Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2010-022 May 5, 2010

Automatic Error Finding in Access-Control Policies

Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin C. Rinard, and Steve Chapin

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Automatic Error Finding in Access-Control Policies

Karthick Jayaraman
Syracuse University
kjayaram@syr.edu

Vijay Ganesh
MIT
vganesh@csail.mit.edu

Mahesh Tripunitara
University of Waterloo
tripunit@uwaterloo.ca

Martin Rinard
MIT
rinard@lcs.mit.edu

Steve Chapin
Syracuse University
chapin@syr.edu

ABSTRACT

Access-control policies are a key infrastructural technology for computer security. However, a significant problem is that system administrators need to be able to automatically verify whether their policies capture the intended security goals. To address this important problem, researchers have proposed many automated verification techniques. Despite considerable progress in verification techniques, scalability is still a significant issue. Hence, in this paper we propose that error finding complements verification, and is a fruitful way of checking whether or not access control policies implement the security intent of system administrators. Error finding is more scalable (at the cost of completeness), and allows for the use of a wider variety of techniques.

In this paper, we describe an abstraction-refinement based technique and its implementation, the MOHAWK tool, aimed at finding errors in ARBAC access-control policies. The key insight behind our abstraction-refinement technique is that it is more efficient to look for errors in an abstract policy (with successive refinements, if necessary) than its complete counterpart.

MOHAWK accepts as input an access-control policy and a safety question. If MOHAWK finds an error in the input policy, it terminates with a sequence of actions that cause the error. We provide an extensive comparison of MOHAWK with the current state-of-the-art analysis tools. We show that MOHAWK scales very well as the size and complexity of the input policies increase, and is orders of magnitude faster than competing tools. The MOHAWK tool is open source and available from the Google Code website: `http://code.google.com/p/mohawk/`

1. INTRODUCTION

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 [38]) that have considerable expressive power, and can be used to specify complex policies.

However, a significant problem is that system administrators need to be able to verify whether the policies managed by them capture their security intent or not. Manual inspection of such policies is not sufficient. Access-control policies for reasonably large systems are simply too complex for manual inspection to be effective. Hence, there is a clear need for automated tools that can *verify the correctness* of access-control policies.

Automated analysis and verification of access-control policies is an active area of research [11,15,19–23,28,29,31,41,43,46,49,50]. Model checking [8] has emerged as a promising, automated approach [15, 23, 46] to the verification problem. 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. The user would like to use the model checker to check whether the property always holds for all program behaviors (or all possible authorizations allowed by the policy). However, the model-checking problem for access-control policies is intractable in general (PSPACE-complete [23, 41]), and scalability for practical verification tools remains a significant issue despite considerable progress (see Section 5).

In this paper, we propose a different approach to the problem of checking whether or not access-control policies implement the security intent of the system administrator. Instead of focusing on complete verification of such policies, we propose that it is fruitful to look for errors in them (*the error-finding problem*).

Error finding complements verification in many ways. Error-finding tools can be used to find shallow, localized errors, while verification can be used for checking global properties. Automatic error-finding tools can also be used during the design and implementation of policies to help prune errors early, and refine the security intent of the system administrators, thus making subsequent verification work easier. Finally, techniques developed in the context of verification can often be adapted for error-finding.

In the context of access-control policies, an error constitutes allowing an unauthorized user to access a resource. The question posed by the system administrator remains the same irrespective of whether complete verification or error finding is the goal. The difference is in terms of the underlying technique used and the completeness of the technique.

There are several advantages in changing one's point of view from verification to finding errors:

- First, finding errors is in general more tractable than verifying correctness, at the cost of completeness.
- Second, while it is true that verification tools can also be used for the error-finding problem, changing the viewpoint from verification to error finding allows us to leverage a wider range of techniques. This has been observed in software reliability research, where bounded model checking [6] and incomplete/unsound program analysis have had a significant impact in automatically exposing bugs that had been impossible to find otherwise [47, 48].
- Third, automatic error-finding tools for access-control policies can serve a purpose similar to automated bug finding tools [5, 13, 14] for computer programs. Just as computer programmers use tests and automatic test generation tools to not only reveal errors in their programs, but also to guide the development process, error-finding tools can similarly be

used by system administrators to not only find errors in their access control policies, but also to refine their security intent.

Abstraction Refinement. The basic intuition behind abstraction-refinement strategies is that it is more efficient to look for errors in an abstract version of the input policy (with successive refinements, as necessary) compared to the full input policy. A desirable property of such an abstraction is that if an error is found in the abstract version, it must be an error in the original input policy.

Abstraction refinement, a well-known paradigm in verification and program analysis literature [7], has been successful in the context of bug finding and verification in computer programs [2, 3, 7, 17,35]. We show here that the abstraction-refinement paradigm can be successfully adapted to the context of finding errors in ARBAC policies.

Key Insights. There are two key insights behind our abstraction-refinement technique as applied to access-control policies. First, it is often the case that, during analysis, we can *abstract* most of the access-control policy, while preserving errors. The result is that verification or error-finding on the abstract policy is much faster compared to the complete original policy. Second, most errors tend to be few *refinements* away from the initial state. Therefore, if parts of the policy need to be *refined*, it can be done in very few refinement steps subsequently. As we show in our experimental evaluation (Section 5), model checking, bounded model-checking and other analysis techniques are not able to leverage the above-mentioned aspects of access-control policies.

1.1 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 policies increase.

   Our technique can also be used for error finding in frameworks other than ARBAC. The reason is that 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. [19].

   Although we focus on error finding in this paper, our abstraction-refinement technique can be used in conjunction with model checking to perform verification. The combination of abstraction-refinement technique and bounded model checker (with a pre-determined bound) that we present in this paper is incomplete, i.e., it may not find all the errors in buggy input policies.

2. An implementation of our technique in a tool called MOHAWK, and an evaluation of the tool using a variety of complex access-control policies obtained from previous literature, as well as ones developed by us. The MOHAWK tool is available from the Google Code website: `http://code.google.com/p/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 pre-determined bound.

3. We provide a detailed comparison of MOHAWK against NuSMV [36], a well known model checking and bounded model checking tool, and RBAC-PAT [15, 46], a tool specifically designed for analyzing ARBAC policies. Unlike these other approaches, MOHAWK scales very well as the size and complexity of the input policies increase.

   We also provide a policy-generation tool that automatically generates complex policies, and a benchmark suite comprising large, complex, and expressive ARBAC policies that capture all the known sources of complexity. The tool and the benchmark suite can be downloaded from the MOHAWK website.

4. The abstraction-refinement implementation in MOHAWK is *configurable*, i.e., the user can control the aggressiveness of the abstraction and refinement steps using a run-time option. We show that for small, relatively simple access-control policies it is better to abstract less, while for larger, more complex policies it is better to abstract more aggressively.

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 how MOHAWK deals with the sources of complexity from the standpoint of error finding. 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 [16,19] and Role-Based Access Control (RBAC) [10, 40]. 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 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.

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: RBAC state and state-change rule for a simple ARBAC policy

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 are used to manage the accesses of tens of thousands of users to millions of 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 [38] 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 [15, 31, 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 [27]. 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 [38] 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 [26].

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 as the one from above has received considerable attention in the literature [23, 31, 41, 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 can be reduced to the basic safety question. This observation has been made before [23, 46].

We consider only such basic safety questions. 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

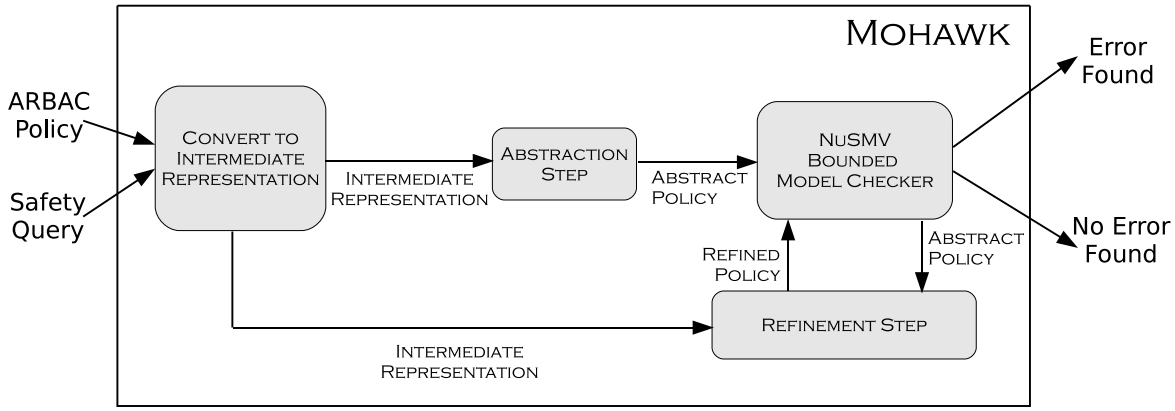

Figure 2: Architecture of MOHAWK illustrating abstraction-refinement-based technique for error finding in ARBAC policies.

$\psi = \langle can\_assign, can\_revoke \rangle$.

The separate administration restriction

Previous work [15,41,46] considers what is called the separate administration restriction in the context of such error-finding problems. In this context, the question that is asked is whether the roles we consider in an error finding instance include roles that pertain to administration. We recall our discussions from Section 2.1 in which we mention that one of the components of *can_assign* and *can_revoke* we omit is an administrative role, r_a.

If our state changes allow for memberships in r_a to change, then a consequence is that r_a may, in some state, have no members. This would result in a change to the state-change rules, as any rule that has r_a as its administrative role cannot fire.

The original specification on ARBAC [38] clearly specifies that administrative roles are not administered by the same rules as "regular" roles. We adopt this assumption. In other words, 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.

The role hierarchy. We assume that an RBAC state has no role hierarchy. The reason is that there is a straightforward 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 [41].

3. ARCHITECTURE OF MOHAWK

In the following, we describe MOHAWK's architecture (§3.2-§3.5), and illustrate our approach using an example (§3.6). 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 on *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 input policy and safety query into an intermediate representation.
- **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 (Note that this does not necessarily imply there are no errors in the input policy). MOHAWK may execute the verify-refine loop multiple times, until either MOHAWK identifies an error or no further refinements are possible.

3.1 Abstraction Refinement in Mohawk

Abstraction refinement in MOHAWK is goal oriented and driven by the safety question. MOHAWK creates a priority queue of roles based on their *Related-by-Assignment* relationship with the query role (See Section 3.2). MOHAWK uses this stratification to incrementally add roles, *can_assign*, and *can_revoke* rules to the abstract policy for verification. The resulting strategy helps MOHAWK efficiently identify shallow errors without burdening the model checker with extraneous input.

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 3.6, 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 atleast 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 modeled 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 analysis 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 analysis the *can_assign* rules that have the role being analyzed as their target role. All the roles involved 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. 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. 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.

3.3 Abstraction Step

In the initial abstraction step, MOHAWK constructs an abstract policy $\langle U', R', UA', \textit{can\_assign}', \textit{can\_revoke}' \rangle$, 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 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 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$\wedge\neg$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 1 contains the abstraction-refinement steps for the example policy in Figure 3. Figure 4 contains the tree for 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

```
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

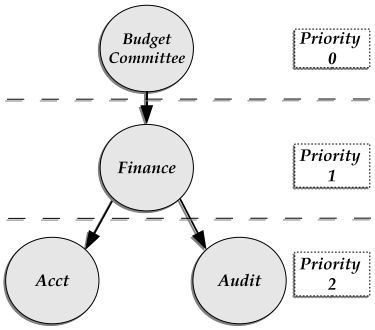

Figure 4: Related-by-assignment (RBA) relationship between roles with respect to BudgetCommittee.

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, BudgetCommittee⟩		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

involve roles other than roles 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* rule. Bob 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* roles 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.

3.7 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 [23, 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* [23] and *backward pruning* [23, 46]. Both techniques retain only the admin users and the user involved in the safety question. *Forward pruning* removes *can_assign* rules that can never be applied in the current UA. *Backward pruning* computes two sets of roles, namely R_{po} and R_{ne}. R_{po} is the set of roles on which r positively depends, and R_{ne} is the set of roles on which r negatively depends. The sliced policy retains only the assignment rules associates with roles in R_{po}, and revocation rules associated with roles in R_{ne}.

We implemented these techniques and analyzed the effect of static slicing on the ARBAC policies used in our experiments. We found that static slicing helps, but does not scale with the size and complexity of policies.

4. SOURCES OF COMPLEXITY FOR ERROR FINDING

In this section, we discuss how we deal with sources of complexity in ARBAC polices from the standpoint of error-finding. These sources of complexity are discussed in previous work in the context of verification [23, 41, 46]. There are four such sources of complexity: (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 that 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 only in *can_assign* rules. In Appendix A, we discuss in more detail why these introduce complexity in the verification and error-finding of ARBAC policies.

We point out that in the worst-case, the change in perspective from verification to error finding is inconsequential. Indeed, we may end up doing more work as we go through additional abstraction-refinement steps. However, we argue that in practice, our approach is effective.

One aspect from our approach that assuages the compexity is that we are goal-oriented in our abstraction-refinement algorithm (see Section 3). Our stratification of roles begins with the role from the safety instance. 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) (see Appendix A). 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

The objective of our experiments was to ascertain the effectiveness of the abstraction-refinement technique implemented in MOHAWK, and compare it with existing state-of-the-art methods for finding errors in access-control policy, namely symbolic model checking, bounded model checking, and RBAC-PAT [46]. We chose NuSMV [36] as the reference implementation for both symbolic and bounded model checking. Note that although all the competing techniques were developed in the verification context, they can also be used for the purposes of error finding. In the following, we will use the terms MC and BMC to refer to NuSMV's symbolic model checker and NuSMV's bounded model checker respectively. All the experiments were conducted on Macbook Pro laptop with Intel Core 2 Duo 2.4GHz processor and 4GB RAM.

5.1 Benchmarks Used

We have used two sets of policies to evaluate the various tools. Both are based on prior work [15, 23, 33, 42, 46]. The first set has not been used previously; we built it based on data from RBAC deployments [42], and benchmarks used in role mining [33]. The second has been used in the context of verification of ARBAC policies in prior work [15, 46] (These policies are simpler compared to the policies in the first set).

Complex Policies (Set 1): We built three test suites of policies that were designed to factor in the known sources of complexity, be reflective of the sizes of policies we expect to see in real deployments, and exercise the full expressiveness of ARBAC. As explained in section 4 and Appendix A, the four known sources of complexity in ARBAC policies are disjunctions in the *can_assign* rules, number of positive preconditions, number of mixed preconditions, and number of irrevocable roles. Jha et al. [23] provides the restrictions under which safety analysis for ARBAC, which is the error-finding problem in our context, is PSPACE-Complete, NP-Complete, and solvable in polynomial time. Accordingly, we created three test suites:

- **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. Error-finding problem is PSPACE-Complete for these policies.

The preconditions for the *can_assign* rules were randomly created. Unlike the policies in Set 2 obtained from [15,46], these policies may have more than one positive precondition per *can_assign* rule.

We implemented a policy-generation tool that accepts the following inputs, namely number of users, number of roles, number of *can_assign* rules, number of *can_revoke* rules, type of can_assign rules, and type of UA, and dumps an ARBAC policy into an output file. The generated policies have an administrative role and an administrative user assigned to administrative role, by default. The administrative role is used in all the *can_assign* rules. The *can_revoke* rules can be chosen to be either positive conjunctive or mixed. The initial UA can be either empty or randomly assigned. For each policy, the reachability of a role by a user is random. Also, for each policy, the tool creates a safety question at random. The safety question asks whether a role is reachable by a user.

In these policies we are hoping to capture the typical security intent of a system administrator, namely, a subset of users may not reach a subset of roles. Since the policies are randomly built, it is possible that some randomly generated rules allow for a user to eventually reach a designated role, i.e., the policy may have an error. It is these kind of policies that interest us.

Simple Policies (Set 2): We used three ARBAC policies used in previous work for the evaluation of the verification tool, RBAC-PAT [15, 41, 46]. The first policy is an ARBAC policy for a hypothetical hospital, while the second policy is for a hypothetical university. The third policy from [15] is a test case having mixed preconditions. The first two policies were used in [46] for case studies. The third policy was used in [15], and a complete state-space exploration took 8.6 hours in RBAC-PAT. An important restriction in

	Num. of Roles, Rules	MC	BMC	RBAC-PAT		MOHAWK
				Forward reachability	Backward reachability	
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

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. [15,46] Err - Segmentation fault

Table 2: Evaluation of error-finding tools on positive conjunctive ARBAC policies. (Test suite 1)

	Number of Roles, Rules	MC	BMC	RBAC-PAT		MOHAWK
				Forward reachability	Backward reachability	
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

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. [15,46] Err - Segmentation fault

Table 3: Evaluation of error-finding tools on mixed conjunctive ARBAC policies with no *can_revoke* rules. (Test suite 2)

these policies is that they have atmost one positive pre-condition per *can_assign* rule. As we will explain later, answering the safety question for these policies was fairly easy for all the tools.

5.2 Experimental Methodology

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 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 also remove the users who are not connected to the safety question.

5.3 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.

5.3.1 Results on Complex Policies

The results of our evaluation on complex policies (Set 1) are tabulated in Table 2 (Test suite 1), Table 3 (Test suite 2), and Table 4 (Test suite 3). Static slicing was not effective for the policies in our test suite. This is because the randomly built preconditions for the *can_assign* rules may relate a majority of the roles in the policies to one another.

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 policy #8 in all the test suites. 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 policy #8 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 policies 1 and 2 in all the test suites, but gave a segmentation fault for all policies 3-8 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 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 maintainability and extensibility of the tool.

	Num. of Roles, Rules	MC	BMC	RBAC-PAT		MOHAWK
				Forward reachability	Backward reachability	
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

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. [15,46] Err - Segmentation fault

Table 4: Evaluation of error-finding tools on mixed conjunctive ARBAC policies. (Test suite 3)

	Num. of Roles, Rules	MC	BMC	RBAC-PAT		MOHAWK
				Forward reachability	Backward reachability	
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. [15,46] Err - Segmentation fault

Table 5: Evaluation of error-finding tools on ARBAC policies obtained from previous literature.

MOHAWK, although slower compared to MC and BMC for small size policies, can be reliably used to analyze policies of very large sizes. Moreover, MOHAWK's abstraction-refinement step can be configured based on the policy size and complexity. We ran MOHAWK with less aggressive abstraction-refinement for the smaller policies and got performance comparable to BMC. Furthermore, since our technique is not tied to specific model-checking algorithms, it can used in conjunction with other algorithms such as RBAC-PAT's forward reachability.

5.3.2 Results on Simple Policies

We first analyzed three ARBAC policies that were used in prior work [15,46], and the results are summarized in Table 5. 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 (Note that the degree of abstraction-refinement can be configured by the user) is faster compared to RBAC-PAT's forward reachability, but slower compared to BMC and RBAC-PAT's backward reachability. However, the absolute slow down in each case is less than a second and is imperceptible to the user. The reason for the slowdown experienced by MOHAWK with aggressive abstraction-refinement for these policies is that these policies are so small that BMC can analyze them easily, while 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. 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.

We also ran MOHAWK with different configuration of abstraction-refinement on these policies. It turns out that the less aggressive the abstraction-refinement step, the faster MOHAWK gets for these small policies. In the limit it has similar performance as BMC.

6. RELATED WORK

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. [24, 25], Hughes et al. [22], Hu et al. [20], Martin and Xie [32], Rao et al. [37], Kolovski [28], Zhao et al. [50], and Fisler et al. [11].

Model checking has been proposed in some of the state-only contexts. For example, the work of Jha et al. [24, 25] proposes modeling the distributed authorization framework SPKI/SDSI using push-down systems. The intent is to leverage efficient model-checking algorithms for authorization checks. This is different from the state-reachability problem that we recast as error-finding in this paper. Furthermore, the problems considered there are in $\mathbf{P}$.

It is conceivable that our approach can be used in state-only contexts. Indeed, the work of Martin and Xie [32] 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 [49]. 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. [19] 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 [18]. Jones et al. [26] 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) [1] and the Typed Access Matrix model [39]. Budd [4] and Motwani et al. [34] 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 [30] were the first to consider security analysis in the context of ARBAC. Jha et al. [23] 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 [15].

Comparison to RBAC-PAT. RBAC-PAT contains two algorithms for analysing 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.

Abstraction Refinement and Bugfinding in Programs. The idea of counter-example guided abstraction refinement was originally developed in the context of model checking [7]. Since then the basic idea has been adapted in different ways in the context of bounded model-checking and program analysis to find errors in computer programs [3]. 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.

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. The abstraction-refinement technique in MOHAWK is configurable, thereby enabling users to adapt the technique based on the nature of the input policies. 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 size 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] P. Ammann and R. Sandhu. Safety analysis for the extended schematic protection model. *Security and Privacy, IEEE Symposium on*, 0:87, 1991.

[2] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *16th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 457–461, Boston, MA, USA, 2004. Springer.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

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

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

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

[7] 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.

[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: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[12] V. Ganesh and D. L. Dill. A decision procedure for bitvectors and arrays. In *Computer Aided Verification, number 4590 in LNCS*, 2007.

[13] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[15] 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 *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505, pages 46–49. Springer-Verlag, 2009.

[16] G. S. Graham and P. J. Denning. Protection — principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. AFIPS Press, May 1972.

[17] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 443–458. Springer, 2008.

[18] M. A. Harrison and W. L. Ruzzo. Monotonic protection systems. *Foundations of Secure Computation*, pages 461–471, 1978.

[19] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On protection in operating systems. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 14–24, New York, NY, USA, 1975. ACM.

[20] H. Hu and G. Ahn. Enabling verification and conformance testing for access control model. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 195–204, New York, NY, USA, 2008. ACM.

[21] V. C. Hu, D. R. Kuhn, and T. Xie. Property verification for generic access control models. In *EUC '08: Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 243–250, Washington, DC, USA, 2008. IEEE Computer Society.

[22] 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.

[23] 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.

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

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

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

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

[28] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 677–686, New York, NY, USA, 2007. ACM.

[29] 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.

[30] N. Li and M. V. Tripunitara. Security analysis in role-based access control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 126–135, New York, NY, USA, 2004. ACM.

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

[32] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 667–676, New York, NY, USA, 2007. ACM.

[33] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating role mining algorithms. In *SACMAT '09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 95–104, New York, NY, USA, 2009. ACM.

[34] R. Motwani, R. Panigrahy, V. Saraswat, and S. Ventkatasubramanian. On the decidability of accessibility problems (extended abstract). In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 306–315, New York, NY, USA, 2000. ACM.

[35] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[37] P. Rao, D. Lin, and E. Bertino. XACML function annotations. In *POLICY '07: Proceedings of the Eighth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 178–182, Washington, DC, USA, 2007. IEEE Computer Society.

[38] 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.

[39] R. S. Sandhu. The typed access matrix model. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 122–136, 1992.

[40] 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.

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

[42] 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: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 3–9, New York, NY, USA, 2001. ACM.

[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. *Security and Privacy, IEEE Symposium on*, 0:56, 2004.

[45] M. Soshi. Safety analysis of the dynamic-typed access matrix model. In *Computer Security - ESORICS 2000*, volume 1895 of *Lecture Notes in Computer Science*, pages 106–121. 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: Proceedings of the 14th ACM conference on Computer and communications security*, pages 445–455, New York, NY, USA, 2007. ACM.

[47] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *17th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 139–143. Springer, 2005.

[48] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *SIGPLAN Not.*, 40(1):351–363, 2005.

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

[50] C. Zhao, N. Heilili, S. Liu, and Z. Lin. Representation and reasoning on rbac: A description logic approach. In *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, volume 3722 of *Lecture Notes in*

Computer Science, pages 381–393. Springer, 2005.

APPENDIX

A. SOURCES OF COMPLEXITY

In this appendix, we identify 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 [23, 41, 46] on the verification problem alludes to some of these sources of complexity.

There are three broad aspects that can bring complexity to verification in access control systems. (1) The syntax for the state, (2) the state-change rules, and, (3) the verification question of interest. In our context, (1) is not a source of complexity. Indeed, access control schemes are often designed so that the syntax for the state lends itself to efficient access enforcement. In RBAC, for example, enforcement can be performed in worst-case time linear in the size of the state.

The potential source (3) is not a source of complexity either, 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 [23] 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.

Our main source of complexity, then, are the state-change rules. The component within the state-change rules that is relevant is the precondition. We now discuss the specific aspects of preconditions in ARBAC that are the sources of complexity.

We point out, however, that these are not unique to ARBAC. The access matrix scheme due to Harrison et al. [19], 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.

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 [23].

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] prove 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. [41] 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 $\mathbf{P}$. If this restriction is lifted, then the corresponding satisfiability problem is NP-complete, as we discuss above under "Disjunctions."

[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.