Cree: a Performant Tool for Safety Analysis of Administrative Temporal Role-Based Access Control (ATRBAC) Policies

Jonathan Shahen, Jianwei Niu, and Mahesh Tripunitara

Abstract—Access control deals with the roles and privileges to which a user is authorized, and is an important aspect of the security of a system. As enterprise access control systems need to scale to several users, roles and privileges, it is common for access control models to support delegation: a trusted security administrator is able to give semi-trusted users the ability to change portions of the authorization state. With delegation comes the danger that semi-trusted users, perhaps in collusion, may effect a state that violates enterprise policy, which in turn results in the problem called safety analysis, which is regarded as a fundamental and technically challenging problem in access control. Safety analysis is used by a trusted security administrator to answer "what if" questions before she grants privileges to a semi-trusted user. Safety analysis has been studied for various access control schemes in the literature; we address safety analysis in the context of Administrative Temporal Role-Based Access Control (ATRBAC), an administrative model for TRBAC, which is an extension to the traditional RBAC. ATRBAC has new features, which introduce new technical challenges for safety analysis: (i) a time-dimension: two new components in each administrative rule that specify in which time periods an administrative action may be effected, and a user is authorized to a role, and, (ii) two new kinds of rules for whether a role is enabled for administrative action. We propose a software tool, which we call Cree, for safety analysis of ATRBAC policies. In Cree we reduce ATRBAC-Safety to model checking and use an off-the-shelf model checker, NuSMV. The foundation for Cree is the observation from our prior work that ATRBAC safety is **PSPACE**-complete. Along with an efficient reduction to model checking, we include in Cree four techniques to further improve performance: Polynomial Time Solving when possible, Forward and Backwards Pruning, Abstraction Refinement, and Bound Estimation. These are inspired by prior work, but our algorithms are different in that they address the new challenges that ATRBAC introduces. We discuss our design of Cree, and the results of a thorough empirical assessment across our approach, and five other prior tools for ATRBAC safety. Our results suggest that there are input classes for which Cree outperforms existing tools, and for the remainder, Cree's performance is no worse. We have made Cree available as open-source for public download.

Index Terms—Safety Analysis, Access Control, Verification, Computer Security, Computational Complexity.

1 Introduction

Access control deals with whether a principal may exercise a privilege on a resource; e.g. whether the user Alice is allowed to exercise the 'read' privilege on a file. It is an important aspect of the security of a system. Whether an attempted access is permitted is customarily specified in an access control policy. Such a policy may change over time; for example, Alice may, at some later time, lose the 'read' privilege to a file that she possesses.

Effecting and intuiting the consequences of changes to an access control policy is called administration. An aspect of administration is delegation, with which a trusted administrator empowers another principal to change the policy in limited ways. Delegation is used so administrative efficiency can scale with the size of an access control system. For example, a trusted administrator may delegate to a user Bob the ability to add users within his team, to a project Bob owns. Thus, Bob is able to control which members of his team can access the resources of a particular project without requiring assistance from a trusted administrator.

With delegation arises the need for safety analysis, which has been recognized as a fundamental problem in access control since the work of Harrison et al. [1]. Safety analysis asks, in the presence of delegation, whether some partially trusted users may effect changes to the authorization policy in a manner that a desirable security property is violated.

Safety analysis has been addressed for various access control schemes in the literature. Our focus is safety analysis in the context of Administrative Temporal Role-Based Access Control (ATRBAC) [2]. ATRBAC is an administrative scheme for Temporal Role-Based Access Control (TRBAC). TRBAC is Role-Based Access Control (RBAC) with temporal-extensions. In RBAC, rather than assigning a user directly to a permission, we adopt the indirection of a role. A user is authorized to a set of roles, and each role is authorized to a set permissions. The temporal extensions that TRBAC adds to RBAC constrain the time intervals that (i) a user may exercise a privilege, and, (ii) a role may be active. In addition, ATRBAC constrains the time intervals that (iii) an administrative action can change the authorization policy. (See Section 2 for a more details.)

An instance of the safety analysis decision problem comprises the following three inputs. A concrete example of the scheme we consider, ATRBAC, is in Section 2.

(1) **Start State** – an instance of an access control policy, i.e., which users currently have what privileges or roles.
(2) **State Change Rules** – set of administrative rules by

• *J. Shahen and M. Tripunitara are with the Department of Electrical and Computer Engineering, University of Waterloo, Canada. J. Niu is with the Department of Computer Science, University of Texas at San Antonio. E-mail: {jmshahen, tripunit}@uwaterloo.ca, jianwei.niu@utsa.edu*

Manuscript received x; revised y.

which a policy can change, e.g., rule: ⟨Bob may grant users membership to Bob's Project if that user is a member of Bob's team⟩.

(3) **Security Query** – a statement that can judge if the system is secure or insecure; typically whether a particular user possesses a particular privilege. Examples: "Can Alice get write access to Payroll", "Can Bob adopt the role of Administrator", and "Any user have read access to New Project and write access to Public Repository."

An instance of safety analysis is 'TRUE' if the security query can never become true, i.e., the system is indeed safe, and 'FALSE' otherwise. The utility of such analysis is that a trusted administrator can assess whether the current, or a prospective, set of authorizations and delegations may lead to a violation of a desirable security property. She can then modify it if indeed such a violation may result.

ATRBAC-Safety is an important and non-trivial problem. We discuss this in the context of a concrete example for ATRBAC in Section 2 after we have presented the details of ATRBAC and safety-analysis in its context (see the paragraph titled, "The example, revisited" towards the end of that section). Here we provide a broader discussion.

Jones [3] discusses why safety analysis can be a technical challenge: it is in the difference in the manner in which administrative rules and security properties are specified. Administrative rules are "phrased in a procedural form", while security properties are "formulated as predicates." "Though procedural definitions make individual system state transitions easy to understand and to implement, they combine to form a system that exhibits complex behaviour. It is difficult to intuit and to express the behaviour of a procedurally defined system." [3] As in the case of other access control schemes, security queries in ATRBAC-Safety are in predicate form and the rules are in procedural form.

In addition to this difference in the manner in which rules and properties are specified, there is the issue of scale. Enterprise policies can be large and thereby preclude manual safety analysis. For example, prior work [4] discusses an ARBAC policy, i.e., without the additional features of ATRBAC, of a financial company which comprises 1363 roles and 8885 rules. Such a policy is likely too large for manual safety analysis. We seek an automated approach.

Given the above discussions on the technical challenges that underlie safety analysis in general, one may ask what makes safety analysis in ATRBAC, the scheme on which we focus, a particular challenge. That is, what technical challenges does our work specifically address that prior work on safety analysis, for example, in the context of ARBAC, does not? ATRBAC introduces new syntactic constructs with associated semantics. We need safety analysis to "catch up" with these new constructs. As an example from the literature, Harrison et al. [1] originally proposed an administrative scheme for the access matrix model, and safety analysis results for it. Their work establishes that for their scheme, safety is undecidable. Sandhu [5] syntactically extends their work with the Monotonic Typed Access Matrix (MTAM) model. Safety analysis for MTAM, as it turns out, is decidable. The new features in ATRBAC are: (i) an administrative rule that specifies time periods during which (a) an administrative action may be effected, and, (b) a user is authorized to a role, and, (ii) two new kinds of administrative rules that are used to enable and disable administrative roles. (See Section 2 for a comprehensive description of ATRBAC.) It is unclear as to the manner in which these new features impact safety analysis. Our prior conference paper [6] establishes that the problem remains in **PSPACE**, and proposes an approach based on reduction to safety analysis in ARBAC. In this work, we leverage the "in **PSPACE**" result in a different manner: we reduce ATRBAC-safety directly to model-checking. As our work establishes (see Section 6), and as we hypothesized at the start of this work, this approach is more efficient in practice. There are a number of technical innovations we have devised in carrying out this reduction, and complementing it with optimizations (see Section 5.) While these are inspired by prior work, they need to be custom to ATRBAC. Nonetheless, our work in turn may be directly useful to future schemes, or at the minimum, provide inspiration on general directions as prior work has for this work.

Layout The remainder of this paper is organized as follows. In the next section, we formalize and discuss the problem that this work addresses, ATRBAC-safety. In Section 3, we discuss relevant prior work. In Section 4, we provide an overview of our work and contributions. In Section 5, we introduce our tool Cree and discuss the reduction to Model Checking and four optimizations that are inspired by, but different from, prior work: Polynomial Time Solving when possible, Static Pruning, Abstraction Refinement, and Bound Estimation. In Section 6, we discuss our empirical assessment and results across Cree, and five other prior tools. We conclude with Section 7.

2 ATRBAC-Safety

In this section, we describe ATRBAC, and then pose the ATRBAC-safety problem. We do this in stages. We first introduce RBAC, ARBAC and a version of ARBAC-safety that is relevant to ATRBAC-safety. Then, we describe TRBAC, ATRBAC and ATRBAC-safety.

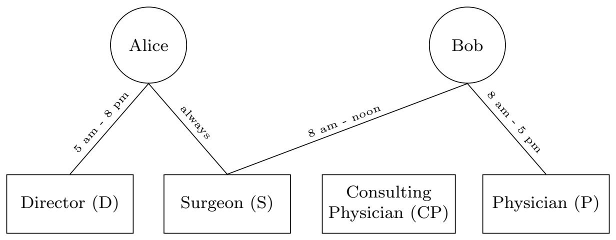
We then clarify that various versions of ATRBAC- and ARBAC-safety are addressed in the literature, and discuss the choices we have made with regards to the various features of the problem. Specifically, that we have chosen the most general of each feature.

2.1 RBAC, ARBAC and ARBAC-Safety

ATRBAC addresses temporal extensions to RBAC and ARBAC. In this section we discuss RBAC, ARBAC and the version of safety analysis in ARBAC that we call ARBAC-safety that is relevant to our work on ATRBAC-safety.

RBAC RBAC [7] is used to specify an authorization policy — who has access to what. An RBAC policy, in the context of this work, is a set UA, the *user–role assignment* relation. An instance of UA is a set of pairs of the form $\langle u, r\rangle$. A user u is authorized to the role r if and only if $\langle u, r\rangle \in UA$. RBAC has other constructs, such as role-permission assignment and a role-hierarchy, that are not relevant to ATRBAC-safety with which we deal in this paper. Indeed, a role-hierarchy can be flattened as a pre-processing step without affecting the correctness or efficiency of our techniques.

ARBAC ARBAC is a syntax for specifying the ways in which an RBAC policy may change. As our work deals with



Rule Type	Rule
t_can_assign	⟨D, 8 am – 9 am, S ∧ ¬P, 8 am – noon, CP⟩
t_can_revoke	⟨D, 8 am – 9 am, true, 8 am – noon, P⟩
t_can_enable	⟨TRUE, 6 am – 8 am, true, 8 am – 5 pm, D⟩
t_can_disable	*No Rules*

Fig. 1: An example of the Start State component (referred to as *TUA* and *RS*) of a TRBAC policy is the figure on top. No roles are enabled. Example ATRBAC administrative rules are in the table. An ATRBAC policy contains: *TUA*, *RS*, and a rule set. Figure 2 contains examples of safety queries.

the *UA* component of an RBAC policy only, by ARBAC we mean its *URA* portion [8], via which users are authorized to and revoked from roles.

There are only two ways in which an instance of *UA* may change. One is the addition of an entry $\langle u, r\rangle$ to *UA*, which is the authorization of u to r. The other is the removal of an entry $\langle u, r\rangle$, which is the revocation of u's authorization to r. An instance of ARBAC is a collection of *rules*, and addresses two issues with regards to such changes to *UA*: who may carry out one of those operations, and under what conditions.

A set of *can_assign* rules control additions to *UA*, and a set of *can_revoke* rules control removals from *UA*. A *can_assign* rule is of the form $\langle a, C, t\rangle$, where a, t are roles and C is a precondition. The precondition C is a set in which each entry is either a role r, or its negation, $\neg r$. The semantics of the *can_assign* rule $\langle a, C, t\rangle$ is that a member of the role a may assign a user u to the role t provided u is already a member of every non-negated role in C and is not a member of any negated role in C.

In a *can_assign* rule $\langle a, C, t\rangle$, the role a is called an administrative role and the role t is called a target role. A *can_revoke* rule has the form $\langle a, t\rangle$ where both a and t are roles. The semantics is that a member of the administrative role a is allowed to revoke a user's authorization from the target role t. The reason that a *can_revoke* rule has no precondition is that revocation is seen as an inherently safe operation [8].

ARBAC-safety We now discuss a version of safety analysis in ARBAC that is relevant to our work. We call it ARBAC-safety. As our work deals with user-role authorization only, ARBAC-safety refers to that aspect only. More general versions of safety analysis for ARBAC have been considered in the literature [9], that reconcile not only the user-role authorizations, but also role-role relationships. Nevertheless, all the versions of safety analysis in ARBAC of which we are aware lie in the same complexity-class — they are all **PSPACE**-complete.

ARBAC-safety is a state-reachability problem. It takes three inputs:

(1) A *query*, which is a pair $\langle u, r\rangle$, user u and role r.
(2) A current- or start-state, which is an instance of *UA*.
(3) A state-change specification, which is an instance of ARBAC, i.e., instances of *can_assign* and *can_revoke* rules.

The output of the ARBAC-safety instance is 'FALSE,' if there exists a state that is reachable from the start-state in which the user u from the query is a member of the role r from the query. Otherwise, the output is 'TRUE.'

ARBAC-safety is known to be **PSPACE**-complete [10]. Several techniques have been proposed to address instances that are likely to arise in practice. For example, Gofman et al. [11] propose a tool called RBAC-PAT, and Jayaraman et al. [4] propose a tool called Mohawk.

2.2 TRBAC, ATRBAC and ATRBAC-Safety

We now discuss the temporal extensions to RBAC and ARBAC that give us TRBAC [12] and ATRBAC respectively. We also the problem ATRBAC-safety. We first present a model and encoding of time that is the basis for the syntax for temporality in ATRBAC. The version we adopt is the same as prior work [2].

Time An intuition for an instance in time, m, can be thought of as represented by a real number. An example of real number time is the Unix timestamp, which is number of seconds since 00:00:00 Jan. 1, 1970 [13]. A time-slot represents some duration of time, and is represented as a non-negative integer. In an instance of ATRBAC-safety, no two distinct time-slots overlap in time. Given time-slots i, j where $i < j$, the time-slot j is associated with a duration of time that occurs later than time-slot i. A time-instant m falls within a time-slot.

We assume that the earliest time-slot with which an instance of ATRBAC-safety is associated is 0, and there is some integer, $T_{\max}$, such that $T_{\max} - 1$ is the latest time-slot that pertains to the ATRBAC-safety instance. We discuss how time progresses under ATRBAC-safety below.

A generalization of a time-slot is a time-interval. A time-interval is a pair of integers $\langle i, j\rangle$ where $i \leq j$. It represents the set of time-slots $\{i, i+1, \ldots, j\}$. We say that a time-instant m falls within a time-interval if m falls within one of the time-slots in that time-interval.

The mindset that underlies the above notions for time is that each time-slot represents some realistic, recurring, fixed time period, such as "9 AM – 10 AM." Example timeslots for T_i are: 9 AM to 5 PM (typical work day), Monday at 9 AM to Friday at 5 PM (typical work week), Jan 1 to March 31 (first quarter of the fiscal year). In ATRBAC, the current instance of time dictates what permissions a user has and what rules an administrator can execute. The particular, the actual time periods, to which time-slots in an instance of ATRBAC-safety map, are irrelevant to the analysis.

TRBAC From the standpoint of our work, TRBAC generalizes RBAC in two, temporal ways. (1) The set *UA* is generalized to *TUA*, each of whose elements is a triple $\langle u, r, l_{u,r}\rangle$, where $l_{u,r}$ is a time-interval. The semantics is that u is a member of r during the time-interval $l_{u,r}$ only. (2) Each role r that appears in *TUA* is annotated with a time-interval, l_r. We say that l_r is the time-interval during which the role r is enabled. The semantics is that outside of the

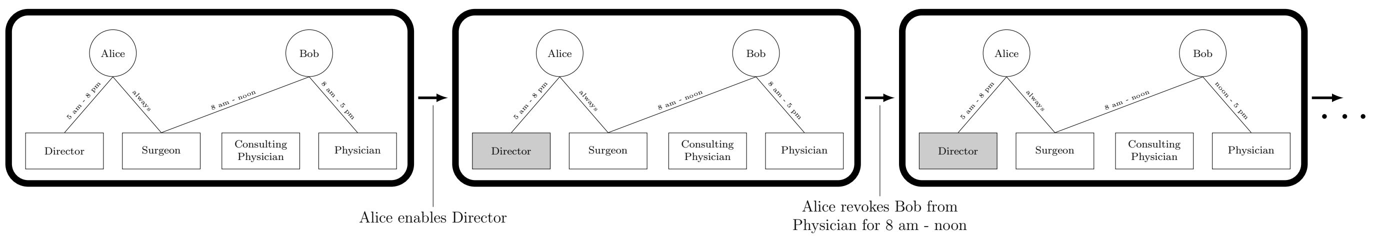


Fig. 2: An example ATRBAC safety query for the policy in Figure 1. We show that the following security query is true, provided we adopt a version of the problem that does not require the target role to be enabled. Security Query: "Can Bob ever become a member of the role Consulting Physician during 8 am to noon?" This security query is shown to be true if there exists a path from the initial TRBAC state to a TRBAC state where Bob is a member of the Consulting Physician role for the timeslot representing 8 am to noon. There are many TRBAC states where this security query is true, and many paths exists to reach these states. Our example path above starts with Alice enabling the Director role (shown shaded). This allows Alice to carry out administrative tasks. Alice then exercises the *t_can_revoke* rule so Bob is revoked from the Physician role for 8 am – noon. She then is able to assign him to the Consulting Physician role for 8 am – noon. This last state-change is not shown in the figure. "Can Bob ever become a member of the role Consulting Physician during 1 pm to 5 pm?," is an example of a safety query that is not true because there exists no paths from the initial TRBAC state to a state where the query is true.

time-interval l_r, no user can exercise a permission that she acquires via the role r. The set of all pairs, $\langle r, l_r \rangle$, is denoted *RS*.

Thus, a TRBAC policy, and therefore a state in the verification problem we consider, is a 3-tuple, $\langle \mathit{TUA}, \mathit{RS}, m \rangle$, where *TUA* and *RS* are as described above, and m is a time-instant. A user u is authorized to a role r at the time-instant, m, if and only if there exists an entry $\langle u, r, l_{u,r} \rangle \in \mathit{TUA}$ such that m is within $l_{u,r}$. The entries in *RS* matter when a user attempts to make an administrative change, i.e., a change to the authorization state. We discuss this under ATRBAC below.

ATRBAC ATRBAC generalizes ARBAC by providing rules for changes to TRBAC policies. As we discuss under "Versions of the problem" below, the version we discuss generalizes prior versions. Under ATRBAC, there are two ways in which a state, which is a TRBAC policy, can change: (1) via an administrative action, or, (2) the passage of time.

Under (1), four kinds of administrative actions are possible to a TRBAC policy, $\langle \mathit{TUA}, \mathit{RS}, m \rangle$. It is possible to add an entry to, and remove an entry from *TUA*, and it is possible to add an entry to, and remove an entry from *RS*. The first two kinds of changes are called assign and revoke administrative actions, and the next two are called role enabling and disabling administrative actions. We have the corresponding sets of tuples *t_can_assign*, *t_can_revoke*, *t_can_enable*, and *t_can_disable*. (As in Uzun et al. [2], we employ the prefix "*t_*" to distinguish clearly that these are rules for ATRBAC, rather than ARBAC.)

Each such set contains 5-tuples. Each tuple is of the form $\langle C_a, L_a, C_t, L_t, t \rangle$. The first two components, C_a, L_a are conditions on the administrator that seeks to effect the action. The next two components, C_t, L_t, are conditions on the user or role to which the rule pertains. The last component, t, is the target-role; the role that is affected by the action. C_a is either the mnemonic 'true,' or a condition, i.e., a set of negated and non-negated roles. L_a is a set of time-intervals. We specify their semantics below for each kind of administrative rule. The entry t is the target role, i.e., the role that is affected by the firing of the rule. C_t is a role-condition similar to C_a above. There are some important differences between C_a and C_t, however, and we discuss these below for each kind of rule. L_t is a set of time-intervals, similar to L_a above. We discuss the semantics of L_t below for each kind of rule as well.

Such a 5-tuple $\langle C_a, L_a, C_t, L_t, t \rangle$ applies when an administrator, say, Alice, attempts an administrative action at a particular time-instant, m. Each administrative action takes inputs, one of which is the administrator that attempts it, i.e., Alice, and others that we discuss below. In Figure 1 we show an example ATRBAC policy and in Figure 2 we show it in the context of a security query. The example in Figure 1 is of 2 users that exist in a hospital. There are 4 roles, where only the Director role is able to act under delegation.

Role enabling: the inputs are Alice, a target role t, and a set of time-intervals, L. Alice succeeds in her attempt at enabling the role t if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t \rangle \in$ *t_can_enable* for which all of the following are true.

(1) The time-instant, m, at which Alice attempts the action falls within some time-interval in L_a. (2) Alice and the current time-instant m together satisfy the administrative condition, C_a. That is, if p is a non-negated role in C_a, then Alice is a member of p at time-instant m in the current state, $\langle \mathit{TUA}, \mathit{RS}, m \rangle$, and p is enabled at the time-instant m. If n is a negated role in C_a, then either Alice is not a member of n at time-instant m in the current state, or the role n is not enabled, or both. If C_a is the mnemonic 'true,' then the rule may fire provided m is within some time-interval in L_a. (3) The set of time-intervals L is contained within the set of time-intervals L_t. That is, for every time-interval $l \in L$, there exists a time-interval $l_t \in L_t$ such that l is within l_t. (4) The set of time-intervals L satisfies the target condition C_t for every $l \in L$. That is, if p is a non-negated role in C_t, then for every $l \in L$, p is enabled during the time-interval l, in the current-state, i.e., *RS*. And if n is a negated role in C_t, then for every $l \in L$, n is not enabled during l, in the current-state.

The effect of a successful role enabling by Alice is that the component *RS* of the current-state is updated as follows to get a new state: $RS \leftarrow RS \cup \{\langle t, l\rangle : l \in L\}$.

Example: In Figure 1 Alice must first enable the role of "Director" so that she can later be allowed to exercise rules where the "Director" role is required by the administrative condition, C_a. Alice may exercise the *t_can_enable* rule during 6 am – 8 am as she satisfies C_a during that time. Once she enables it, Alice must wait before she exercises the *t_can_revoke* rule, where the "Director" role is required by *t_can_revoke*'s C_a, until the current time falls within 8 am – noon.

Role disabling: the inputs are Alice, a target role t, and a set of time-intervals, L. Alice succeeds in disabling t via her action at time-instant m if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in$ *t_can_disable* for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The set of time-intervals L satisfies the target condition C_t for every $l \in L$.

The effect of a successful role disabling by Alice is that the component *RS* of the current-state is updated as follows to get a new state: $RS \leftarrow RS \setminus \{\langle t, l\rangle : l \in L\}$.

User-role assignment: the inputs are the administrator, Alice, a user u, a target role t to which she seeks to assign u, and a set of time-intervals L. The assignment action that she attempts at time-instant m succeeds if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in$ *t_can_assign* for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The user u and the set of time-intervals L satisfy the target condition C_t for every $l \in L$. That is, if p is a non-negated role in C_t, then u is a member of p during every time-interval $l \in L$. If n is a negated role in C_t, then u is not a member of n in any time-interval $l \in L$. If C_t is the mnemonic 'true,' then there are no constraints on the current role-memberships of the user u.

The effect of a successful assignment by Alice is that the component *TUA* of the current-state is updated as follows to get a new state: $TUA \leftarrow TUA \cup \{\langle u, t, l\rangle : l \in L\}$.

Example: The example in Figure 1 shows that Alice is able to assign the "Consulting Physician" role to Bob during 8 am – noon. She is able to exercise this rule because Bob has the role "Surgeon," and does not have the role "Physician" during 8 am – noon, and Alice satisfies the administrative condition by having the role "Director."

User-role revocation: the inputs are an administrator Alice, a user u that she seeks to revoke from a role, a target role, t from which she seeks to revoke u, and a set of time-intervals, L. The revocation action she attempts at some time-instant m succeeds if and only if there exists an entry $\langle C_a, L_a, C_t, L_t, t\rangle \in$ *t_can_revoke* for which all of the following are true.

(1) The current time-instant, m, falls within some time-interval in L_a. (2) Alice and the current time-instant, m, together satisfy the administrative condition, C_a. (3) The set of time-intervals, L, is contained within the set of time-intervals, L_t. (4) The user u and the set of time-intervals L satisfy the target condition C_t for every $l \in L$.

The effect of a successful revocation by Alice is that the component *TUA* of the current-state is updated as follows to get a new state: $TUA \leftarrow TUA \setminus \{\langle u, t, l\rangle : l \in L\}$.

Time-change: Another way that a state, $\langle TUA, RS, m\rangle$, can change is in its time component, m. The manner in which passage of time is modelled [2], [14] is simply by allowing the m component to increase without any change to the other two components, *TUA* and *RS*. That is, a possible state-change is from $\langle TUA, RS, m\rangle$ to a new state, $\langle TUA, RS, m'\rangle$, where $m' > m$.

An issue we clarify in this regard of passage of time is whether, once we reach the time-slot $T_{\max} - 1$ to which an instance of ATRBAC-safety pertains, the time-slot 0 recurs, followed by time-slot 1 and so on, forever. The assumption in prior work [2] is that it does. The reason regards the semantics of a time-slot — it maps to some realistic, recurring period of time. We refer to this property as periodicity, and revisit it in the context of the software tools.

Example: Time periodicity is what allows the rules in Figure 1 to be described by just the time of day. The intention of the rules is that they are contained within a day. Thus when a day ends and the next day begins, the rules should still apply to the new day.

ATRBAC-safety The safety analysis problem for ATRBAC takes three inputs. (1) A query, $\langle u, C, L, t\rangle$, user u, condition C (set of negated and non-negated roles), set of time-intervals L, and t is some units of time. (2) A start-state, $\langle TUA, RS, m\rangle$, which is an instance of TRBAC. (3) A state-change specification, which is an instance of ATRBAC, i.e., four sets of rules, *t_can_assign*, *t_can_revoke*, *t_can_enable*, and *t_can_disable*.

The output is 'FALSE,' if there exists a TRBAC state $\langle TUA', RS', m'\rangle$ that is reachable from the start-state in which: (i) the user u is a member of every non-negated role in C in every time-interval in L, and is not a member of any negated role in C in any time-interval in L, (ii) every non-negated role in C is enabled for every time-interval in L, and no negated role in C is enabled in any time-interval in L, and, (iii) the time-instant m' of this state is within t time-units of the time-instant of the start-state. Otherwise, the output is 'TRUE.' We point out that it is possible to specify t that is large enough that the query pertains to any time-slot.

In Figure 2 we discuss two ATRBAC safety questions: "could Bob become a member of the role Consulting Physician between 8 am and noon?" and "could Bob become a member of the role Consulting Physician between 1 pm and 5 pm?". As the caption of the figure discusses, the former is true, provided we do not require the role Consulting Physician to be enabled when Bob becomes a member of it. The latter question is not true.

Versions of the problem Different versions of ATRBAC-safety appear in relevant prior work. In [6], we created a general version from prior works of Uzun et.al [2], and

Ranise et.al. [14]. We use this general version of the problem as input into our empirical implementation, Cree.

We previously provided a Reduction Toolkit [6], which allows for this general version to be reduced to previous version of ATRBAC-Safety and to a version of ARBAC-Safety. Here we outline the input format of the general version of ATRBAC-Safety we support.

(1) **Rule Types:** *t_can_assign*, *t_can_revoke*, *t_can_enable*, and *t_can_disable*
(2) **Rule Format:** $\langle R_a, Ti_a, C_t, Ts_t, R_t \rangle$; where:
   - Administrator Condition R_a – single role or TRUE
   - Admin Time Condition Ti_a – a time interval in the format of $t_a - t_b$ where $a, b \in \mathbb{Z}$ and $0 \leq a \leq b$
   - Precondition C_t – a list of positive and negative roles (eg: $r_1 \wedge \neg r_2 \wedge r_4$)
   - Target Timeslot Array Ts_t – a list of time slots (eg: t_2, t_3, t_5, t_6)
   - Target Role R_t – single role
(3) **Initial Condition:** empty *TUA* and empty *RS*
(4) **Security Query:** $\langle t_g, R_g \rangle$
   - Goal Timeslot t_g – single timeslot
   - Goal Roles R_g – a list of goal roles

The example, revisited We now revisit the example from Figure 1 and Figure 2 to illustrate the non-triviality of ATRBAC-safety analysis for even such a small example. The system has three rules only, and each rule is essential. For example, if the *t_can_assign* rule is removed, then no user can be assigned to the CP role except by a super-user who operates outside the constraints that the ATRBAC rules impose, which is exactly the scalability issue that such delegation schemes as ATRBAC address. Thus, even though the removal of any rule, in this example, is sufficient to ensure that "could Bob become a member of the role Consulting Physician between 8 am and noon?" is not true, such removal is not a viable way to address the unsafety of the system for that query in this example. One way to ensure that the system is safe for both the queries we discuss above is to change the "8 am – noon" component of the *t_can_revoke* rule to "8 am – 5 pm." Another way is to change the "8 am – noon" constraint in the *t_can_assign* rule to "8 am – 5 pm," and also change the current state so all surgeons, including Bob, are authorized to the Surgeon role during 8 am – 5 pm, rather than 8 am – noon only. We suggest that identifying that such a change simultaneously renders the system safe for both queries, and also from any other queries of interest, is not necessarily straightforward. We point out also that the above discussion suggests that it is not straightforward to label a system as "under-" or "over-constrained." Changing the "8 am – noon" constraint in the *t_can_assign* rule to "8 am – 5 pm" can be seen as easing a constraint. But the consequence of then precluding Bob from certain privileges can be seen as further constraining the system.

3 Prior work

Prior work that is relevant to ours can be dichotomized into: (a) work on access control models and schemes, such as RBAC and its variants, and, (b) work on safety and security analysis in the context of such models and schemes. A comprehensive survey of these is well beyond the scope of this work. In this section, we discuss prior work from the standpoint of safety analysis, (b).

The work of Harrison et al. [1], in the context of the HRU access matrix model, is, to our knowledge, the first to pose and address safety analysis in the context of access control. Since that work, there has been work, for example, on safety analysis of variants of the HRU model [1], and other models such as those for trust management and negotiation [15], [16], usage control models [17], [18], and schemes based on RBAC [7], [19], including ARBAC [8], [20]. There has been work also in generalizing safety analysis to so-called security analysis [21], and the use of safety and security analysis to characterize the expressive power of authorization schemes [22], [23]. Extensions have been made with regards to ATRBAC safety in [24], most notably the security properties: availability, liveness, and mutual exclusion of privileges for TRBAC.

As for ATRBAC-safety, the topic of this work, we are aware of the following prior work. Our prior work, [6], identifies that safety analysis of ATRBAC is **PSPACE**-complete. In addition, it proposes an approach for safety analysis of ATRBAC based on reduction to ARBAC and then use of a prior solver for ARBAC-safety; we call this prior approach Mohawk+T. Our approach in this work, Cree, was designed and created after we gathered experience with Mohawk+T. Our experience with Mohawk+T suggested that directly reducing to model-checking in conjunction with other improvements, such as static pruning/abstraction refinement/bound estimation, would yield better performance. Thus, the work in this paper is different from our prior work [6] – the two propose different tools, with different approaches to the problem.

The work of Uzun et al. [2] is, to our knowledge, the first work to propose ATRBAC and pose the safety-analysis problem for it. In addition, that work discusses the design of two software tools, TredRole and TredRule to address instances in practice. These tools are part of our empirical assessment in Section 6. The work of Ranise et al. [14] syntactically generalizes some aspects of the version of ATRBAC from Uzun et al. [2]. It then presents a result on the computational-complexity of ATRBAC-safety — it proves that the problem is decidable. It then discusses the design, construction and evaluation of 2 different software tools, AsaspTime-Sa and AsaspTime-Nsa, to address problem instances in practice. These tools are also part of our empirical assessment in Section 6.

Safety analysis has been applied to ATRBAC where role hierarchies are allowed in [25], [26], [27]. Safety analysis with role hierarchy was also been applied to ARBAC in [26]. This suggests a limitation in Cree — we do not directly address role hierarchy. We hypothesize that an enhancement to our reduction is possible that incorporates role hierarchy and still yields an efficient approach in practice. We leave this for future work.

Techniques used in this paper include static pruning, abstraction refinement, and bound estimation. For these techniques, we are inspired by prior work on ARBAC-Safety, but with different algorithms to address the new technical challenges posed by ATRBAC's new features. Jha et.al [10] propose forward and backward pruning for ARBAC-safety.

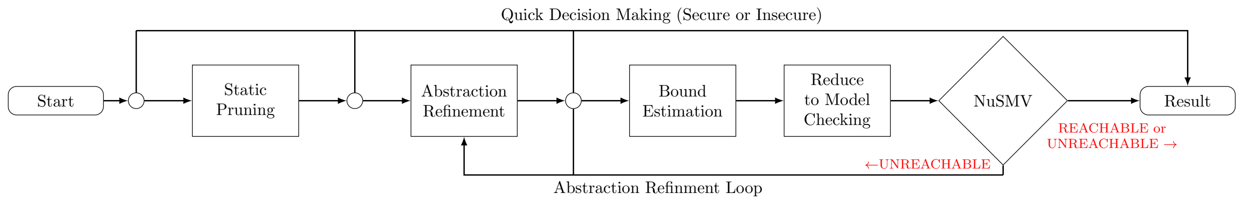


Fig. 3: A high level overview of Cree operates. The Polynomial Time Solving are run at each of the white circles. We translate the Model checking results as follows: REACHABLE inputs are UNSAFE; UNREACHABLE inputs are SAFE.

This was augmented by Jayaraman et.al [4] with abstraction refinement and bound estimation. In [19], Ferrara et.al. provides methods of pruning and bound estimations for ARBAC-Safety. Ferrara extends the forward- and backwards-pruning from Jha et.al [4]. They include more exclusion rules, to create a more aggressive static pruning method. Ferrara's bound estimation found an upper bound on the number of users to be the number of administrative users plus 1. We implement our own versions of static pruning, abstraction refinement, and bound estimation. This is due to the difference between ARBAC-Safety and ATRBAC-Safety. ATRBAC-Safety includes timeslot/time-intervals within to *can_assign* and *can_revoke* rules, and introduces 2 new rule types: *t_can_enable* and *t_can_disable*.

4 Our work

In prior work [6], we propose a tool called Mohawk+T for ATRBAC-Safety. Mohawk+T reduces ATRBAC-Safety to ARBAC-Safety, and then uses the solver, Mohawk [4], that was built for ARBAC-Safety. Our empirical assessment of Mohawk+T suggests that while it is superior to prior tools for some classes of inputs, there are other classes of inputs where the other tools outperform it. This leads us to ask: is there a fundamentally different design we could adopt for ATRBAC-Safety that results in a tool that is more performant? Our answer to this question is 'yes,' and Cree, which we discuss in this work, is such a tool.

In Cree, we reduce ATRBAC-safety directly to model checking. Such a direct reduction gives us performance gains. In addition, it gives us an avenue to design and implement other performance improvements: Polynomial Time Solving when possible, forward- and backward-pruning, abstraction refinement, and bound estimation (see Section 5 for the reduction to model checking, and these algorithms). Thus, Cree is a "from scratch," new tool for ATRBAC-safety, and as our empirical results in Section 6 establish, it is superior to both Mohawk+T and other prior tools for several classes of inputs, and no worse for others. The format of ATRBAC-safety that Cree supports is the same as the one Mohawk+T formalized, and this version generalizes the tools from prior work [2], [14] (see Table 1 in [6]).

Static pruning is a method of reducing the input access control policy by removing rules/roles/timeslots from the policy, which has been shown to greatly improve performance. Static pruning has been discussed for ARBAC in [4], [10], [19], [20], but ATRBAC introduces not only time-intervals/timeslots to each *can_assign* and *can_revoke* rule, but introduces 2 new rule types *t_can_enable* and *t_can_disable*. These additions require new algorithms for static pruning. We outline our forward and backwards pruning techniques in Section 5.2.

Abstraction refinement is an optimization technique for solving problems by solving sub-problems with positive one-sided error. Abstraction refinement starts with the smallest sub-problem, solves the problem and halts if we get a positive answer (the system is unsafe). Otherwise we refine the sub-problem and make it bigger by adding rules and repeat until we halt or have tested a policy equivalent to the original policy. The hope of abstraction refinement is that the solution to the original problem can be solved using only a small percentage of the original problem. Abstraction refinement allows parallel execution for ATRBAC-Safety. We have provided our abstraction refinement technique in Section 5.3.

Our static pruning and abstraction refinement techniques have the ability to produce low complexity ATRBAC-Safety problems. We leverage this by creating an optimization technique called Polynomial Time Solving when possible. This technique is a set of linear/low-order-polynomial timed algorithms that are able to quickly solve specific classes of input of ATRBAC-Safety polices. The class of randomly generated inputs, created in [2], is able to be solved almost exclusively using this technique. When this technique is able to solve our policy, we are able to skip running the model checker, which has high overhead cost. We discuss our Polynomial Time Solving algorithms in Section 5.1.

The previous 3 techniques can be used in conjunction with all ATRBAC-Safety solvers. Our last technique can only be applied to solvers where maximum path length is taken into consideration. Our last optimization technique is Bound Estimation. Where, given an ATRBAC-Safety policy, we calculate an upper-bound on the path length for the smallest path from initial state to goal state. The upper bound from Bound Estimation can be used by the model checker NuSMV with no modification. We discuss out Bound Estimation upper bound calculation in Section 5.4.

Cree is written in Java and is able to run on all modern operating systems. We have provided Cree for public download [28].

5 Cree

We now discuss our tool, Cree. Cree's ATRBAC-safety input format is the same as that of Mohawk+T [6] which in turn generalizes those of prior tools [2], [14]. Following is a list of how the rules and security query are formatted.

Security Query The query is a tuple $\langle s_q, R_q \rangle$, where s_q is a timeslot and R_q is a set of positive roles. The security query is satisfied if any user u is assigned to every role in R_q for the time-slot s_q.

Initial Conditions All ATRBAC-safety instances have the same initial conditions. These are: (1) The user–role assignment set, *TUA*, is empty, and (2) The role–enablement set, *RS*, is empty.

Rules Every rule takes the format $\langle a, L_a, C_t, S_t, t \rangle$. (1) a is an administrative role or the mnemonic 'TRUE'. (2) L_a is a time-interval, ex: $t1 - t5$, or the mnemonic 'T_{all}'. (3) C_t is a set of positive and/or negative roles. (4) S_t is a set of time-slots. (5) t is a role that is to be assigned/revoked/enabled/disabled. Please refer to Section 2.2 to compare formats of the implemented and theoretical rules.

Cree solves instances of ATRBAC-safety by reducing directly to Model Checking and then running a state-of-the-art model checker, NuSMV [29]. This gives us more control over, and insight into, input instances that are harder for our previous tool (Mohawk+T) when compared to prior tools. In the next few sections, we present a number of techniques that exploit this stronger control we have. Specifically, we are able to design and incorporate a number of complementary techniques into Cree that make it considerably more scalable than without those techniques. How Cree operates is shown in Figure 3. The following sub-sections discuss the constituent modules in Cree that are shown in the figure.

5.1 Polynomial Time Solving when possible for ATRBAC-Safety

In practice there are many classes of input where the solution can be determined by using a polynomial timed algorithm. The choice to perform these quick solvers frequently throughout Cree's operation provides many opportunities of increased performance by identifying classes of inputs and skipping labour intensive analysis. The quick solvers run at before and after each modification to the ATRBAC-safety policy.

The following Polynomial Time Solving are performed at each white bubble in Figure 3. The order of execution is important.

(1) Empty Query Role Array Such an input is unsafe when no roles are provided in the query's role array. The result is true for all instances because all users satisfy the condition, regardless of the roles they are members of. Run time of $\Theta(1)$.

(2) No *t_can_assign* Rules for the Security Query's Role Array and Timeslot Such an input is safe because the initial conditions have no users assigned to any roles and with no *t_can_assign* rules to obtain the query's roles, for the query's timeslot, it is impossible for a user to obtain the goal roles. Run time of $\Theta(|t\_can\_assign|)$.

(3) No "Truly-Startable" *t_can_assign* Rules A "Truly-Startable" rule is one that has C_t = TRUE and a = TRUE. This means that a Truly-Startable *t_can_assign* rule is satisfied in the initial conditions. If no Truly-Startable *t_can_assign* rules exists then the initial user-role assignment *TUA* can never be changed. Thus, such an input is safe. Run time of $\Theta(|t\_can\_assign|)$.

5.2 Static Pruning

Static Pruning is a method of removing Rules, Roles, and Timeslots from an ATRBAC-safety policy that are irrelevant for safety analysis. Static pruning is one of the first steps to be performed, because reducing policy size can increase performance and minimizes memory usage for all operations following. Figure 4 shows an example of a policy that can be made smaller without affecting the safety analysis. Specifically, the policy has several rules (highlighted in red) that can be removed without impacting safety analysis. We are able to remove these rules using a combination of forward and backwards pruning. This, in turn, can improve performance by reducing the memory size required to store each state, and by reducing the number of states needed to search. Static pruning has been discussed in ARBAC [4], [10], [19], [20]. We have modified the methods of forward and backwards pruning, from [4], [10], [20], to update the algorithms to introduction time into *t_can_assign*/*t_can_revoke* rules and to apply similar pruning methods to *t_can_enable*/*t_can_disable* rules (which do not exists in ARBAC). Both pruning techniques can be turned off in Cree, but due to empirical evidence, both techniques were used to obtain the best results in Section 6.

```
// Goal State: CE1(admin) -> CE3(a) -> CA6(a) ->
      CA6(user) -> CA4(u) -> CR2(u) -> CA2(u) ->
      CA6(u)
Query : t2, [r3, r4]
CanAssign:
  /*CA1*/ <TRUE, t1-t3, TRUE, [t2,t3], r1>
  /*CA2*/ <r3, t1-t3, r2 & NOT r3, [t2,t3], r4>
  /*CA3*/ <r1, t1-t3, r1, [t2,t3], r5>
  /*CA4*/ <r3, t1-t3, r3, [t1], r2>
  /*CA5*/ <r1, t1-t3, r5 & NOT r3, [t2,t3], r6>
  /*CA6*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
CanRevoke:
  /*CR1*/ <TRUE, t1-t3, TRUE, [t1,t2], r1>
  /*CR2*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r3>
  /*CR3*/ <TRUE, t1-t3, TRUE, [t1,t2,t3], r2>
CanEnable:
  /*CE1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CE2*/ <TRUE, t1-t2, TRUE, [t2], r1>
  /*CE3*/ <TRUE, t1-t2, r1 & NOT r2, [t1], r3>
  /*CE4*/ <TRUE, t1-t2, r1, [t1], r2>
CanDisable:
  /*CD1*/ <TRUE, t1-t2, TRUE, [t1], r1>
  /*CD2*/ <TRUE, t1-t2, TRUE, [t1], r2>
```

Fig. 4: An example that shows rules/roles/timeslots, that can be removed without changing the security of the policy. All lines highlighted in red are not required to show that this policy is unsafe. Only the lines highlighted in green are required. Removing all unnecessary rules/roles/timeslots with static pruning ensures: (1) the pruned policy's size will be less than or equal to the input policy's size, and (2) the safety response remains unchanged.

Forward Pruning is a technique that disregards the security query and focuses on removing rules that are unable to fire. It does this by looking at every rule in the policy and verifying if the admin role is assignable and enable-able, and if each positive role in C_t is assignable. If any of these are false then the rule cannot fire and we can safely remove the rule from the policy. The previous assertion is only true if we start from an empty state, or if we start from a state where every user has obtained their roles using the current set of rules.

ATRBAC forward pruning is in P, our algorithm has a run time of $\Theta(|\text{rules}|)$. Our forward pruning algorithm differs from previous works by introducing the *t_can_enable t_can_disable* pruning and adding the condition that administrator roles must also be enable-able. The forward pruning algorithm is supplied in the supplemental material.

Backward Pruning focuses on the security query, $\langle R_q, s_q \rangle$, to determine which rules it can add to a new policy P''. Where the security is true if there exists a state transition path from the initial condition to a user who is assigned all roles in the set R_q for the timeslot s_q. Backward pruning collects all *t_can_assign* rules, $\langle a, L_a, C_t, S_t, t \rangle$, where the target role t equals a role in the security query's role set R_q and the timeslot array S_t contains the security query's timeslot s_q. These initial *t_can_assign* rules is the set of rules that is required to obtain each of the goal roles in the security query. Not all of the rules in this set are necessary to satisfying the security query, but a subset of these *t_can_assign* rules are required. From this initial set we extract all of the conditions that each rule requires in order to fire: (1) all positive roles in C_t must be satisfied for all timeslots in S_t, (2) all negated roles in C_t, that also appear as a target role in some *t_can_assign* rule, must be revocable (if possible), (3) all administrator roles a must be assigned (matching *t_can_assign* rule(s)) and enabled (matching *t_can_enable* rule(s)). We generate the first *t_can_assign* set, extract all conditions that the rules require to fire, and then find all corresponding rules that can satisfy these conditions. We loop this process until all conditions are possibly met; some policies have conditions that cannot be satisfied because no satisfying rule exists.

ATRBAC backward pruning is in P, our algorithm has a run time of $\Theta(|\text{rules}|^2)$, in the worst case. Our backward pruning algorithm differs from previous works by introducing *t_can_enable*/*t_can_disable* rules, and the more difficult addition of introducing time to every satisfying condition. The algorithm for backward pruning is provided in the supplementary material.

5.3 Abstraction Refinement

Abstraction Refinement is a technique that can increase performance of safety analysis for insecure policies. It uses the assumption that only a small portion of an ATRBAC policy is required to determine if a policy is insecure. Abstraction refinement works by creating a very small policy P_0 from a policy P, then performing safety analysis. If the small policy P_0 is insecure, then we know P is insecure and can halt early. If P_0 is secure, then we don't know if P is secure or insecure and thus we create a slightly bigger policy P_1 and repeat. We repeat until we find a policy that is insecure or we do a safety analysis on the original policy P.

In static pruning we removed rules that could not fire in any situation and rules not relevant to the safety query, this ensures the security is unchanged due to pruning for all instances of ATRBAC-safety. Abstraction refinement has 1-sided error, if a sub-policy P_i is insecure, then the original policy is also insecure. For secure policies we must perform security analysis on all sub-policies and the original in order to confirm that the policy is secure.

In this section we devise a strategy to produce a small sub-policy and then iteratively increase the policy size by adding more rules from the original policy until the original policy is reproduced without any irrelevant rules.

ALGORITHM 1: Step 0 for Abstraction Refinement

```
Input: $P \leftarrow$ Cree Policy
Result: $P_0 \leftarrow$ Reduced Cree Policy (Size: $|P_0| \leq |P|$)
Func AbsRef_Step0($P$)
    $P_0 \leftarrow$ Empty Policy; $P_0$.query $\leftarrow \langle s_q, R_q \rangle \leftarrow P$.query;
    $R_0 \leftarrow \{P.R_q\}$; $T_0 \leftarrow \{P.s_q\}$;
    for $r \in$ all rules in $P$ do
        if $(r.t \in R_0) \wedge (\exists s | s \in r.S_t \wedge s \in T_0)$ then
            $P_0 \leftarrow P_0.rules \cup r$ ;
```

Abstraction Refinement improves performance for instances where only a small subset of rules are required to prove that a policy is insecure. In cases where a large set of rules is required for an insecure result, or if the policy is secure, this option would have a negative impact on the run-time. We can eliminate the negative performance impact by running on a computer able to solve the original policy in parallel with the abstraction refinement steps; halting as soon as one finishes. Abstraction Refinement can be turned off and is suggested if the ATRBAC-safety policy is assumed secure.

The aggressiveness of the initial policy and the iterative steps can effect performance. More aggressive algorithms reduce the number of abstraction refinement steps by allowing more rules with each iteration, thus speeding up performance for secure policies. Less aggressive algorithms reduce the load on the solver by producing smaller and easier to solve sub-policies, thus speeding up performance for insecure policies. We have found a balance that works for the empirical testing set we used. To have an apples-to-apples comparison, Cree was run with abstraction refinement turned on for all empirical results reported in this paper. All algorithms below run in polynomial time, and the number of abstraction refinement steps is bounded linearly by the number of rules. Thus Abstraction Refinement is in P.

Step 0: Initialization and Smallest Policy

Algorithm 1 shows how that initial abstraction refinement step is performed and the smallest sub-policy, P_0, is created. P_0 is a new empty policy, with the same security query as the policy. We first create 2 variables: R_0 and T_0. $R_0 = \{\text{all of the roles from the Query}\}$. $T_0 = \{\text{timeslot from the Query}\}$. P_0 contains the set of rules from the original policy P if the rule satisfies the following conditions: the target role is in R_0, and at least 1 time-slot s in the target timeslot array must be in T_0 $(\exists s | s \in r.S_t \wedge s \in T_0)$.

Empirically we have found that this semi-aggressive algorithm performed best. Other initial algorithms considered were: limiting rule selection to only CanAssign rules, removing the timeslot restriction (using the ARBAC abstraction refinement initial step in [4]), and skipping the initial step and using the first refinement iteration.

Step N: Iterative Refinement

Algorithm 2 shows how iterative sub-policies are created from the previous sub-policy, P_{N-1}, and the original ATRBAC-safety policy P. The initial abstraction refinement step uses the security query to add all rules from P where

ALGORITHM 2: N^{th} Step for Abstraction Refinement

```
Input:
• $P$ and $P_{N-1}$ – Cree Policies
• $R_{N-1}$ – Set of Roles from $P_{N-1}$
• $T_{N-1}$ – Set of Timeslots from $P_{N-1}$
Result: $P_N$ - Reduced Policy (Size: $|P_{N-1}| \leq |P_N| \leq |P|$)
Func AbsRef_StepN($P, P_{N-1}, R_{N-1}, T_{N-1}$)
    $P_N$.query ← $P_{N-1}$.query; $R_N \leftarrow R_{N-1}$; $T_N \leftarrow T_{N-1}$;
    for $r \in$ all rules from $P_{N-1}$ do
        $R_N \leftarrow R_N \cup r.t \cup r.C_t \cup r.a$;
        $T_N \leftarrow T_N \cup r.S_t \cup r.L_a$;
    for $r' \in$ all rules from $P$ do
        if $(r'.t \in R_N) \wedge (\exists s | s \in r'.S_t \wedge s \in T_N)$ then
            $P_N \leftarrow P_N.rules \cup r'$
```

$t \in R_q$ and $s_q \in S_t$. Each iterative step after that increases the set of target roles to include all roles administrative a and roles in precondition C_t from all rules in the previous abstraction refinement step. Increasing the target role set and the timeslot set increases the number of rules that can be added to the new policy.

Algorithm 2 is a semi-aggressive algorithm and was chosen based on empirical evidence. A less aggressive alternative is similar to Tightening 3 in Section 5.4, where R_N is split into 2 sets, roles that need to be assigned and roles that require enablement. A more aggressive alternative would be the same as the ARBAC scheme in [4], where there is no time constraint.

Step M: Termination The last refinement step occurs when $|P_{M-1}.rules| = |P_M.rules|$. This usually occurs when $R_{M-1} \cap R_M = \emptyset$ and $T_{M-1} \cap T_M = \emptyset$. Note that $|P_M.rules| \leq |P.rules|$. $|P_M.rules| < |P.rules|$ can occur when there exists rules in P that do not help a user satisfy the security query. These rules will not be added to P_M. Table 1 shows the abstraction refinement steps when applied to the example policy in Figure 4. In that example, the rules CA3 and CA5 are not in the last abstraction step. If we had performed static pruning before abstraction refinement we could have reduced the size of the last policy by 5 rules (rules highlighted in red in Figure 4).

5.4 Bound Estimation

NuSMV has two modes of operation: Symbolic Model Checking (SMC) and Bounded Model Checking (BMC). SMC mode combines states that share aspects in the state tree and utilizes these groups to traverse the tree more efficiently then using single step. BMC mode uses single step to traverse the state tree, but limits the distance from the initial state. In BMC mode, an additional integer input, called a *bound*, is required by NuSMV. This bound changes the problem to whether an unsafe state can be reached within that diameter from the start-state. To ensure we get a correct answer to our original safety query, we require a bound significantly large enough that if unsafe states exists, then at minimum 1 unsafe state must exist within the state tree with our fixed bound. Tightening this bound allows NuSMV to run more efficiently in BMC mode.

Part of Cree is an algorithm for estimating this bound; we call this algorithm bound estimation. It works by calculating an initial upper bound and then applying what

Step i	ΔR_i	ΔT_i	Δ Rules
0	r_3, r_4	t_2	CA2,CA6,CR2,CE3
1	r_2, r_1	t_1, t_3	CA1,CA4,CR1,CR3,CE1, CE2,CE4,CD1,CD2
2	$\emptyset$	$\emptyset$	$\emptyset$

TABLE 1: Example Abstraction Refinement steps using Algorithm 1 for step 0 and Algorithm 2 for steps 1 and 2. Notice that the columns are showing the difference in sets from the current step and the previous step. It should be noted that the policies below will be reduced if static pruning is applied before step 0.

we call tightenings that reduce this upper-bound, while maintaining the invariant that the original input is safe if and only if it is safe with our estimate for the bound. In practice, the initial estimate tends to be loose upper bound, with the tightenings gradually decreasing it.

Cree has the option to use SMC mode and forgo bound estimation; all results in Section 6 were found using BMC mode and bound estimation. If we used only the initial upper bound, we found that SMC was empirically quicker on average, but when utilizing tightening 3 we found BMC mode to be on average the quicker mode.

Initial Upper Bound calculates the length of a simple path that visits every possible state in an ATRBAC policy. In the context of ATRBAC, a state represents the user-role assignments and the role-enablement status; i.e. which users are assigned to which roles for which timeslots and which roles are enabled for which timeslots. We can represent this state as a binary string, where the length is equal to $|\text{User Role Assignments}| + |\text{Role Enablement}| = (|\text{Users}| \cdot |\text{Roles}| \cdot |\text{Timeslots}|) + (|\text{Roles}| \cdot |\text{Timeslots}|)$. From this we can calculate the maximum simple path to visit every state; the length of this simple path is calculated using $2^{|\text{Binary String}|}$. If the User-Role Assignment or the Role Enablement status had a more complex state than on/off, then the above calculation for simple path would not be accurate.

We require a $Users$ variable to compute the initial upper bound. If there exists no known limit to the number of users to a specific ATRBAC policy, then we can use the worst case: in a single state we have 1 user able to represent every role for every single timeslot at once ($|\text{Users}| = |\text{Roles}| \cdot |\text{Timeslots}|$), thus we have the ability to solve every possible administration condition in an ATRBAC policy.

User-Role-Timeslot Assignment:

$$u = |\text{Users}| \cdot |\text{Roles}| \cdot |\text{Timeslots}| = (|\text{Roles}| \cdot |\text{Timeslots}|) \cdot |\text{Roles}| \cdot |\text{Timeslots}|$$

Role-Timeslot Enablement:

$$r = |\text{Roles}| \cdot |\text{Timeslots}|$$

Upper Bound:

$$d_0 = 2^{u+r} = 2^{(|\text{Roles}|^2 \cdot |\text{Timeslots}|^2) + (|\text{Roles}| \cdot |\text{Timeslots}|)}$$

Tightening 1: Required Number of Users The initial upper-bound uses the variable *Users*, which is not defined in a standard ATRBAC policy, thus it uses a large value which it knows will contain enough users. This tightening defines a tighter upper-bound by reducing the number of users required. It is important to notice that in an ATRBAC policy User A cannot effect User B's assignments unless User A acts as an administrator; i.e. there is no condition in our CA/CR rule which checks which roles another user is assigned to, we only have conditions checking the administrator and the target user. From this observation we can limit the number of users that we need to satisfy in our simple path to the number of administrative roles (1 user per role) and 1 user target user. To intuit the proof, in the worst case we require that each administrator role be assigned to a different user (i.e. *t_can_assign* rules: $\langle \text{TRUE}, t_{all}, \neg a_1 \wedge \neg a_2 \wedge \ldots \wedge \neg a_m, [t_0], a_0 \rangle \ldots$, no *t_can_revoke* rules). This means that we require the number of administrator users and 1 target user. We then calculate the length of a simple path to visit all states for all users, as this ensures that if an unsafe state exists then it will exist within our bound. This result is similar to that reported in [19], for the number of users required for analysis in an ARBAC policy.

$$|\text{Users}| = |\text{Admin Roles}| + 1$$
$$d_1 = 2^{(|\text{Admin Roles}|+2)\cdot|\text{Roles}|\cdot|\text{Timeslots}|}$$

Tightening 2: Leveraging Initial Conditions The initial conditions of an ATRBAC policy is that every user is assigned with no roles and every role is disabled. We can infer that the only rules that can change the initial condition is *t_can_assign* and *t_can_enable* rules. To get an intuitive sense of how this is true, imagine every possible user to a system as a 3 dimensional array: users$\times$roles$\times$time-slots $\mapsto \{0, 1\}$, and imagine whether a role is enabled or disabled as a 2 dimensional array: roles$\times$time-slots $\mapsto \{0, 1\}$. The first array, a specific user u, role r, and timeslot t point to a binary variable stating whether user u is a member of role r for the timeslot t. The second array is similar but indicating if a role is enabled for a specific timeslot. The initial state sets all of these binary values to 0, thus any call to any *t_can_revoke* or *t_can_disable* will not change the state ($\underset{init}{\longrightarrow} 0 \underset{CR}{\longrightarrow} 0$). In order for *t_can_revoke* or *t_can_disable* to change state, a specific *t_can_assign* or *t_can_enable* call must precede it ($\underset{init}{\longrightarrow} 0 \underset{CA}{\longrightarrow} 1 \underset{CR}{\longrightarrow} 0$). From this intuitive knowledge, and by building on tightening 1, we obtain the upper bound in Figure 5.

From our observation, we can limit the number of roles and the number of time slots required by tightening 1. For User-Role-Timeslot Assignment, we only require to visit roles which we might need to obtain as a target user or as an administrator. The list of roles can be found in the set (CA-CR-Positive-Precondition$\cup$Admin-Roles$\cup$Goal-Roles). We can limit this set to only include roles which can actually be assigned using *t_can_assign* rules in the ATRBAC policy (i.e. target roles of all *t_can_assign* rules). We can limit the number of timeslots required to be the set of target timeslots which appear in all *t_can_assign* rules. We then perform similar tightenings for the Role-Timeslot Enablement path.

Tightening 3: Longest Simple Path to Goal State Tightening 3 uses the longest simple path from the initial state to the goal state as a means of measuring the maximum number of role/timeslots required by the target user, each admin user, and for the role enablement. Using these roles, we calculate our bound as the length of a simple path which visits every state.

User-Role-Timeslot Assignment:

$$\begin{aligned}|u_2| =&(|\text{Admin-Roles}| + 1) \cdot |\text{CA-Target-Roles}\\ &\cap (\text{CA-CR-Positive-Precondition} \cup \text{Admin-Roles}\\ &\cup \text{Goal-Roles})| \cdot |\text{CA-Target-Timeslots}|\end{aligned}$$

Role-Timeslot Enablement:

$$\begin{aligned}|r_2| =&|\text{CE-Target-Roles} \cap (\text{CE-CD-Positive-Precondition}\\ &\cup \text{Admin-Roles})| \cdot |\text{CE-Target-Timeslots}|\end{aligned}$$

Upper Bound:

$$d_2 = 2^{|u_2|+|r_2|}$$

Fig. 5: Tightening 2, reduces the required number of roles and timeslots required by the Assignment and Enablement portions of the upper bound. By leveraging the initial state of ATRBAC-safety analysis, we can determine that unless a *t_can_assign* rule or *t_can_enable* is executed, then all *t_can_revoke* or *t_can_disable* rules will not change the state tree. This allows us to limit the number of roles to only the role that are required to be assigned and required to be enabled. We further limit this by only including the roles that are able to be assigned and able to be enabled. We limit the timeslots based on just on the *t_can_assign* and *t_can_enable* rules.

To understand the bound in Figure 6, we split the required state size into three sections: (1) a target user obtaining the goal roles, (2) other users obtaining required administrator roles, and (3) having each administer role enabled at the right time. The longest simple path (LSP) is calculated by looking at all the rules that assign, or enable, a specific state (i.e. role a for timeslot t_3) and then working backwards, to collect all rules that a target user require and returning the longest path. This is similar to backwards pruning in Section 5.2. We identify rules that might be required based on the positive preconditions. We do not consider the assignment or enablement of admin role in the LSP algorithm, as they are specifically handled in U_{admin} and RE.

The intuition behind tightening 3 is to minimize the number of role/timeslots variables in our state representation for the upper bound on the minimum length path from initial to goal state. Tightening 1 reduced the number of users required to solve all ATRBAC policies. From tightening 2, we recognize that if there exists no *t_can_assign*/*t_can_enable* rule for a particular role/timeslot, then we can remove it from our state regardless if there exists a *t_can_revoke t_can_disable* rule. Tightening 3 utilizes these observations and applies them based on the security query.

We notice that to achieve our goal state, we must have a user contain all goal roles for the goal timeslot. In order for this user to obtain the above, we must have administrator users able to execute rules on the user. In order for an administrator user to fire a rule, their administrative role must be enabled. This is how the three steps above were created. For each step we determined the longest shortest path (LSP) from the initial conditions to a state (i.e. user obtaining a single goal role for the goal timeslot, or an admin user obtaining their admin role for a specific timeslot). We use the LSP algorithm to determine the maximum number of role/timeslots to keep for that particular user. The exact

$$U_{goal} = \left|\text{CA-Target-Roles} \bigcap \left[\bigcup_{g \in \text{Goal-Roles}} \text{Get-Roles}[LSP_{CA,CR}(g)]\right]\right| \cdot \left|\bigcup_{g \in \text{Goal-Roles}} \text{Get-Timeslots}[LSP_{CA,CR}(g)]\right|$$
$$U_{admins} = \left[\sum_{a \in \text{Admin-Roles}} |\text{CA-Target-Roles} \cap \text{Get-Roles}[LSP_{CA,CR}(a)]| \cdot |\text{Get-Timeslots}[LSP_{CA,CR}(a)]|\right]$$
$$RE = \left|\text{CE-Target-Roles} \bigcap \left[\bigcup_{a \in \text{Admin-Roles}} \text{Get-Roles}[LSP_{CE,CD}(a)]\right]\right| \cdot \left|\bigcup_{a \in \text{Admin-Roles}} \text{Get-Timeslots}[LSP_{CE,CD}(a)]\right|$$
$$d_3 = 2^{U_{goal} + U_{admins} + RE}$$

Fig. 6: Tightening 3, calculates a diameter for NuSMV by returning the longest simple path for a user to be assigned the goal roles and for all required administrative roles to be assigned and enabled. We calculate the possible simple paths by working backwards from the goal roles and using rule preconditions to add rules to the simple path. The function $LSP_{x,y}(r)$ returns a set of rules from the input ATRBAC policy's *t_can_assign*/*t_can_revoke* ($x, y = CA, CR$) or *t_can_enable*/*t_can_disable* ($x, y = CE, CD$) sections, that represents the longest simple path to the role r.

roles/timeslots are not important, as we are only using this as a method of estimating the minimum path length from the initial state to the goal state, and not actually changing the underlining TRBAC state.

In the previous 2 tightenings, we assumed each administrator user required the same number of states. In this tightening we calculate the required state size for each administrator user and add the sizes together (see U_{admins} in Figure 6). In the worst case, we find that all administrative users require the same of role/timeslots and thus see no improvements from tightening 2. We modify the Role Enablement state size by limiting the number of roles to those required to enable the admin roles.

From Figure 6, below we explain the functions used. Get-Roles() is a polynomial time function that takes a set of rules and returns the set of roles that contains all target roles and positive preconditions. Get-Timeslots() is a polynomial time function that takes a set of rules and returns the set of timeslots that contains all target timeslots. LSP is a polynomial time algorithm that can be used on the *t_can_assign*/*t_can_revoke* or the *t_can_enable*/*t_can_disable* rulesets, it takes a target role l and returns the set of rules which is associated with the longest shortest path. The LSP functions works backwards from the state where a the role is assigned, or enabled (depends on the ruleset), starts with the set of rules with target role equal to l, and uses the preconditions to build a set of rule paths. Since we are dealing with rule sets, the set size is linearly limited by the number of rules, and the number of rule sets is limited linearly by the number of rules.

We have provided a full example of calculating each tightening in the supplemental material.

5.5 Reduction to Model Checking

A correct reduction from ATRBAC-Safety to Model Checking must ensure: (1) that every state an ATRBAC policy can reach is reachable by our reduced model, and (2) every state that our reduced model can reach must be reachable by the ATRBAC policy. We chose to reduce to model checking because of the similarities between ATRBAC-safety and model checking, and to leverage the mature model checking community for their robust and highly optimized software.

Model checkers have 2 variable types: state variables and control variables. The state variables have controlled values, where state transitions dictate future values. Control variables are left to the model checker to choose a value for. This allows the model checker to perform optimizations to dictate which state transitions to pick when a choice is required.

Our reduced model has 2 state variables: (1) a $|users| \times |roles| \times |timeslots|$ binary array *TUA* that indicates if a user is assigned to a role in a specific timeslot, and (2) a $|roles| \times |timeslots|$ binary array *RS* that indicates if a role is enabled for a specific timeslot. The value for $|roles|$ is extracted from the list of all roles in the ATRBAC policy, $|timeslots|$ is extracted using the polynomial reduction algorithm to convert time intervals into non-overlapping timeslots (see Reduction 2 in [6]). We use tightening 1 to calculate the value for $|users|$. Both *TUA* and *RS* are initialized to false (ATRBAC-safety initial conditions) and each variable has a state transition where the default case keeps the current state. *TUA* and *RS* accurately represent the underlining TRBAC state in an ATRBAC policy.

We have 3 control variables that we let the model checker control:

$rule$ an enumerated integer that can take the value of any rule in the ATRBAC policy (ex: $rule$: {CA01, CA02, . . ., CR01, . . ., CE01, . . ., CD01, . . .};),

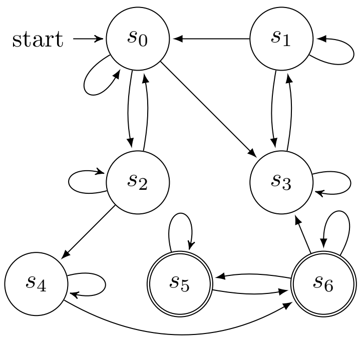
$user$ an integer between 1 and the maximum number of users (|Admin-Roles| + 1, see Section 5.4), to represent a selected target user, and

$admin$ which is modelled the same as $user$ but is independently assigned a value, this represents a selected admin user.

For each *t_can_assign*/*t_can_revoke* rule, we add a state transition to each affected *TUA* variable if the admin and target user satisfy the rule's condition in the current state. The model checker is able to change the state of *TUA* by assigning values to $rule$, $user$, and $admin$ such that:

1) the $rule$ is a *t_can_assign* or *t_can_revoke* rule,
2) the $user$'s values in *TUA*, in the current state, satisfy the $rule$'s preconditions C_t,
3) the $admin$'s values in *TUA*, in the current state, satisfy the $rule$'s administration condition, and
4) the $rule$'s administration condition is enabled for at least 1 timeslot in *RS*, in the current state.

For each *t_can_enable*/*t_can_disable* rule, we add a state transition to each affected *RS* variable if the admin and current state satisfy the rule's condition. The model checker



(a) Diagram Model

```
1   LTLSPEC G !(u[1][3][2] = TRUE & u[1][1][2] = TRUE) // Query: t2, [r3, r4]
2   // User 1, Role 3 Assign/Revoke
3   next(u[1][3][1]) := FALSE;
4   next(u[1][3][2]) := case
5   user=1 & rule=CA1 & ( (u[admin][1][1]&e[1][1])|(u[admin][1][2]&e[1][2])|(u[admin][1][3]&e
            [1][3]) ) & (u[user][2][2]&u[user][2][3]) & !(u[user][1][2]|u[user][1][3]) : TRUE;
6   TRUE : u[1][3][2];
7   esac;
8   next(u[1][3][3]) := case
9   user=1 & rule=CA1 & ( (u[admin][1][1]&e[1][1])|(u[admin][1][2]&e[1][2])|(u[admin][1][3]&e
            [1][3]) ) & (u[user][2][2]&u[user][2][3]) & !(u[user][1][2]|u[user][1][3]) : TRUE;
10  TRUE : u[1][3][3];
11  esac;
12  // Role 1 Enable/Disable
13  next(e[1][1]) := case
14  rule=CE2 & (e[4][1]) & !(e[2][1]) : TRUE;
15  TRUE : e[1][1];
16  esac;
17  next(e[1][2]) := FALSE;
18  next(e[1][3]) := FALSE;
```

(b) SMV Code

Fig. 7: Figure (a) depicts the diagram view of model checking, where s_0 is the initial ATRBAC state and each state transition is effecting a rule r with an admin and target user that satisfy the rule's admin and target role conditions. States s_5 and s_6 are the accepting states, where a user in *TUA* satisfies the security query. Figure (b) shows a version of (a) using a small snippet from the reduction of Figure 4 to NuSMV's model checking language. The TRBAC state is encoded as such: *TUA* is defined by u and *RS* is defined by e in out reduction above. In s_0, u and e are set to FALSE, where u is a 3 dimensional array: $users \times roles \times timeslots$, and e is a 2 dimensional array: $roles \times timeslots$. The reduction has a rule for each user/role/timeslot variable. If no rule touches that variable (lines 3,17,18), then the next state value will always be the initial condition: **FALSE**. Otherwise, a case statement is used to check the rule/user/admin values to see if the rule's conditions are satisfied. The default route for the case statement is to keep their current value. There are 2 parts to understand in a case statement (line 5): to the left of the colon is the condition to check and the right side is the value we set when the condition is true. The query on line 1 asks the question "For all future time, will user 1 never be assigned r3 and r4 for timeslot t2?".

is able to change the state of *RS* by assigning values to $rule$, $user$, and $admin$ such that:

1) the $rule$ is a *t_can_enable* or *t_can_disable* rule,
2) the current state in *RS*, satisfy the $rule$'s precondition,
3) the $admin$'s values in *TUA*, in the current state, satisfy the $rule$'s administration condition, and
4) the $rule$'s administration condition is enabled for at least 1 timeslot in *RS*, in the current state.

We can intuit the correctness of the reduction by comparing state transitions to rule firings. In an ATRBAC policy, to fire a rule r the target and admin users must both satisfy their conditions, and the admin role must be enabled for the current time of day. If these conditions are true then the TRBAC instance is updated for the *TUA* (*t_can_assign*/*t_can_revoke*) or the *RS* variable (*t_can_enable*/*t_can_disable*). The reduced model checks all conditions except for the check if the admin role is enabled during the current time. In Section 2.2, we define timeslots to be periodic, in our reduce model we utilize this to optimize the model by ignoring time. We are able to achieve this because we can stay in the current state until the current time reaches one of the timeslots required to enable the admin role. Without this optimization, we would require a time state variable to keep track of the current timeslot and state transitions to increment, and wrap, to the next timeslot.

In Figure 7, we show a small snippet of the reduction from Figure 4 to model checking. The state variables user/admin/rule are defined as above. To save space, *TUA* and *RS* were renamed to u and e respectively. We can understand u and e by their index values: $u[user][role][timeslot]$, $e[role][timeslot]$. Figure 7 does not show the variable declaration or initialization, and hides most of the rules for user1 and all rules for user2. The rules to dictate rule enablement are truncated.

The query on line 1 can be understood as: "Does the statement after G hold true for all future time instances?". We translate the query to model checking by negating the query "can any user ever obtain the goal roles for the goal timeslot?". An optimization to the query is that we force user1 to be our target user, by limiting our query to "can $u[1]$ ever obtain the goal roles for the goal timeslot?".

In Figure 7 each state variable must have a state transition statement (`next` or `case`). Without these statements the model checker would be able to change the value between states, thus obtaining states that are unreachable to the ATRBAC policy. For role×timeslots that are not touched by a *t_can_assign* rule, we set the next value in u to **FALSE**, for all users, as the value cannot change from the initial value $FALSE$. We do the same with e and *t_can_enable* rules.

The format for state transitions for the *TUA* variable u are: (correct user) $\wedge$ (correct rule) $\wedge$ (admin satisfies condition for any timeslot in interval) $\wedge$ (user satisfies condition for all timeslots) : TRUE. The format for state transitions for the *RS* variable e are: (correct rule) $\wedge$ (admin satisfies condition for any timeslot in interval) $\wedge$ (e satisfies condition for all timeslots) : TRUE. Admin and target condition which are TRUE are satisfied in all states. The last line of each case statement is the default route, and this prevents state changes outside of the rules in the ATRBAC policy.

5.6 Performance Considerations

The performance techniques described above are able to give a performance increase in the best case, and in the worst case provide extra overhead that decreases overall performance. In this section we will consider each of the performance techniques and outline conditions where a performance increase or decrease is expected.

The Polynomial Time Solving module of Cree, is optimized to run very quickly because they are executed before each technique. We see a performance increase if the ATRBAC policy is relatively simple or when running with abstraction refinement on. We have found that the initial abstraction refinement steps are able to be solved almost exclusively using Polynomial Time Solving. We also found that we do not see any noticeable decrease in performance if this technique is not able to solve the policy.

Forward pruning is effective when the ATRBAC policy contains rules that cannot fire. There are a few different ways that this can be the case for a rule. Forward pruning has been shown to greatly reduce the policies created in [2]. We see an increase in performance for backwards pruning if there exists low coupling between the set of goal roles and the rules which reference them. Backwards pruning decreases performance if there is high coupling within the policy, as this results in very few roles/rules/timeslots from being removed.

A performance increase from abstraction refinement is only possible for UNSAFE ATRBAC-safety policies. This due to the one sided error that abstraction refinement implements. If the policy is assumed SAFE, then this feature should be turned off to increase performance. Parallelizing abstraction refinement would mitigate the negative effects on performance for SAFE policies, but this feature was not implemented in Cree due to time constraints.

The performance considerations for bound estimation rely on the optimizations of the ATRBAC-Safety solver being used. Our bound estimation produces an upper bound which can be exponential in the size of the input. The performance effects of bound estimation relies on the ingenuity of the solver. Empirically, we have noticed that NuSMV [29], when running in bounded model checking, is much faster than its symbolic model checking variant only when given our upper bound. If only using Tightening 1, we have found that symbolic model checking was quicker on average.

6 Empirical Assessment

We have implemented Cree, and made it available for public download [28]. In this section, we discuss an empirical assessment we have conducted of Cree, in comparison to five prior tools for ATRBAC-safety to which we have access. Our intent with such an empirical assessment is to ask whether Cree's design and implementation does indeed result in a performant tool when compared to prior tools.

We have conducted empirical assessments on the three benchmark classes from prior work [6]. Our results are shown in Figure 8, Figure 9, and Figure 10. The curves interpolate the average of 5 runs. The error-bars show the standard deviation from the average. The red wavy lines represent the point where a tool is unable to solve the rest policies due to timing out or crashing.

Benchmark Class (a) in Figure 8, first presented by [2] but altered here, are randomized test-cases where:

- Roles subplot: Rules are fixed at 200 and Timeslots at 20.
- Rules subplot: Roles are fixed at 200 and Timeslots at 20.
- Timeslots subplot: Rules and Roles are fixed at 200.

Everything about the rules created in the Benchmark Class (a) rules are randomized: • Random start and end times for the administrator time-interval, where start ≤ end time. • For every role that exists there is a $\frac{1}{5}$ chance that it will be added to the rule's precondition as a positive role condition, and $\frac{1}{5}$ chance for a negative precondition. These factors differ from the original code in [2] where there was an equal probability of $\frac{1}{3}$ for each case. The change is to reduce the number of role preconditions is to allow for more rules that have zero preconditions, and thus are allowed to be executed. Without rules with empty preconditions the query will always be unreachable and thus a safe system. • The target role is randomized. • The target time-interval is a set of time-slots and there is a $\frac{1}{2}$ probability of a time-slot being added to the "role-schedule". • The type of rule is randomized with a $\frac{1}{2}$ probability for *t_can_assign* or *t_can_revoke*. • The administrator is "TRUE".

Benchmark Class (b) in Figure 9, presented by [14], are ARBAC policies that have "temporality" randomly added to them. We would like to thank Ranise et al. for providing these test-cases for us to use. This set of 13 policies all have "TRUE" as the administrator and only contain *t_can_assign* and *t_can_revoke* rules.

Benchmark Class (c) in Figure 9, presented by [14], is generated similarly to Benchmark Class (b) but these rules allow for arbitrary administrator roles.

In Figure 10, we present testcases taken from [4], which are ARBAC policies and we convert them to ATRBAC policies by introducing 1 time-slot and having every rule associate with that time slot. We first converted this set in our prior wrok [6]. Given an example ARBAC rule: $\langle a, C, t\rangle$, we can convert it with a single time-slot ts such that the policy still reflects it's original guarantees on safety: $\langle a, ts, C, ts, t\rangle$. The Mohawk test-cases are split into 3 complexity classes: polynomial time, NP-Complete, and PSPACE-Complete. This reflects what is contained within the test-cases:

- Polynomial Time: *can_assign* and *can_revoke* rules where the administrator is "TRUE", only positive preconditions for *can_assign* rules or "TRUE", and *can_revoke* rule's preconditions are "TRUE".
- NP-Complete: *can_assign* rules where the administrator is "TRUE" and preconditions can include positive or negative roles, or be "TRUE".
- PSPACE-Complete: *can_assign* and *can_revoke* rules where the administrator is "TRUE" and preconditions can include positive or negative roles, or be "TRUE".

For Benchmark Class (a), Figure 8, Cree is within 0.5 seconds of the tools TredRole, TredRule, and AsaspTime-Sa and outperforms our prior tool Mohawk+T by almost 6 seconds for the bigger policies. AsaspTime-Nsa was unable to run most these tests. Cree's Polynomial Time Solving when possible is the factor that reduces the run time down to comparable times to the other software as we are able to skip the expensive overhead of running the model checker.

For Benchmark Class (b), Figure 9, all tools, except Asasp Time-Nsa and AsaspTime-Sa, solve the policies within 1 second. This is the only instance where Mohawk+T outperforms Cree, but the amount of the order of 10 milliseconds.

For Benchmark Class (c), Figure Figure 9, Cree either performs the quickest or is tied with the other fastest tool AsaspTime-Nsa. The only exception to this is the policy Hospital 4, where Cree performs the worst overall. Note

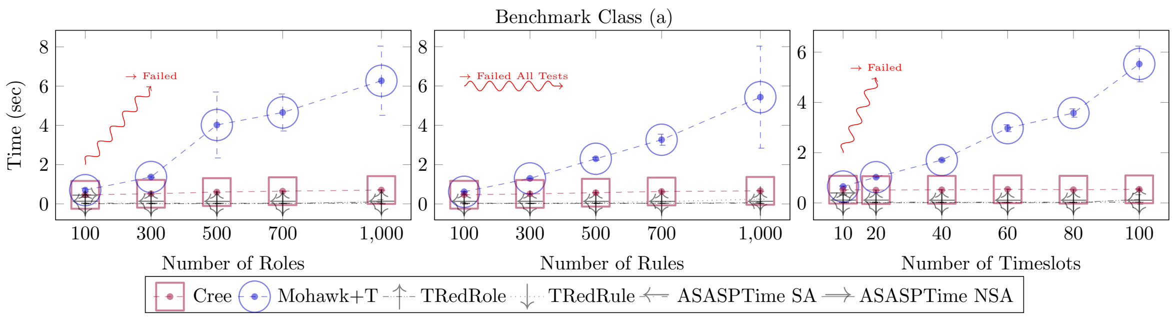


Fig. 8: Results on all tools for Benchmark Class (a). It comprises random input instances from a generator from Uzun et al. [2]. The curves interpolate averages, and the error-bars show the standard deviation.

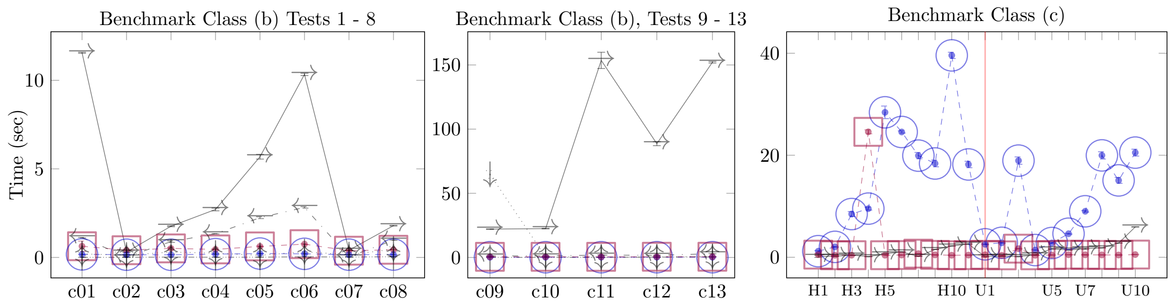


Fig. 9: Results for Benchmark Class (b) (two graphs to the left), and Benchmark Class (c) (right). These comprise input instances from the work of Ranise et al. [14].

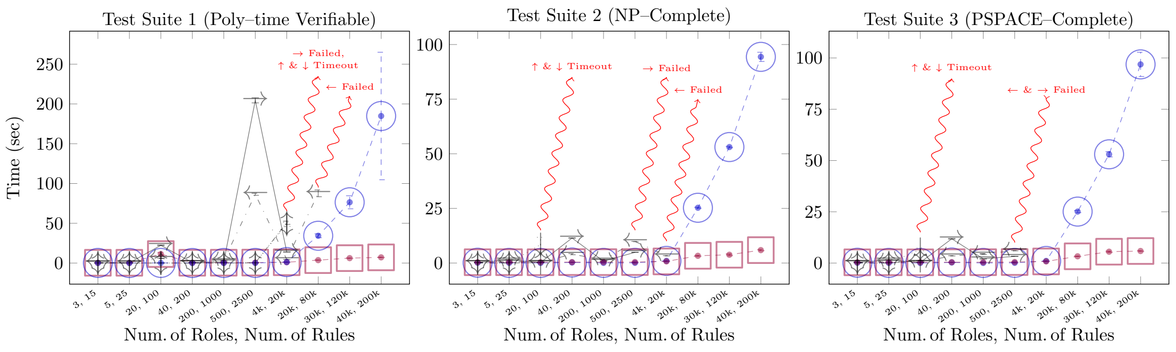


Fig. 10: Mohawk inputs [4] converted to ATRBAC-safety instances using one time-slot. Test Suite 1 are inputs with non-negated preconditions only. Test Suite 2 are inputs with no revoke rules. Test Suite 3 are both positive and negated preconditions, and assign/revoke rules. Red wavy lines are tools that crashed/timedout during testing for certain input sizes.

that as in prior work [14], we did not try this Benchmark Class on the TREDROLE and TREDRULE tools.

For the Mohawk inputs, Figure 10, Cree significantly outperforms all the existing tools. Furthermore, besides Mohawk+T, the existing tools are unable to withstand the input instances from Mohawk [4] beyond a certain threshold. For the polynomial-time verifiable sub-class, for example, which is Test Suite 1, none of the existing tools were able to handle inputs beyond 20,000 roles and 80,000 rules.

Cree is no worse than any of the prior tools for any input we tried. Furthermore, for each of the prior tools, there exists an input for which Cree is strictly better. Mohawk+T is strictly worse than Cree for Benchmark (a), Benchmark(c), and the Mohawk inputs. ASASPTIME-SA is strictly worse than Cree for input Benchmark (b). For the tests 9-13, ASASP TIME-SA performs 0.5 to 3 seconds slower than Cree. ASASP TIME-NSA is strictly worse than Cree for the Mohawk inputs. TREDROLE and TREDRULE perform much faster than Cree in almost all instances it was able to handle, but Cree is able to solve much larger policies without crashing or timing out.

7 CONCLUSIONS

We have proposed a new approach and corresponding tool which we call Cree, for addressing safety analysis in the context of Administrative Temporal Role-Based Access Control (ATRBAC). ATRBAC introduces new features, and therefore technical challenges for safety analysis: support for time-intervals in policies, and rules for enabling and disabling roles. In Cree, we reduce the problem to model checking and then leverage an existing model checker, NuSMV. In addition, we incorporate several heuristics for performance: Polynomial Time Solving when possible, forward and backward pruning, abstraction-refinement, and bound-estimation. While these heuristics are inspired by prior work, our algorithms are customized for the specific technical challenges that ATRBAC introduces. We have conducted a thorough empirical assessment in which we compare Cree to five prior tools. Cree is no worse than any of the other tools across all inputs we tried, and outperforms every other tool for at least one of the input cases.

Acknowledgments

We thank the creators of the prior tools [2], [14] for making their tools available to us and helping us with their use. We thank also Ranise et al. [14] for making all of their inputs from their empirical assessment available to us.

References

[1] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, Aug. 1976. [Online]. Available: http://doi.acm.org/10.1145/360303.360333

[2] E. Uzun, V. Atluri, S. Sural, J. Vaidya, G. Parlato, A. L. Ferrara, and M. Parthasarathy, "Analyzing Temporal Role Based Access Control Models," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 177–186. [Online]. Available: http://doi.acm.org/10.1145/2295136.2295169

[3] A. Jones, "Protection mechanism models: their usefulness," *Foundations of secure Computation*, pp. 237–252, 1978.

[4] K. Jayaraman, M. Tripunitara, V. Ganesh, M. Rinard, and S. Chapin, "Mohawk: Abstraction-refinement and bound-estimation for verifying access control policies," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 4, pp. 18:1–18:28, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2445566.2445570

[5] R. S. Sandhu, "The typed access matrix model," in *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, ser. SP '92. Washington, DC, USA: IEEE Computer Society, 1992, pp. 122–.

[6] J. Shahen, J. Niu, and M. Tripunitara, "Mohawk+t: Efficient analysis of administrative temporal role-based access control (atrbac) policies," in *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '15. New York, NY, USA: ACM, 2015, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2752952.2752966

[7] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed nist standard for role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, Aug. 2001.

[8] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The arbac97 model for role-based administration of roles," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, pp. 105–135, Feb. 1999. [Online]. Available: http://doi.acm.org/10.1145/300830.300839

[9] A. Sasturkar, P. Yang, S. D. Stoller, and C. R. Ramakrishnan, "Policy analysis for administrative role based access control," *Theoretical Computer Science*, vol. 412, no. 44, pp. 6208–6234, Oct. 2011.

[10] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough, "Towards formal verification of role-based access control policies," *Dependable and Secure Computing, IEEE Transactions on*, vol. 5, no. 4, pp. 242–255, Oct 2008.

[11] 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*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 46–49. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00768-2_4

[12] E. Bertino, P. A. Bonatti, and E. Ferrari, "Trbac: A temporal role-based access control model," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 191–233, Aug. 2001.

[13] *Seconds Since the Epoch*. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_16

[14] S. Ranise, A. Truong, and A. Armando, "Scalable and Precise Automated Analysis of Administrative Temporal Role-based Access Control," in *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '14. New York, NY, USA: ACM, 2014, pp. 103–114. [Online]. Available: http://doi.acm.org/10.1145/2613087.2613102

[15] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *Proceedings 1996 IEEE Symposium on Security and Privacy*, May 1996, pp. 164–173.

[16] A. J. Lee, K. E. Seamons, M. Winslett, and T. Yu, *Automated Trust Negotiation in Open Systems*. Boston, MA: Springer US, 2007, pp. 217–258.

[17] P. Rajkumar and R. Sandhu, "Safety decidability for pre-authorization usage control with finite attribute domains," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 05, pp. 582–590, sep 2016.

[18] P. V. Rajkumar and R. Sandhu, "Safety decidability for pre-authorization usage control with identifier attribute domains," *IEEE Transactions on Dependable and Secure Computing*, 2018.

[19] A. L. Ferrara, P. Madhusudan, and G. Parlato, "Policy analysis for self-administrated role-based access control," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 432–447.

[20] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, "Efficient policy analysis for administrative role based access control," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 445–455.

[21] N. Li and M. V. Tripunitara, "Security analysis in role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 4, pp. 391–420, Nov. 2006.

[22] P. Ammann, R. J. Lipton, and R. S. Sandhu, "The expressive power of multi-parent creation in monotonic access control models," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 149–166, 1996. [Online]. Available: https://doi.org/10.3233/JCS-1996-42-303

[23] M. Tripunitara and N. Li, "A theory for comparing the expressive power of access control models," *Journal of Computer Security*, vol. 15, no. 2, pp. 231–272, Feb. 2007. [Online]. Available: http://doi.org/10.3233/JCS-2007-15202

[24] E. Uzun, V. Atluri, J. Vaidya, S. Sural, A. Ferrara, G. Parlato, and P. Madhusudan, "Security analysis for temporal role based access control," *Journal of Computer Security*, vol. 22, pp. 961–996, 06 2014.

[25] S. Ranise, A. Truong, and L. Viganò, "Automated analysis of rbac policies with temporal constraints and static role hierarchies," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: ACM, 2015, pp. 2177–2184.

[26] A. Truong and D. H. T. That, "Solving the user-role reachability problem in arbac with role hierarchy," in *2016 International Conference on Advanced Computing and Applications (ACOMP)*, Nov 2016, pp. 3–10.

[27] S. Ranise, A. Truong, and L. Viganò, "Automated and efficient analysis of administrative temporal rbac policies with role hierarchies," *Journal of Computer Security*, no. Preprint, pp. 1–36, 2018.

[28] J. Shahen, "Cree: Source Code and Supplementary Material," https://ece.uwaterloo.ca/~jmshahen/cree/, Jan 2019.

[29] "NuSMV," http://nusmv.fbk.eu/, Jun 2016.

Jonathan Shahen received his MASc and BASc in Computer Engineering from the University of Waterloo, in Canada, where he is currently working towards his PhD. He researches information security and machine learning.

Jianwei Niu is a Professor and the interim Chair of Computer Science at the University of Texas-San Antonio. She researches formal methods to improve software dependability, and has contributions to security and privacy policies, authorization decision engines, and enforcement mechanisms. Her research has been supported by numerous grants from NSF, NHARP, Microsoft and the NSA.

Mahesh Tripunitara is a Professor in the Electrical and Computer Engineering (ECE) Department at the University of Waterloo, Canada. He researches various aspects of information security. His work, with students, has won paper-awards at the ACM Symposium on Access Control Models and Technologies 2013 and 2015, and the Usenix Security Symposium 2013.