraction-refinement) is slower compared to MC and BMC for small size policies. 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. However, MOHAWK's abstraction-refinement step can be configured to be less aggressive for such policies. We ran MOHAWK with less aggressive abstraction-refinement for the smaller policies and got performance comparable to BMC.

MOHAWK is very efficient in finding errors in all the three test suites, although each of them belong to a different 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.

5.4.2 Results on Simple Policies

The results of our evaluation on simple policies are summarized in Table 2. The first and second policies did not satisfy separate administration restriction, so we removed

can_assign roles 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 absolute 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 absolute 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

Counter-example guided abstraction refinement was originally developed in the context of model checking [6]. Since then the basic idea has been adapted in different ways in the context of bounded model-checking for hardware verification [27] and program analysis [4] 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 [12]. To the best of our knowledge, MOHAWK is the first tool to adapt the paradigm of abstraction-refinement for finding errors in access-control policies.

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

The work that falls in state-only considers only a given state, and verification of properties within that state. Examples of work that fall in this category include those of Jha et al. [22, 23], Hughes et al. [20], Hu et al. [18], Martin and Xie [31], Rao et al. [34], Kolovski [26], Zhao et al. [48], and Fisler et al. [11]. It is conceivable that our approach can be used in state-only contexts. Indeed, the work of Martin and Xie [31] considers testing of XACML policies by introducing what they call faults that are used to simulate common errors in authoring such policies. However, in this paper, we focus on access control systems that are characterized as state-change systems. Consequently, we focus on work that considers verification of such systems.

Plain model-checking approach has also been proposed for some state-change schemes [47]. 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. [17] 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 [16]. Jones et al. [24] 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) [3] and the Typed Access Matrix model [37]. Budd [5] and Motwani et al. [32] studied grammatical protection systems. Soshi et al. [45] studied safety analysis in Dynamic-Typed Access Matrix model. These models all have subcases where safety is decidable. Solworth and Sloan [44] 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 paper 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 [29] were the first to consider security analysis in the context of ARBAC. Jha et al. [21] 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. [46] 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 [13].

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. [14] 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 paper.

7. CONCLUSION

We presented an abstraction-refinement based technique, and its implementation, the MOHAWK tool, for finding errors in 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 against current state-of-the-art tools for policy analysis. Our experiments show that in comparison with the current tools, MOHAWK scales very well with the complexity of policies and is also orders of magnitude faster. Analysis tools such as MOHAWK enable policy administrators to quickly analyze policies prior to deployment, thereby increasing the assurance of the system.

Acknowledgements

We thank Mikhail Gofman, Scott Stoller, C. R. Ramakrishnan, and Ping Yang for providing access to the RBAC-PAT tool and their experimental data.

8. REFERENCES

[1] Aveska. http://www.aveksa.com/solutions/access-control-automation.cfm.

[2] SailPoint. http://www.sailpoint.com/product/compliance-manager/policy-enforcement.php.

[3] P. Ammann and R. Sandhu. Safety analysis for the extended schematic protection model. *IEEE Symposium on Security and Privacy*, 1991.

[4] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proc. of the 29th ACM symposium on Principles of programming languages*, New York, NY, USA, 2002. ACM.

[5] T. A. Budd. Safety in grammatical protection systems. *Intl. Journal of Parallel Programming*, 12(6):413–431, 1983.

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

[7] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. The completeness threshold for bounded model checking.

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

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

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

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proc. of the 27th Intl. conference on Software engineering*, 2005. ACM.

[12] V. Ganesh and D. L. Dill. A decision procedure for bitvectors and arrays. In *Computer Aided Verification,LNCS*, 2007.

[13] M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller. Rbac-pat: A policy analysis tool for role based access control. In *Proc. of the 15th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505, Springer-Verlag, 2009.

[14] M. I. Gofman, R. Luo, and P. Yang. User-role reachability analysis of evolving administrative role based access control. In *Proc. of the 15th European conference on Research in computer security*, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] G. S. Graham and P. J. Denning. Protection — principles and practice. In *Proc. of the AFIPS Spring Joint Computer Conference*, volume 40, AFIPS Press, May 1972.

[16] M. A. Harrison and W. L. Ruzzo. Monotonic protection systems. *Foundations of Secure Computation*, 1978.

[17] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On protection in operating systems. In *SOSP '75: Proc. of the fifth ACM symposium on Operating systems principles*, 1975. ACM.

[18] H. Hu and G. Ahn. Enabling verification and conformance testing for access control model. In *SACMAT '08: Proc. of the 13th ACM symposium on Access control models and technologies*, New York, NY, USA, 2008. ACM.

[19] V. C. Hu, D. R. Kuhn, and T. Xie. Property verification for generic access control models. In *EUC '08: Proc. of the 2008 IEEE/IFIP Intl. Conference on Embedded and Ubiquitous Computing*, 2008. IEEE Computer Society.

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

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

[22] S. Jha and T. W. Reps. Model Checking SPKI/SDSI. *Journal of Computer Security*, 12(3–4):317–353, 2004.

[23] S. Jha, S. Schwoon, H. Wang, and T. Reps. Weighted Pushdown Systems and Trust-Management Systems. In *Proc. of TACAS*, New York, NY, USA, 2006. Springer-Verlag.

[24] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *SFCS '76: Proc. of the 17th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1976. IEEE Computer Society.

[25] A. Kern. Advanced features for enterprise-wide role-based access control. In *ACSAC '02: Proc. of the 18th Annual Computer Security Applications Conference*, Washington, DC, USA, 2002. IEEE Computer Society.

[26] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *WWW '07: Proc. of the 16th Intl. conference on World Wide Web*, 2007. ACM.

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

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

[29] N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *SACMAT '04: Proc. of the ninth ACM symposium on Access control models and technologies*, New York, NY, USA, 2004. ACM.

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

[31] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *WWW '07: Proc. of the 16th Intl. conference on World Wide Web*, 2007. ACM.

[32] R. Motwani, R. Panigrahy, V. Saraswat, and S. Ventkatasubramanian. On the decidability of accessibility problems (extended abstract). In *STOC '00: Proc. of the thirty-second annual ACM symposium on Theory of computing*, New York, NY, USA, 2000. ACM.

[33] NuSMV. `http://nusmv.irst.itc.it/`.

[34] P. Rao, D. Lin, and E. Bertino. XACML function annotations. In *POLICY '07: Proc. of the Eighth IEEE Intl. Workshop on Policies for Distributed Systems and Networks*, Washington, DC, USA, 2007. IEEE Computer Society.

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

[36] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[37] R. S. Sandhu. The typed access matrix model. In *Proc. IEEE Symposium on Research in Security and Privacy*, 1992.

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

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

[40] A. Sasturkar, P. Yang, S. D. Stoller, and C. Ramakrishnan. Policy analysis for administrative role based access control. Technical report, Stony Brook University, 2006.

[41] A. Schaad, J. Moffett, and J. Jacob. The role-based access control system of a european bank: a case study and discussion. In *SACMAT '01: Proc. of the sixth ACM symposium on Access control models and technologies*, New York, NY, USA, 2001. ACM.

[42] Security Architect of a Leading Bank. Personal communication, 2010.

[43] K. Sohr, M. Drouineaud, G.-J. Ahn, and M. Gogolla. Analyzing and managing role-based access control policies. *IEEE Transactions on Knowledge and Data Engineering*, 20:924–939, 2008.

[44] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable safety properties. *IEEE Symposium on Security and Privacy*, 2004.

[45] M. Soshi. Safety analysis of the dynamic-typed access matrix model. In *Computer Security - ESORICS 2000*, LNCS, Springer Berlin / Heidelberg, 2000.

[46] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS '07: Proc. of the 14th ACM conference on Computer and communications security*, 2007. ACM.

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

[48] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and reasoning on rbac: A description logic approach. In *ICTAC'05: Proc. of the 2nd Intl. Colloquium on Theoretical Aspects of Computing*, LNCS, Springer, 2005.