Ambit: Verification of Azure RBAC

Matija Kuprešanin
km200328d@student.etf.bg.ac.rs
University of Belgrade
Belgrade, Serbia

Pavle Subotić
pavlesubotic@microsoft.com
Microsoft
Belgrade, Serbia

ABSTRACT

In this paper, we present an access control verification approach for Role-Based Access Control (RBAC) mechanisms. Given a specification that models security boundaries (e.g., obtained from a threat model, best practices etc.), we verify that a change to an RBAC state adheres to the specification (i.e., remains within the security boundaries). We demonstrate the practical utility of our approach by instantiating it for Microsoft's Azure AD. We have realized our technique in a tool called Ambit which leverages SMT (Satisfiability Modulo Theory) solvers to efficiently encode and solve the resulting verification problem. We demonstrate the scalability and applicability of our approach with a set of generated benchmarks that attempt to simulate real-world RBAC configurations.

CCS CONCEPTS

• **Security and privacy → Domain-specific security and privacy architectures**; • **Theory of computation → Program reasoning**.

KEYWORDS

RBAC, SMT, Verification

ACM Reference Format:
Matija Kuprešanin and Pavle Subotić . 2023. Ambit: Verification of Azure RBAC. In *Cloud Computing Security Workshop (CCSW '23), November 26, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605763.3625200

1 INTRODUCTION

Access management for cloud resources is a critical function for any organization that is using the cloud. Organizations are dynamic hierarchical entities comprising of different users, roles, and resources. Thus, managing how the vast volumes of sensitive data are safely accessed is a well-established cloud security challenge.

To effectively manage access control at scale, system designers have resorted to software defined permissions as opposed to using static access matrix entries etc. Defining access control as code allow users to define expressive rules using various domain specific languages that define *constraints* on the interaction between users, actions and resources.

CCSW '23, November 26, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0259-4/23/11…$15.00
https://doi.org/10.1145/3605763.3625200

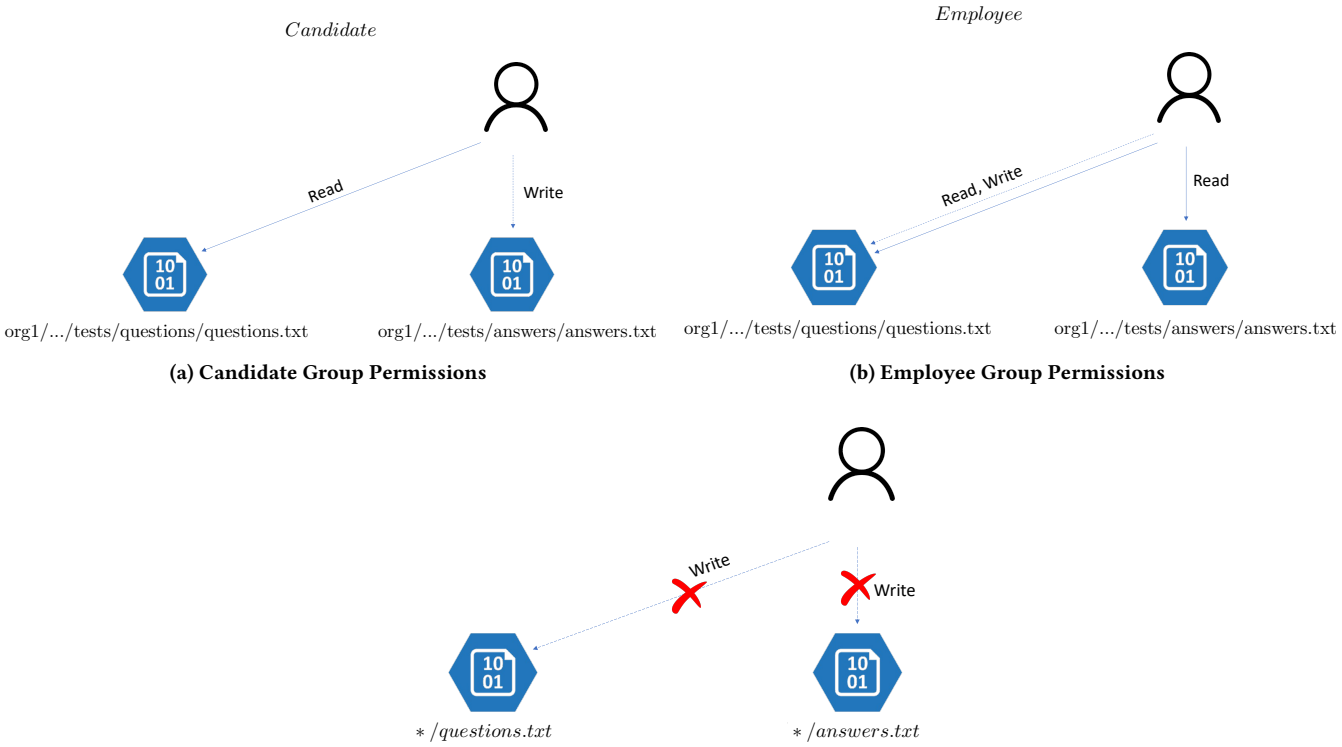
A common access management scheme is Role-Based Access Control (RBAC). Here an access control management system stores action permission constraints (roles) between users and resources. Thus constraints determine what actions users can perform on what resources.

While access control management aids in improving an organization's security, the mere existence of access controls does not automatically ensure all security rules are enforced. Rules embodied in roles existing are typically time bound, however they do not model invariant *security specifications* e.g, best practices, or threat models, that transcend roles and encode fundamental security boundaries of an organization. The dynamic nature of RBAC e.g., adding/removing roles, attaching to groups, as well as the intrinsic hierarchical nature of organizations and their resources, makes it difficult to guarantee that these invariant specifications will be adhered to. This has been highlighted by several notable security breaches. For instance, a recent investigation by CyberArk [2] found that millions of Azure blobs were exposing sensitive data including personal identifiable information, personal health records, financial data, invoices, contracts among others. The study highlights that misconfigurations by organizations are a real threat and raises the question:

"Can we provide tooling to customers to help them avoid RBAC misconfigurations?"

Example 1.1 (Motivation). Consider a simple interviewing scenario presented in Figure 1. We depict two groups: a *candidate* in Figure 1a and an *employee* in Figure 1b. A candidate is assigned roles that allow for reading (full line) questions and writing (dotted line) the answers to the questions. And an employee has roles that allows for reading and writing questions and reading answers. Suppose we create a group for internal candidates. Since an internal candidate is an both employee and a candidate, the internal candidate inherits both employee and candidate roles. However, as a general organization security measure, we never (regardless of the user) want to allow someone to be able to write both questions and answers as depicted in the threat model in Figure 1c. However, the internal candidate group breaks this threat model. Thus, when we add a role assignment for the employee or candidate groups we implicitly violate the threat model via the internal candidate group and put the RBAC system in an unsafe state.

While Example 1.1 depicts a simple scenario, even in this case it is not unreasonable to expect a threat model violation to occur due to the implicit inheritance of roles. This is further exacerbated for larger organizations. Due to complex organization structures e.g., user group hierarchies and dynamically changing structures and permissions, manual inspection and detection is difficult and often infeasible. Thus, there is considerable utility in automating the validation process with formal guarantees.



(a) Candidate Group Permissions

(b) Employee Group Permissions

(c) General Threat Model Specification

Figure 1: Violating RBAC State

In this paper, we propose a technique that uses automated reasoning to verify the state of a RBAC system given an event (e.g., adding a role assignment) and a specification. A slightly simplified conceptual view of our approach can be summarised as a constraint graph as shown in Figure 2. Here the y-axis denotes the set of all principals and the x-axis the set of all scopes (resources). For simplicity, we assume a single action (to avoid a third dimension). Graphically, a specification defines invariant regions that allow (blue-diagonal-lines) or do not allow (red-dotted) an action for the set of principals and scopes. Similarly, the state of a RBAC system can be mapped. Let the state be denoted by a green-checkered shape. For a state to be safe, it must be disjoint from any disallowed region and within an allowed region. In Figure 2a, we depict a scenario when the state is safe i.e., does not cross any security boundaries. In contrast, Figure 2b depicts a scenario when another assignment is added to the scenario in Figure 2a and results in an unsafe state i.e., a security violation.

Our technique works as follows: when the state of an RBAC is due to be changed, we perform a *what-if* analysis to warn the user if the change can put the access control configuration in a bad state. To perform this analysis at scale, we first prune the search space by determining the *impact* a given event has and generate a *static snapshot* that comprises of a reduced RBAC state. This static snapshot is then encoded into a logical formula. We leverage automated theorem provers such as Microsoft's Z3 [7] to efficiently verify adherence to security specifications.

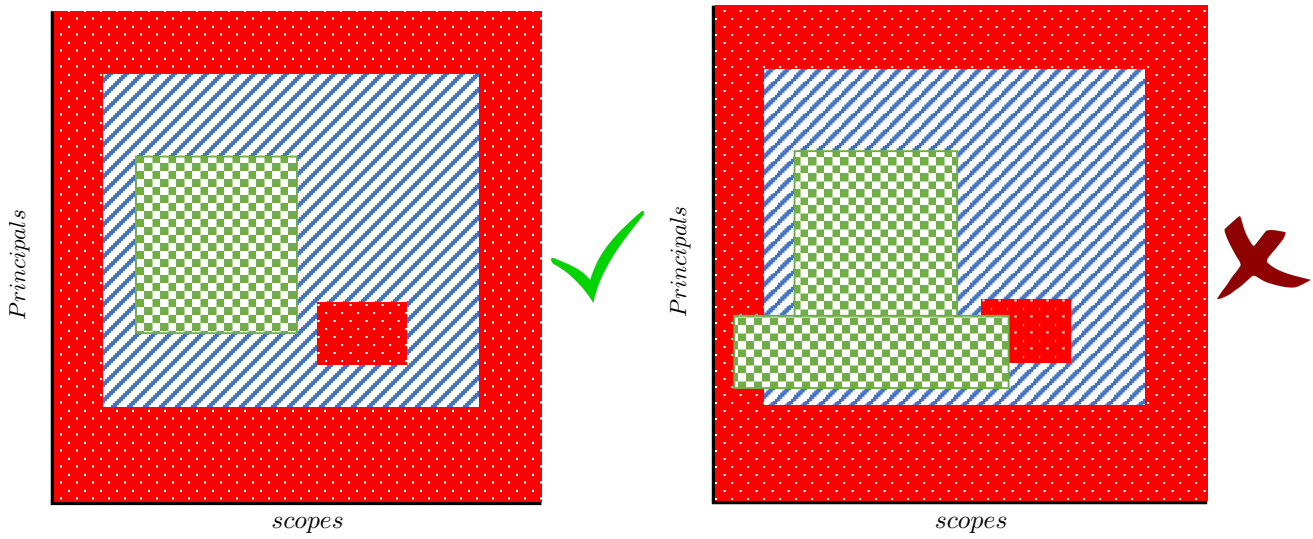
We have implemented our technique in a tool called Ambit that is specialized for Microsoft Azure AD. Ambit aims to aid customers in avoiding misconfigurations by allowing organizations to define *specifications* which encode best practices, threat models etc. and warning them of events that will not adhere to the specification. As a result, customers are aware of breaking changes as soon as possible and can avoid putting the RBAC state to an unsafe state.

We have evaluated Ambit on 1000 automatically generated benchmark scenarios where it exhibits a geometric mean runtime of 1.29 seconds to perform a verification.

We summarize our contributions as follows:

(1) A verification method for RBAC systems that allows users to specify security boundaries and verify RBAC state changes do not cross these boundaries
(2) An implementation of our technique in the tool Ambit
(3) A preliminary evaluation to show the scalability of Ambit on a set of 1000 generated synthetic benchmarks

We structure the remainder of the paper as follows: In Section 2 we provide a background of Azure AD RBAC. In Section 3 we give a technical description of our approach. In Section 4 we provide a preliminary evaluation of our technique. In Section 5 we contrast our work with related works in the literature and we present some future work in Section 6 and conclude in Section 7.



(a) Safe state (b) Unsafe state

Figure 2: Security Boundaries and RBAC State: the y-axis denotes the set of all principals and the x-axis the set of all scopes. The specification allow region is define in the blue-diagonal-lines area and disallow regions in red-dotted area, the RBAC state is denoted by the green-checkered area.

2 BACKGROUND

In this section, we describe RBAC with an emphasis on Azure AD, Microsoft Azure's RBAC service. We note that this formalization largely corresponds to standard definitions of RBAC, aside from some nomenclature specific to Azure.

A note on the notation of this paper. With a slight abuse of notation, we denote a set of elements of type x as $\{x\}$. This is just a syntactic marker that $\{x\}$ is a set of elements of type x where x has been defined previously. Where there is ambiguity on if we are discussing a set with a singleton element or a name of a set, we clarify this. We also denote access of sub elements with a . operator, so that if x has a subelement y we can access y from x using $x.y$.

Azure AD's RBAC governs how users interact with resources by assigning roles to users that assert what permissions a user has on a resource. Thus RBAC maps a *security principal* which represents users, groups of users, applications etc. to Azure resources (scope), for a set of *role definitions* that contains *actions* that can be performed on the resources by the principal.

A principal $p \in P$ is a string identifier. Principals adhere to an organizational hierarchy and thus are partially ordered. We define an ordering $\sqsubseteq_P$ where $p \sqsubseteq_P p'$ iff p *is a* p'. For example, *Ann Doe* $\sqsubseteq_P$ *full time employee.*

A scope $s \in S$ is a string path that specifies a scope. It has the form $mg/sub/rg/rsc$ where mg (management group), sub (subscription), rg (resource group) and rsc (resource) are all strings. We define a $\sqsubseteq_S$ ordering where if $s \sqsubseteq_S s'$ then s' is a prefix of s. For example, $test1/t/abc \sqsubseteq_S test1/t/$.

We define a role $r \in R$ such that $r = \langle \{a\}, \{\tilde{a}\}, s \rangle$ where $\{a\}$ is a set of *actions*, $\{\tilde{a}\}$ is a set of *notActions* and s is the assignable scope. The set of actions allowed by a role is equal to $\{a\} \setminus \{\tilde{a}\}$. Actions are of type *wcString*. We define an accessor notation on a role such that $r.a$ accesses actions and $r.\tilde{a}$ accesses notActions. $r.s$ accesses the assignable scope of the role. Note, we don't model the semantics of each action and simply interpret them in the domain of wild card strings (*wcString*).

An RBAC state $\sigma \in \Sigma$ is a set of role assignments. An RBAC event $e \in E$ is a user initiated event such as adding a role assignment, removing a role assignment (and by extension modifying a role assignment). Given an existing RBAC state σ an event transitions the RBAC system into a new state σ', we denote this transition $\sigma \rightarrow_e \sigma'$.

A *role assignment* assigns a role to a principal at a scope thus attaching the permissions in the role to the principal for the resources in the scope. We note that role assignments are *transitive* for groups which means that if a user is a member of a group and that group is a member of another group that has a role assignment, the user will have the permissions in the role assignment as well i.e., roles flow downwards in the $\sqsubseteq_P$ hierarchy. For example, if a principal p is a subgroup of principal p' then p inherits all the roles of p'.

We define a role assignment $\lambda \in \Lambda$ as a triple $\langle p, r, s \rangle$ where $p \in P$ is a principal, $r \in R$ is a role and $s \in S$ is a scope . We say an assignment is *valid* iff $r.s \sqsubseteq_S s$.

In order to support a wide range of customer use cases, Azure AD RBAC allows for broad expressivity and extensibility. In particular, it supports an additive role assignment model which allows for a complex set of overlapping role assignments. This is further enhanced by the implicit inheritance of roles for principals.

Example 2.1 (Azure AD Role Assignments). Below we show how Azure AD Role assignments are described in JSON format. Here we represent role assignments by defining a principal identifier, a

role identifier and a scope. We omit auxiliary data irrelevant for our verification problem.

```
{
    "principalId": "<InternalCandidateGroupID>",
    "roleDefinitionId": "<ReadWriteRoleID>",
    "directoryScopeId": "org1/.../tests/questions
    ...
}
```

Similarly, we present an example role definition below. The role definition is stored in Azure AD and attached to a principal and scope. Again, we omit auxiliary data irrelevant for our verification problem.

```
{
  "id": "<ReadWriteRoleID>",
  "permissions": [
    {
      "actions": [
        "Read",
        "Write"
      ],
      "notActions": [],
      ...
    }
  ],
  ...
}
```

3 TECHNICAL DESCRIPTION

In this section, we provide a technical description of our technique that is realized in the tool Ambit.

3.1 Overview

In Figure 3, we provide an overview of our technique. In the proceeding subsections we expand on each component. Ambit can act both as a proxy to an RBAC system or operate internally to the RBAC system. Figure 3 describes the former approach, where role assignment commands are issued to RBAC via Ambit. If the role assignment does not put the RBAC in a bad state, it is forwarded and otherwise, an error warning is issued.

Once a role assignment event is given, the *specializer* component uses it along with the property and the existing RBAC state to generate a *static snapshot*. To do this, the specializer component queries the RBAC system in such a way as to only obtain relevant information with respect to the property and role assignment. Next, the *encoder* component takes a static snapshot and translates into a mathematical formula that is given to an SMT solver e.g., Z3. The SMT solver soundly solves the mathematical model and returns the satisfiability or unstatisfiability of the formula, that determines whether the assignment should proceed or the user should be warned.

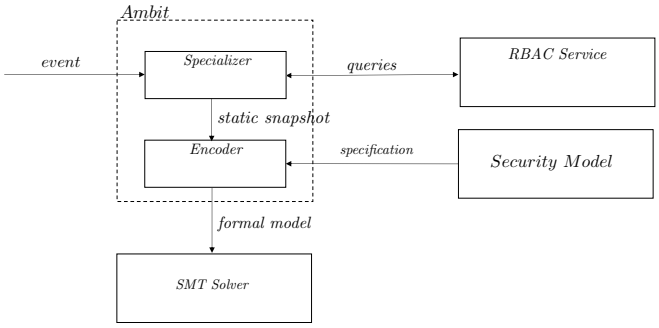


Figure 3: Overview of Ambit

3.2 Specification

The specification defines the potentially infinite regions on the dimensions of principals, actions or non-actions and scopes connected by connectives by combining atomic specifications with logical connectives.

We define a specification Φ to be a set of sets of atomic specifications ϕ, where we define atomic specifications as $\varphi = \langle p, \langle\{a\}, \{\tilde{a}\}\rangle, s, n\rangle$ and where $\{a\}$ and $\{\tilde{a}\}$ are actions and notActions as found in static snapshots. However, unlike static snapshots, $p, s \in wcString$ and an additional negation flag n is included to allow atomic specifications to be negated. Similar to roles we define accessors on all sub-components of an atomic specification e.g., $sp \in \mathcal{S}$, $sp.p$ denotes the $p \in wcString$ sub-component of specification sp.

Example 3.1 (Motivating (Cont.)). Given our role assignment and roles in the motivating example, we define a specification in JSON for Figure 1c. Our specification says: *any principal cannot write both answers and questions*. We first define atomic specifications PID1 and PID2:

```
{
  [
  {
      "formulaId": "PID1"
      "principal": "*",
          ...
      "actions": ["Write"],
          ...
      "scopes": "*/answers.txt",
      "negated": true

  }
  ...
  ]
}
...

{
  [
  {
      "formulaId": "PID2"
```

```
        "principal": "*",
            ...
        "actions": ["Write"],
            ...
        "scopes": "*/questions.txt",
        "negated": true
    }
    ...
    ]
}
```

Next, we this define our final specification the JSON:

```
{
    "specs": [["PID1"], ["PID2"]]
}
```

As we show in Section 3.4, this models our threat model in Figure 1c i.e., that we don't want a principal to have write access to both answers and questions.

3.3 Specialization

The naïve approach to performing an RBAC verification is to translate the entire RBAC state along with the role assignment into a verification problem. However, aside from being unscalable, this would not be sound, as it would miss new assignments generated from the inheritance model of RBAC systems. We therefore construct a specializer that generates an expanded and yet pruned RBAC state, excluding irrelevant role assignments and includes newly generated ones from the effects of principal/group inheritance.

The specializer first queries the RBAC system to create a set of role assignments that accounts for principal hierarchies ($\sqsubseteq_P$) and scope hierarchies ($\sqsubseteq_S$). Assuming a role assignment $\langle p, r, s\rangle$ and an RBAC state σ, this process traverses all assignments in σ that have a principal p' such that $p \sqsubseteq_P p'$. From these assignments any that do not have a scope s' such that $s \sqsubseteq s'$ are further discarded. All of them are added to the snapshot, with their principals altered to be equal to p.

Next, we collect any assignments that can be impacted by $\langle p, r, s\rangle$. That is, we collect all assignments that have a $p' \sqsubseteq p$ and $s' \sqsubseteq s$. These assignments together with the inherited assignments form a static snapshot.

A static snapshot Π is defined as a set of tuples $\{\pi\}$ where $\pi = \langle p, \langle\{a\}, \{\tilde{a}\}\rangle, s\rangle$ where $p, s \in string$ and $a, \tilde{a} \in \wp(wcString)$

3.3.1 Role Assignment Specialization. We describe the core of our specialization for the addition case in Algorithm 1. Instances for removing a role assignment, changing a role assignment largely follow the same logic with a few minor technical changes.

Initially, we initialize the set *ra_set* with the role assignment that we are adding (line 1). Note, $\{ra\}$ is a singleton set. In the first loop in line 2 we do two things: in the first conditional statement (line 3) we add the role assignments that the principal referred to in *ra* inherits and that are relevant in terms of scope. Note that the assignment is altered to refer to $ra.p$, the principal that inherits the role.

Algorithm 1: SpecializeRA(data, ra)

Data: Data queried from Azure AD: *data*;
The role assignment being added: *ra*
Result: A subset of the currently present Azure AD role assignments set along with the role assignment being added.

```
1  ra_set ← {ra};
2  foreach ass in data.role_assignments do
3      if ra.p ⊑_P ass.p ∧ ra.s ⊑_S ass.s then
4          ra_set ← ra_set ∪ {⟨ra.p, ass.r, ass.s⟩};
5          continue
6      if ass.p ⊑_P ra.p ∧ ass.s ⊑_S ra.s then
7          ra_set ← ra_set ∪ {ass};
8          continue
9  foreach p in data.subprincipalsOf(ra.p) do
10     ra_set ← ra_set ∪ {⟨p, ra.r, ra.s⟩};
11 return ra_set
```

In the second conditional statement (line 6) we add the role assignments that the new assignment *ra* might affect, i.e., those that inherit from its principal on a scope smaller than or equal to the one in the assignment *ra*. In the second loop at line 9 we add to *ra_set* the role assignments that are implicitly created with the addition of *ra* by inheritance. Here it's acknowledged that expanding privileges of a group expands privileges of its members. Finally, at line 11 the result is resulting set of pruned role assignments are returned to be passed to the encoder. Note, $\{\langle ra.p, ass.r, ass.s\rangle\}$, $\{ass\}$ and $\{\langle p, ra.r, ra.s\rangle\}$ are all singleton sets.

3.3.2 Group Assignment Specialization. Additionally, we describe specialization for the case of adding a principal to a group in Algorithm 2. Instances for removing from a group, modifying a group largely follow the same logic with a few minor technical changes.

Algorithm 2 iterates over all role assignments (line 2) to find those that apply to group g. Concretely, if a role assignment *ass* has a principal $ass.p$ such that $g \sqsubseteq_P ass.p$ (line 3), we add the role assignment to the *inh_set*. Then for each of the elements in *inh_set* we perform an RA specialization using Algorithm 1 (line 7). Note, $\{\langle p, ass.r, ass.s\rangle\}$ is a singleton set.

3.3.3 Changing roles. Adding actions to a role definition can be treated as assigning a role that equals the set difference between the new and the old definition to all principals that are currently assigned to he said role. It can therefore be treated as an array of addition cases. Removing actions from a role definition can be treated following using the same logic.

Example 3.2 (RBAC Specialization). Consider the example in Figure 4. Here we have a role assignment $\langle FTE, r', s'\rangle$ and an existing RBAC state containing the role assignments $\langle Emp, r_1, s_1\rangle$, $\langle FTE, r_2, s_2\rangle$ and $\langle Ann\ Doe, r_3, s_3\rangle$. Note, the ordering of the RBAC state elements. The specializer creates additional role assignments for FTE, namely r_1 and r_2 because $FTE \sqsubseteq_P Emp$ and $FTE \sqsubseteq_P FTE$ and $s' \sqsubseteq_S s_1$ and $s' \sqsubseteq_S s_2$. $\langle Ann\ Doe, r_3, s_3\rangle$ is impacted by the role assignment, because $s_3 \sqsubseteq_S s'$, thus we also include this in the static

Algorithm 2: SpecializationP(data, p, g)

Data: Data queried from Azure AD: *data*;
The principal p being added to group g
Result: A subset of the currently present Azure AD role assignments set along with the role assignments being implicitly added by inheritance.

```
inh_set ← {};
foreach ass in data.role_assignments do
    if g ⊑_P ass.p then
        inh_set ← inh_set ∪ {⟨p, ass.r, ass.s⟩};
ra_set ← {};
foreach ra in inh_set do
    ra_set ← ra_set ∪ SpecializeRA(data, ra);
return ra_set
```

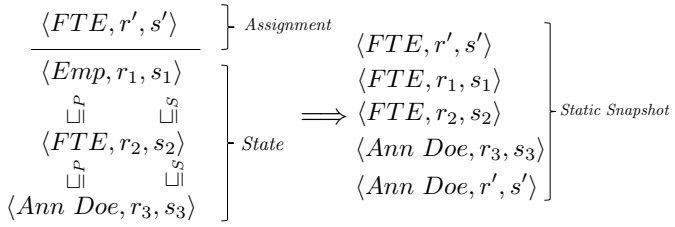


Figure 4: Reduction of State

$$[\![\pi_0 \dots \pi_n \in \Pi]\!] = \bigvee_{0 \dots n} [\![\pi_i]\!] \tag{1}$$

$$[\![\langle p, \langle \{a\}, \{\tilde{a}\} \rangle, s \rangle]\!] = [\![p]\!] \wedge [\![\{a\} \setminus \{\tilde{a}\}]\!] \wedge [\![s]\!] \tag{2}$$

$$[\![a_0 \dots a_n \in \{a\}]\!] = \bigvee_{a_i \in \{a\}} [\![a_i]\!] \tag{3}$$

$$[\![p]\!] = V_p \Leftrightarrow p \tag{4}$$

$$[\![s]\!] = V_s \Leftrightarrow s \tag{5}$$

$$[\![a]\!] = regex(a, V_a) \tag{6}$$

Figure 5: Static Snapshot Semantics

snapshot as this assignment can now potentially violate security properties.

3.4 Logical Encoding

In this section we provide a translation of static snapshots and security specification into a mathematical model. We define a set of semantic functions $[\![\bullet]\!]$ to translate a static snapshot or specification to a mathematical formula. With a slight abuse of notation, we overload $[\![\bullet]\!]$ for various static snapshot and specification sub-structures. In practice we formulae the resulting logic formula in SMT2 format and solve it using the Z3 SMT solver.

In Figure 5 we describe the translation of a static snapshot. Rule (1) simply says that a snapshot can be defined by a union of smaller snapshots. Rule (2) defines a snapshot by converting a tuple to a conjunction of principals, actions and scopes. Rule (3) defines actions as the union of all single actions. Rule (4) defines a free variable for all principals (if there are several principals). Rule (5) does the same for scopes. Rule (6) defines an action as a wild card string.

In Figure 6, the atomic specification translation is defined in similarity to the static snapshot, only that principals and scopes can also be wild card strings, and a specification can be negated. A specification is a DNF formula of atomic specifications, whose JSON format is given in *Example 3.1.*

In Figure 7, we describe the verification problem semantics. We quantify the whole expression over principals in order to add more expressivity, allowing us to solve a wider range of problems, including our motivating example.

Example 3.3 (Motivating Example (Cont.)). We show how our motivating example scenario is translated to logic that is passed on to an SMT solver. First, we define auxiliary symbols:

$$p = \text{“}\langle InternalCandidateGroupID \rangle\text{”}$$
$$r = \text{“}Read\text{”}$$
$$w = \text{“}Write\text{”}$$
$$s_1 = \text{“}org1/tests/pos1/answers.txt\text{”}$$
$$s_2 = \text{“}org1/tests/pos1/questions.txt\text{”}$$
$$s_3 = \text{“}*/answers.txt\text{”}$$
$$s_4 = \text{“}*/questions.txt\text{”}$$

Using the symbols above, we assume a static snapshot Π:

$$\pi_1 = \langle p, \langle \{r, w\}, \{\} \rangle, s_1 \rangle$$
$$\pi_2 = \langle p, \langle \{r, w\}, \{\} \rangle, s_2 \rangle$$
$$\Pi = \{\pi_1, \pi_2\}$$

When translated, using semantics from Figure 5, we get the following logical formula:

$$(V_p \Leftrightarrow p \wedge (regex(r, V_a) \vee regex(w, V_a)) \wedge V_s \Leftrightarrow s_1) \vee$$
$$(V_p \Leftrightarrow p \wedge (regex(r, V_a) \vee regex(w, V_a)) \wedge V_s \Leftrightarrow s_2)$$

Next, we assume a specification $\Phi = \{\{\varphi_1\}, \{\varphi_2\}\}$ where:

$$\varphi_1 = \langle *, \langle \{w\}, \{\} \rangle, s_3, true \rangle$$
$$\varphi_2 = \langle *, \langle \{w\}, \{\} \rangle, s_4, true \rangle$$

When translated, using semantics from Figure 6, we get the following formulae:

$$[\![\varphi_1]\!] = \neg(regex(*, V_p) \wedge regex(w, V_a) \wedge regex(s_3, V_s))$$
$$[\![\varphi_2]\!] = \neg(regex(*, V_p) \wedge regex(w, V_a) \wedge regex(s_4, V_s))$$

which simplifies to:

$$[\![\varphi_1]\!] = \neg(regex(w, V_a) \wedge regex(s_3, V_s))$$
$$[\![\varphi_2]\!] = \neg(regex(w, V_a) \wedge regex(s_4, V_s))$$

$$[\![\langle p, \langle \{a\}, \{\tilde{a}\}\rangle, s, n\rangle]\!] = [\![\langle p, \langle \{a\}, \{\tilde{a}\}, s\rangle]\!] \Leftrightarrow \neg n \quad (7)$$

$$[\![\langle p, \langle \{a\}, \{\tilde{a}\}\rangle, s\rangle]\!] = [\![p]\!] \wedge [\![\{a\} \setminus \{\tilde{a}\}]\!] \wedge [\![s]\!] \quad (8)$$

$$[\![a_0 ... a_n \in \{a\}]\!] = \bigvee_{a_i \in \{a\}} [\![a_i]\!] \quad (9)$$

$$[\![p]\!] = regex(p, V_p) \quad (10)$$

$$[\![s]\!] = regex(s, V_s) \quad (11)$$

$$[\![a]\!] = regex(a, V_a) \quad (12)$$

Figure 6: Atomic Specification Semantics

$$[\![\phi_0 \dots \phi_n \in \Phi, \Pi]\!] = \neg \exists V_p. \neg \bigvee_{0 \dots n} [\![\phi_i, \Pi]\!] \quad (13)$$

$$[\![\varphi_0 \dots \varphi_n \in \phi, \Pi]\!] = \bigwedge_{0 \dots n} \exists V_a, V_s. [\![\Pi]\!] \Rightarrow [\![\varphi_i]\!] \quad (14)$$

Figure 7: Verification Formula Semantics

Table 1: Benchmark State Characteristics

Characteristic	min	mean	max
actionCount	1	34.68	69
groupCount	5	53.142	99
membershipGraphDensity	0.01001	0.0564	0.1
avgRoleSize	3.5	5.5975	7.5
roleCount	1	25.797	49
userCount	5	105.301	199

Example 3.4 (Motivating (Cont.)). Given we have definitions for the atomic specifications and our snapshot, we construct a verification formula according to Figure 7. This yields the following logical formula:

$$\neg \exists V_p. \neg((\exists V_a, V_s. [\![\Pi]\!] \Rightarrow [\![\varphi_1]\!]) \vee (\exists V_a, V_s. [\![\Pi]\!] \Rightarrow [\![\varphi_2]\!]))$$

We use this first-order logic formula (with the theory of regexes) to pass to an SMT solver. Practically, we check

$$\exists V_p. \neg((\exists V_a, V_s. [\![\Pi]\!] \Rightarrow [\![\varphi_1]\!]) \vee (\exists V_a, V_s. [\![\Pi]\!] \Rightarrow [\![\varphi_2]\!]))$$

to report a violation if this formula is satisfiable, or declare the state permissible otherwise. This allows us to provide the model produced by the SMT solver to the user as a witness that provides an example of how the specification can be violated (i.e. the model is a counter-example).

4 EVALUATION

In this section, we present an evaluation of Ambit. We investigate how it performs on various verification problems with varying characteristics.

4.1 Experimental Setup

We use an Mac M1 with 8 GB RAM, and .NET 6.0 Runtime (v6.0.20) with Z3 version 4.8.16 - 64 bit running on a MacOS Monterey version 12.6.7. Ambit is written in approx. 1.5K LOC in the C# language.

Table 2: Benchmark Specification Characteristics

Characteristic	min	mean	max
specificationCount	2	15.119	29
negatedSpecificationCount	0	4.396	9

4.2 Benchmarks

Due to the difficulty of obtaining real world customer data[1], we simulate 1000 verification problems by generating various sized RBAC states and specifications patterns. Namely, for each verification problem, we (pseudo)randomly generate an Azure RBAC state and a set of atomic specifications. Furthermore, we similarly generate a role assignment to be added to that state. We perform the event of adding a role assignment to trigger Ambit and measure the time taken.

Table 1 shows the characteristics of the RBAC states generated. The *membershipGraphDensity* parameter represents the probability of an edge being created in the principal membership graph[2]. *avgRoleSize* represents the average number of actions in roles present in the system. Parameters of the form $*Count$ represent the number of $*$ in the system (e.g. *actionCount* is the number of different actions used in the system). Notably, *user* refers to a principal that has no subprincipals, and *group* refers to one that does.

In Table 2, we summarise the number of atomic specifications in a final specification in a verification problem. The number is split between those that are negated and those that are not.

4.3 Performance Evaluation

In this section, we provide data on how performance of Ambit scales with the size of the system. To avoid information overload by measuring for every characteristic (see Figure 9), we collapse the benchmark characteristics to a single metric that we believe is a fair reflection the syntactic difficulty of the verification problem. For this, we introduce a metric named *size*. It is calculated according to the following formula:

$$\begin{aligned} size = & actionCount + userCount + \\ & (2 + membershipGraphDensity) \cdot groupCount + \\ & 18 \cdot (specificationCount + negatedSpecificationCount) + \\ & avgRoleSize \cdot roleCount \end{aligned}$$

The formula takes into account every significant part of an Azure RBAC state. It acknowledges the complexity inherent to the fact that principals can be grouped, while also taking into consideration that specifications are particularly important to the verification problem as they are not subject to specialization (thus the high coefficient).

In Figure 8, we plot the size metric for verification formulae against runtime (in logarithmic scale).

We observe the specialization phase is negligible in the overall verification runtime. In all instances specialization completed in

[1] Customer security data, even via secondary meta data (element counts, avg. sizes, structure etc.) is highly confidential.

[2] A DAG is a natural way to represent a partial ordering relation such as the principal hierarchy in Azure. While actual Azure RBAC exhibits a somewhat different interface to this information, without loss of generality we will refer to it as such a graph.

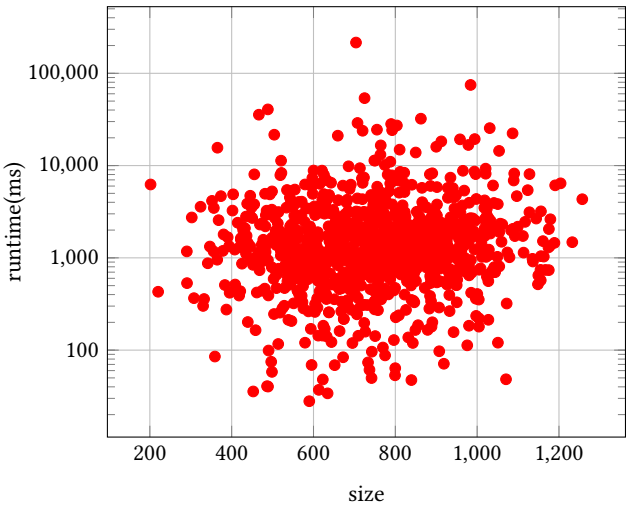


Figure 8: Verification Size Metric vs Runtime

milliseconds, even for the largest benchmark problems. Conversely, the SMT solving phase dominated the runtime.

We attempt to measure how the benchmark characteristics size impact the SMT solving runtime. As shown in Figure 9, the performance data does not correlate well with our single metric, with outliers occurring both on smaller and larger benchmarks. However, we notice a slight tendency for the frequency of outliers to increase with the *size* metric. It is however a well known problem that the syntactic complexity of a SAT or SMT problem does not necessarily correlate with runtimes [6, 12].

In any case, the vast majority of our benchmarks can be verified that it adheres to or violates the specification within a few seconds. More specifically, we exhibit runtime with a geometric mean of 1.29 seconds. For all 1000 benchmarks Ambit returned the correct results.

Overall, we believe that the results are promising for our use case: that Ambit can be used in an interactive manner be it as an internal component in Azure AD or as an external tool that interfaces Azure AD.

5 RELATED WORK

There is a large body of work on the verification of access control policies. The majority of techniques attempt to verify that a policy cannot eventually arrive at an insecure state. For instance, the technique in [20] uses event calculus, techniques such as [9, 16, 18] reduce the RBAC verification to program analysis and use techniques such as abstract interpretation or counter example guided model checking to perform the verification. As described in [13] the general problem of access control is undecidable yet decidable for restricted cases. We indeed take a more pragmatic approach and tackle a decidable fragment. Our formulae for describing the RBAC state and security boundaries is non-recursive and allows a decidable fragment of wild card strings. Thus, our problem can be formulated in a logic whose decision procedure is in the NP-complete class. With the advances of modern SMT solvers, that have been shown to scale to industrial problem [3, 5] we can thus leverage these solvers to verify our problem: given an event, its effect won't cause the RBAC system to be unsafe. In the remained of the related work we focus similar approaches.

In a similar spirit to Ambit, Zelkova [4] translates access control configurations to a SMT formula. However, Zelkova limited to Amazon IAM polices that are static descriptions [1] of a security state snapshot and property. In contrast, RBAC systems have an existing state that must be included in the verification problem. In that sense the verification problem in [4] is different from our work and thus, we require a specialization stage. Moreover, Zelkova does not model principal (users and groups) and resource hierarchies which are common place in RBAC systems. For Azure AD access control this is fundamental and a dynamic RBAC system must be modelled. Lastly, Zelkova only handles stateless logic specifications (the term stateless logic as in [17, 19] with respect to the Craig Interpolation problem). We support a limited stateful logic as required for specifications such as the one in our motivating example. Such a property cannot be enforced by Zelkova.

The technique in [14] transforms XACML policies into Boolean satisfiability problems and use a SAT solver to check partial orders between policies using a bounded analysis. As the case with Zelkova these policies are static and have non-dynamic semantics. Moreover, the encoding to SMT is not sound due to it being bounded. Ambit on the other hand provides a sound encoding.

Similar can be said about Margrave [10], which supports both property driven analysis and change-impact analysis for XACML policies. Like [4] and [14] it does not model principal and resource hierarchies.

SecGuru tool [15] compares Microsoft Azure network connectivity policies using the SMT theory of bit vectors. While supporting Azure, Ambit focuses on a somewhat orthogonal use case, namely, role-based access control.

The approach described in [8], as [4], is limited to comparison between policies in terms of permissiveness in order to facilitate detecting misconfigurations. Notably, among others, it supports Azure policies.

6 FUTURE WORK

Our current implementation of Ambit is preliminary and we plan to integrate it with Azure AD RBAC in the near future so that we can evaluate its utility and performance on real world customer data

Specification expressivity. We have attempted to balance the expressivity and intuition of specifications so that users can limit confusion with specification logic. To do this, we have tried to make our atomic specifications correspond to areas in a graph e.g., Figure 2. On the other hand, we limit the ability to express all possible specifications that can be expressed in first order logic. While we allow the specification to reference two different scopes, the same cannot be done directly for two different principals. To do so we would need to modify the semantics in Figure 7, to change the scope of the existential quantifier that bounds the V_p variable. However, we plan to continue exploring the merits of different expressivity and also tooling to help users visualize both the specification and verification findings.

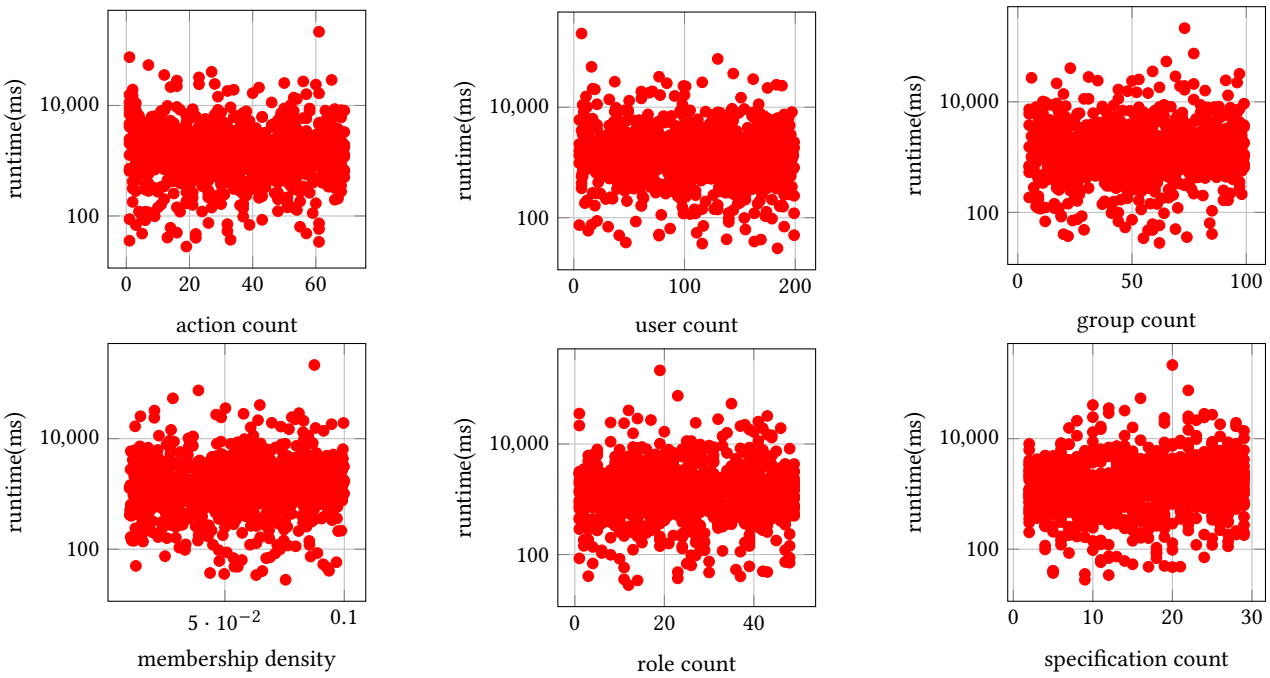


Figure 9: Single Dimension Performance Evaluation

Increasing scalability. We would like to investigate the performance of Ambit on larger instances and investigate performance improvements. For example, performing incremental verification [21] and partial evaluation [11].

Resolving existing misconfigurations. Ambit can also be of utility in a scenario where an erroneous configuration is already present in the system. To detect an already unsafe RBAC instance (i.e. make it comply with a specification) it needs to be iteratively reconstructed, running Ambit throughout the process. Whenever a violation is detected, a model will be returned, hinting at what the violation is caused by. This is a secondary use case for which no benchmarks were run, however it is expected that this is a demanding task. Still, this is likely a one-time cost, and the process can be done off-line.

Support for ABAC constraints. At the time of writing, Azure AD is currently in the process of supporting additional expressivity such as ABAC constraints among other features. We would like to include these features when they become available to the public.

Other Use Cases. We believe our approach can be used for other use cases. For example, an auxiliary use case for Ambit is the ability to keep the role assignments in an RBAC system non-redundant i.e., given a set of role assignments Π_1 and Π_2, then Π_1 is redundant iff: $[\![\Pi_1]\!] \wedge [\![\Pi_2]\!] \Leftrightarrow [\![\Pi_2]\!]$.

7 CONCLUSION

We have presented Ambit, a verifier for Azure AD based access control. Given a specification that defines security boundaries, Ambit can mathematically verify if an event on an RBAC state will be safe or unsafe. Through our evaluation of 1000 generated verification problems, Ambit was able to verify all the problems accurately in a geometric mean time of 1.29 seconds.

While access control verification has been applied to various access policy languages and services, to the best of our knowledge we are the first to propose this form of verification, as well as a verification tool for Azure RBAC.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and our colleagues at Microsoft for their support.

REFERENCES

[1] 2022 (accessed Jan 21, 2022). *Amazon IAM Policy Grammar.* https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_grammar.html

[2] 2022 (accessed Jan 21, 2022). *CyberArk: Hunting Azure Blobs Exposes Millions of Sensitive Files.* https://www.cyberark.com/resources/threat-research-blog/hunting-azure-blobs-exposes-millions-of-sensitive-files

[3] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark A. Stalzer, Preethi Srinivasan, Pavle Subotic, Carsten Varming, and Blake Whaley. 2019. Reachability Analysis for AWS-Based Networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 231–241. https://doi.org/10.1007/978-3-030-25543-5_14

[4] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8602994

[5] Nikolaj S. Bjørner. 2018. Z3 and SMT in Industrial R&D. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10951)*, Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink (Eds.). Springer, 675–678. https://doi.org/10.1007/978-3-319-95582-7_44

[6] Samuel R. Buss and Jakob Nordström. 2021. Proof Complexity and SAT Solving. In *Handbook of Satisfiability*.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*.

[8] William Eiers, Ganesh Sankaran, Albert Li, Emily O'Mahony, Benjamin Prince, and Tevfik Bultan. 2022. Quantifying Permissiveness of Access Control Policies. https://dl.acm.org/doi/10.1145/3510003.3510233

[9] Anna Lisa Ferrara, P. Madhusudan, and Gennaro Parlato. 2012. Security Analysis of Role-Based Access Control through Program Verification. *2012 IEEE 25th Computer Security Foundations Symposium* (2012), 113–125.

[10] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. 2005. Verification and Change-Impact Analysis of Access-Control Policies. https://dl.acm.org/doi/10.1145/1062455.1062502

[11] Yoshihiko Futamura. 1982. Parital Computation of Programs. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings (Lecture Notes in Computer Science, Vol. 147)*, Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa (Eds.). Springer, 1–35. https://doi.org/10.1007/3-540-11980-9_13

[12] Vijay Ganesh and Moshe Y. Vardi. 2020. On the Unreasonable Effectiveness of SAT Solvers. In *Beyond the Worst-Case Analysis of Algorithms*.

[13] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. 1976. Protection in Operating Systems. *Commun. ACM* 19, 8 (aug 1976), 461–471. https://doi.org/10.1145/360303.360333

[14] Graham Hughes and Tevfik Bultan. 2008. Automated Verification of Access Control Policies Using a SAT Solver. *Int. J. Softw. Tools Technol. Transf.* 10, 6 (dec 2008), 503–520.

[15] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies*. Technical Report MSR-TR-2014-102. Microsoft. https://www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/

[16] Karthick Jayaraman, Mahesh V. Tripunitara, Vijay Ganesh, Martin C. Rinard, and Steve J. Chapin. 2013. Mohawk: Abstraction-Refinement and Bound-Estimation for Verifying Access Control Policies. *ACM Trans. Inf. Syst. Secur.* 15 (2013), 18:1–18:28.

[17] Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. 2016. Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica* 53, 4 (2016), 387–424. https://doi.org/10.1007/s00236-015-0236-z

[18] Silvio Ranise, Anh Tuan Truong, and Riccardo Traverso. 2016. Parameterized model checking for security policy analysis. *International Journal on Software Tools for Technology Transfer* 18 (2016), 559–573.

[19] Philipp Rümmer and Pavle Subotic. 2013. Exploring interpolants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 69–76. https://ieeexplore.ieee.org/document/6679393/

[20] Ehtesham Zahoor, Zubaria Asma, and Olivier Perrin. 2017. A Formal Approach for the Verification of AWS IAM Access Control Policies. In *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10465)*, Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen (Eds.). Springer, 59–74. https://doi.org/10.1007/978-3-319-67262-5_5

[21] Aolong Zha, Qiong Chang, and Itsuki Noda. 2023. An incremental SAT-based approach for solving the real-time taxi-sharing service problem. *Discrete Applied Mathematics* 335 (2023), 131–145. https://doi.org/10.1016/j.dam.2022.08.008 Emerging Applications, Models and Algorithms in Combinatorial Optimization.