Efficient Enumeration of Cloud Access Permissions

Lee A. Barnett, Loris D'Antoni, Amit Goel, Rami Gökhan Kıcı, Tim King, Kasper Luckow, Neha Rungta, Yasmine Sharoda
Amazon Web Services
{lbarnt, lorisd, amgoel, ramikici, timaking, luckow, rungta, sharoday}@amazon.com

***Abstract*—Cloud computing platforms like Amazon Web Services require scalable methods for analyzing access control policies and determining which principals can perform which actions on which resources. Prior work has developed SMT-based techniques for analyzing access control policies, but enumerating all allowed principal-resource combinations involves up to millions of checks, making naive enumeration intractable. We formalize the *access enumeration* problem: given sets of principals and resources with their access control policies, compute all pairs where access is possible. We present a sound and complete algorithm that exploits structural independence in access control formulas: principal policies and resource policies share only request variables, enabling separate analysis. Our first technique, *projection-based filtering*, computes the sets of resources each principal could potentially access and the sets of principals each resource could potentially allow, filtering candidate pairs before joint satisfiability checking. Our second technique applies *model generalization* to reuse satisfying assignments across similar domain elements, reducing SMT solver calls. Evaluation on synthetic benchmarks shows 2.2–5.1× speedup over the baseline, and production deployment powering a commercial cloud security feature demonstrates scalability to environments with millions of principal-resource pairs.**

I. Introduction

Cloud computing platforms now host critical applications in finance, healthcare, and other sectors handling sensitive data, motivating the use of formal methods to ensure correct access control [1–4]. However, modern cloud platforms may have thousands of principals and resources, each governed by access control policies that describe under what conditions principals can access certain resources. Answering basic security questions, such as which users can access which databases or what actions each service can perform, requires reasoning about millions of principal-resource combinations. Because cloud providers must offer this type of analysis to millions of customers, scalability in answering such questions is essential.

Prior work has developed SMT-based techniques for reasoning about access control policies. Specifically, Zelkova [1] established how to encode individual Amazon Web Services (AWS) access control policies as SMT formulas, enabling *universal* questions to be answered by checking satisfiability of a single formula representing all possible requests. For example, this approach can determine whether a policy could ever allow public access [5].

In this paper, we are interested in approaches to answer a different question that emerges when performing comprehensive security audits: *Who can access what?* This is not a single query but an enumeration problem: given thousands of principals and resources, identify all pairs where access is possible. A prior approach, called stratified abstraction [6] (which powers AWS IAM Access Analyzer [7]), addressed a related problem by summarizing permissions into human-readable findings, but this approach analyzes each access control policy independently, leading to computational cost that grows proportionally with the product of the number of principals and resources. For organizations with hundreds to thousands of principals and resources, such an approach requires thousands to millions of separate analyses, posing a significant scalability challenge.

The challenge is compounded by practical and operational requirements. Access enumeration is not an analysis that can be performed only once: cloud administrators can modify permissions by changing access control policies, adding or removing principals or resources, and cloud platforms continuously introduce new actions. To provide useful security insights, access enumeration must be repeated regularly, such as on a daily or even more frequent basis, to reflect the current state of each environment. With many customer environments to analyze, each containing thousands of principals and resources, the aggregate computational cost becomes prohibitive. While modern cloud platforms offer abundant compute capacity, cost scales with usage; simply parallelizing a quadratic algorithm across more machines does not yield a practical solution.

Our main contribution is an algorithm, called *projection-based filtering*, that exploits the structure of access control formulas to decompose them into smaller formulas that are easier to solve and avoid issuing satisfiability queries for all pairs of principals and resources. The key observation is that although there are $|P| \times |R|$ principal-resource combinations, the underlying policies decompose into $|P|$ principal-specific components and $|R|$ resource-specific components. This decomposition reflects how authorization logic in practice separates principal and resource concerns, and we deliberately preserve and exploit this structure. Similar to how database engines push "selects before joins" [8–10], by analyzing these components independently, we can often filter the vast majority of pairs before relatively more expensive joint analysis. To further speed up our approach, we develop a *model generalization* technique that reuses satisfying assignments across similar domain elements to avoid performing a large number of SMT queries. Because access control policies often use wildcard patterns, model generalization exploits the fact that often similar principals will be granted access to similar resources via similar requests.

Contributions. This paper makes the following contribu-

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_32

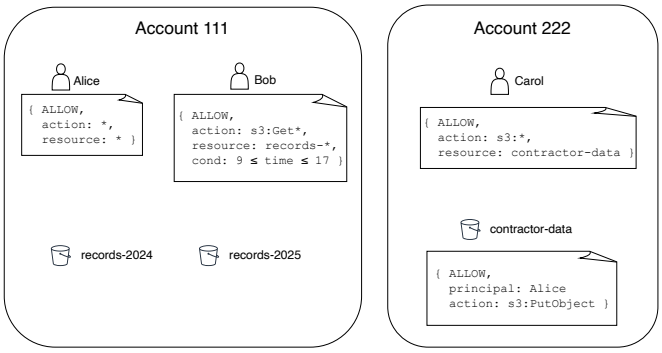

Fig. 1. Principals and resources in the accounts of an example AWS Organization, along with their access control policies. Note that there are no policies attached to either storage resource in account `111`.

tions:

- We formalize the *access enumeration problem* for describing who has access to what in cloud systems (Section II).
- We present *projection-based filtering* and *model generalization*, two techniques that exploit the structure of access control formulas to make enumeration practical at scale (Section III).
- We implement our techniques in a tool that supports the full complexity of real cloud policies and services appearing in AWS (Section IV).
- We evaluate our tool on synthetic benchmarks derived from real policy configurations (2.2–5.1× speedup over the baseline), and report on its production deployment powering a commercial cloud security feature launched in 2025 (Section V).

Section VI surveys related work, and Section VII concludes.

II. Background and Example

To motivate the access enumeration problem we consider an example financial company using AWS to manage their cloud organization. We focus on AWS as there exist established translations for its policies into SMT formulas [1, 11].

In AWS, an organization consists of principals (users, roles, and services that can perform actions) and resources (data storage, compute instances, and other cloud services that can be accessed). In our example, the organization contains two separate accounts: account `111` contains principals for company employees and storage resources for internal data, while account `222` contains principals and storage resources used by external contractors. In our example, as part of a compliance audit, the company must produce a comprehensive report detailing which principals can access which resources and what actions they can perform across their entire AWS Organization.

Figure 1 shows a diagram of the accounts, along with their principals, resources, and policies. Account `111` includes Alice, an administrator with broad access to company resources, and Bob, an analyst with read-only access to financial records during business hours. This account hosts two AWS S3 Buckets (cloud storage containers): `records-2024` and `records-2025`, containing financial records from each respective year. Account `222` contains Carol, an external contractor, and an S3 Bucket named `contractor-data` with files shared between the company and contractors.

A. Access Control in AWS

In AWS, access control operates through Identity and Access Management (IAM) policies: JSON documents that specify which principals can perform which actions on which resources under what conditions. A principal's *identity-based policy* constrains which actions it may perform on which resources, while a resource's *resource-based policy* constrains which principals may access it and what actions they may perform. Policies can also place additional conditions on other properties of authorization requests, such as the time the request was made or the originating IP address.

Whether authorization requests succeed depends on the combination of all relevant policies. Therefore, producing the access enumeration (i.e., the set of what principals have access to what resources) requires a sound and complete analysis of the access control policies attached to each principal and resource, and how they are combined and evaluated during authorization.[1]

For this example, the key distinction in how access is granted is between intra-account and cross-account access:

- **Intra-account access:** If the principal and resource are in the same account, an authorization request is permitted if *either* the principal's policy or the resource's policy allows the request, and no policy denies it.
- **Cross-account access:** If the principal and resource are in different accounts, an authorization request is permitted only if *both* policies allow it, and no policy denies it.

In other words, intra-account authorization combines policies with disjunction, while cross-account authorization combines them with conjunction. This asymmetry complicates reasoning about who has access to what [11] because the same policies can produce different results depending on account boundaries.

Table I lists which specific actions are allowed for each principal-resource pair, drawn from the full list of actions available in AWS S3, which can be found in the S3 documentation [13]. We describe three of the cases in detail.

(Bob, records-2024). Bob and records-2024 are both in account `111`, so either policy (identity or resource) allowing the requested action suffices to grant the action. Bob's policy allows actions starting with "`s3:Get`" on resources whose names start with "records-", subject to a time condition. Since records-2024 starts with "records-" and has no resource policy, access depends entirely on Bob's policy. Bob can perform any

[1] AWS authorization involves several steps and includes other factors not shown in this example, such as organization-level policies, permission boundaries, and session policies. In addition, some accesses such as intra-account IAM role assumption have different behavior than what we explained above, which we leave out for brevity. The full authorization logic is documented in AWS's IAM documentation [12] and described in prior work [11].

TABLE I
A COMPLETE LISTING OF ALL POSSIBLE ACCESSES GIVEN THE PRINCIPALS, RESOURCES, AND POLICIES SHOWN IN FIGURE 1.

Principal	Resource	Allowed Actions
Alice	records-2024	`s3:AbortMultipartUpload`, ..., `s3:UpdateStorageLensGroup`
Alice	records-2025	`s3:AbortMultipartUpload`, ..., `s3:UpdateStorageLensGroup`
Alice	contractor-data	`s3:PutObject`
Bob	records-2024	`s3:GetAccelerateConfiguration`, ..., `s3:GetStorageLensGroup`
Bob	records-2025	`s3:GetAccelerateConfiguration`, ..., `s3:GetStorageLensGroup`
Bob	contractor-data	None
Carol	records-2024	None
Carol	records-2025	None
Carol	contractor-data	`s3:AbortMultipartUpload`, ..., `s3:UpdateStorageLensGroup`

action whose name starts with "`s3:Get`" on the S3 Bucket records-2024 during business hours.

(Alice, contractor-data). Alice is in account `111` and contractor-data is in account `222`, so *both* policies must allow the requested action. Alice's policy allows any action ($\star$) on any resource ($\star$), while the resource policy on contractor-data allows Alice to perform only `s3:PutObject`. Since Alice and contractor-data belong to different AWS accounts, access depends on both policies. Alice can perform only `s3:PutObject` on contractor-data.

(Bob, contractor-data). Bob is in account `111` and contractor-data is in account `222`, so both policies (identity and resource) must allow the requested action to grant the action. Bob's policy restricts resources to those whose names start with "records-", but contractor-data does not. Additionally, the resource policy on contractor-data does not mention Bob. Either policy alone is sufficient to deny access: Bob's policy doesn't permit contractor-data, and the resource policy doesn't mention Bob.

B. Access Enumeration

The goal of access enumeration is to compute the set of principal-resource pairs for which there exists some action the principal can perform on the resource. Table I shows the access enumeration for our example, produced by analyzing all nine principal-resource pairs. For each pair, we want to determine whether there exists an authorization request (including a specific action, time of the request, source IP, and other parameters) that would be permitted by the given policies. For pairs where access is possible, Table I further enumerates which actions are permitted for each pair.

Determining whether access is possible is a satisfiability problem: policies impose constraints on request parameters (e.g., Bob can only access records during business hours), resource and action names must match wildcard patterns, and multiple policies interact to produce authorization decisions. Prior work has developed SMT-based techniques for reasoning about such policies. ZELKOVA established how to encode individual policy documents as SMT formulas [1], enabling checks such as whether a policy could allow unrestricted public access [5]. IAM-MULTIPOLICYANALYZER extended this foundation to model the complete AWS authorization engine, capturing how multiple policy types (identity-based policies, resource-based policies, permission boundaries, and organization-level policies) combine to produce authorization decisions [11].

While the IAM-MULTIPOLICYANALYZER encoding handles the intricate interaction of all policy types, we can express the resulting formula in a way that separates principal and resource concerns. For principal p and resource r, we construct a formula $\varphi_{(p,r)}(V)$ that is satisfiable if and only if p can access r under some assignment to the request variables V. Here V is a fixed set of variables, one for each property of an authorization request, including the action requested, the time of the request, and the source IP address. An assignment to V maps each to a concrete value. This formula has the form:

$$\varphi_{(p,r)}(V) = f_p(r, V) \otimes f_r(p, V) \tag{1}$$

where $f_p(r, V)$ encodes the collection of policies governing what principal p can access, $f_r(p, V)$ encodes policies governing who can access resource r, and $\otimes$ is disjunction ($\vee$) for intra-account pairs or conjunction ($\wedge$) for cross-account pairs.

Importantly, the formula $\varphi_{(p,r)}(V)$ depends on the specific policies attached to p and r. Different principals have different identity-based policies; different resources have different resource-based policies. In our illustrative example, we get formulas like:

$$\begin{aligned}\varphi_{(\text{Alice,contractor-data})}(V) &= f_{\text{Alice}}(\text{contractor-data}, V) \\ &\quad \wedge f_{\text{contractor-data}}(\text{Alice}, V) \\ \varphi_{(\text{Carol,contractor-data})}(V) &= f_{\text{Carol}}(\text{contractor-data}, V) \\ &\quad \vee f_{\text{contractor-data}}(\text{Carol}, V)\end{aligned}$$

The straightforward approach to access enumeration is to generate $\varphi_{(p,r)}(V)$ for every principal-resource pair, and check the satisfiability of each independently. This approach is sound and complete by construction, as it relies on IAM-MULTIPOLICYANALYZER's established formalization of AWS authorization [11]. However, with $|P|$ principals and $|R|$ resources, this requires $|P| \times |R|$ satisfiability checks. For production systems with hundreds to thousands of principals and resources, this yields many thousands to millions of SMT solver queries. While each individual query may be tractable, the sheer number of queries makes comprehen-

sive access enumeration computationally infeasible at scale.[2] Moreover, this approach fails to take advantage of shared structure across the set of formulas. For example, both formulas $\varphi_{(\text{Alice,contractor-data})}(V)$ and $\varphi_{(\text{Carol,contractor-data})}(V)$ share the resource component $f_{\text{contractor-data}}$.

III. Efficient Access Enumeration

The key observation is that although there are $|P| \times |R|$ combined formulas $\varphi_{(p,r)}$, there are only $|P|$ distinct principal formulas f_p and $|R|$ distinct resource formulas f_r. Our approach exploits this structure in two ways: first, by analyzing principal and resource components independently to filter infeasible pairs before joint satisfiability checking; and second, by reusing satisfying assignments across similar domain elements to reduce SMT solver calls during enumeration.

This section formalizes the access enumeration problem (Section III-A), introduces projection-based decomposition to filter candidate pairs (Section III-B), develops model generalization (Section III-C), and combines these into a sound and complete algorithm (Section III-D). Finally, we extend our approach to enumerate the specific actions each principal can perform on each resource (Section III-E).

A. Problem Definition

Let P be a finite set of principals and R a finite set of resources. Recall from Section II-B that for each principal-resource pair (p, r), we can generate a formula $\varphi_{(p,r)}(V)$ (Equation 1) that encodes whether p could be authorized to access r under some assignment to the request variables V.[3] Let V denote the request variables, which in AWS IAM include, for example, the action to be performed, the time of the request, the source IP address, and other contextual properties.

Definition 1 (Access Enumeration)**.** *Given sets P of principals and R of resources, the* access enumeration *problem is to compute the set $\mathcal{A}$ of all pairs $(p, r) \in P \times R$ such that principal p can perform some action on resource r under some valid authorization request. Formally, this is the set:*

$$\mathcal{A} = \{(p, r) \mid \varphi_{(p,r)}(V) \text{ is satisfiable}\}$$

In our running example, we have $P = \{\text{Alice}, \text{Bob}, \text{Carol}\}$ and $R = \{\text{records-2024}, \text{records-2025}, \text{contractor-data}\}$. The set of access-admitting pairs $\mathcal{A}$, and the solution to the access enumeration problem for the principals and resources in Figure 1, is given in Table I. This formulation captures direct access from a principal to a resource. Indirect access, such as a principal assuming a role that in turn accesses a resource, is not represented as a single pair, although each step in such a chain is itself an access-admitting pair. Enumerating indirect access paths is beyond the scope of this paper.

[2]For pairs where access is possible, determining which actions are permitted requires further analysis. This extension is discussed in Section III-E.

[3]To be precise, the formula depends on the set of access control policies. Throughout this section, we assume a fixed collection of policies is given, allowing us to focus on the computational aspects without explicitly parameterizing formulas by policy sets. This assumption is without loss of generality.

B. Decomposing Formulas

Following prior work [11], the formulas representing access control for a given principal p and resource r have specific structures.

Definition 2 (Structure of Access Control Formulas)**.** *Each formula $\varphi_{(p,r)}(V)$ has the structure:*

$$\varphi_{(p,r)}(V) = f_p(r, V) \otimes f_r(p, V)$$

where $f_p(r, V)$ encodes the policies governing what principal p can access, $f_r(p, V)$ encodes the policies governing who can access resource r, and $\otimes \in \{\wedge, \vee\}$ is determined by the relationship between p and r. In AWS IAM, $\otimes = \wedge$ for cross-account pairs and $\otimes = \vee$ for intra-account pairs.

Although we have $|P| \times |R|$ formulas of the form $\varphi_{(p,r)}(V)$, the structural decomposition reveals that we only have $|P|$ distinct principal formulas $f_p(r, V)$ (one per principal) and $|R|$ distinct resource formulas $f_r(p, V)$ (one per resource). The first key insight behind our approach is to analyze these component formulas in isolation: by precomputing which resources each principal's policies could potentially permit access to, and which principals each resource's policies could potentially allow, we can dramatically reduce the number of joint satisfiability checks required.

Definition 3 (Principal and Resource Projections)**.** *For each principal $p \in P$, define the* resource projection*:*

$$\pi_R(p) = \{r \in R \mid \exists V.\ f_p(r, V)\}$$

Symmetrically, for each resource $r \in R$, define the principal projection*:*

$$\pi_P(r) = \{p \in P \mid \exists V.\ f_r(p, V)\}$$

Note that $\pi_R(p) \subseteq R$ for each p, and similarly $\pi_P(r) \subseteq P$ for each r.

In our running example, Bob's policy restricts access to resources starting with "records-," so $\pi_R(\text{Bob}) = \{\text{records-2024}, \text{records-2025}\}$. Regardless of whether the principal and resource are in the same account, we know that Bob does not have access to contractor-data. The resource policy on contractor-data allows only Alice, yielding $\pi_P(\text{contractor-data}) = \{\text{Alice}\}$. Since neither records-2024 nor records-2025 has a policy, $\pi_P(\text{records-2024}) = \pi_P(\text{records-2025}) = \emptyset$.

As illustrated by the above examples, projections can be used to filter out principal-resource pairs.

Theorem 1 (Projection-Based Filtering)**.** *For any principal $p \in P$, resource $r \in R$, and corresponding formula $\varphi_{(p,r)}(V) = f_p(r, V) \otimes f_r(p, V)$*

1) *If $r \notin \pi_R(p)$ and $p \notin \pi_P(r)$, then $\varphi_{(p,r)}(V)$ is unsatisfiable.*
2) *If $\otimes = \wedge$ and either $r \notin \pi_R(p)$ or $p \notin \pi_P(r)$, then $\varphi_{(p,r)}(V)$ is unsatisfiable.*

Proof. For case (1): If $r \notin \pi_R(p)$, then $\neg\exists V.\ f_p(r, V)$, so $f_p(r, V)$ is unsatisfiable. Similarly, if $p \notin \pi_P(r)$, then $f_r(p, V)$

is unsatisfiable. When both conditions hold, neither component can be satisfied, making $f_p(r, V) \otimes f_r(p, V)$ unsatisfiable regardless of $\otimes$.

For case (2): When $\otimes = \wedge$, both $f_p(r, V)$ and $f_r(p, V)$ must be satisfiable for the conjunction to hold. If either projection fails, the corresponding component is unsatisfiable, making the entire conjunction unsatisfiable. □

Theorem 1 allows us to avoid checking satisfiability for many principal-resource pairs by proving them unsatisfiable through projection analysis alone.

We write *candidate pairs* $\mathcal{P} \subseteq P \times R$ to denote all principal-resource pairs that are not proven unsatisfiable by Theorem 1.

C. Model Generalization

Computing each projection is an instance of the finite-domain model enumeration problem.

Definition 4 (Finite-Domain Model Enumeration)**.** *Given $\psi(x, V)$ where x ranges over a finite domain D, the* finite-domain model enumeration problem *is to compute the set*

$$\{d \in D \mid \exists V.\ \psi(d, V)\}$$

For example, computing $\pi_R(p)$ requires finding all the resources $r \in R$ such that $f_p(r, V)$ is satisfiable. The naive approach to finite-domain model enumeration issues one SMT solver call per domain element, which in the case of $f_p(r, V)$ would require $|R|$ satisfiability queries.

Our next key insight is to avoid so many satisfiability queries by exploiting the following simple observation. Suppose the solver has found a resource r_{solved} together with a concrete, complete assignment c_{solved} to the request variables that satisfies the formula $f_p(r_{solved}, c_{solved})$. Because of how policies are usually written, there often exist many other resources for which the same context would make the formula satisfied. Therefore, we can iterate over every $r_{check} \in R$ and simply check if the formula $f_p(r_{check}, c_{solved})$ is valid. Such a validity check is much cheaper than a satisfiability check (it just requires evaluating a formula); when a formula is valid for a value $r_{check} \in R$, we can add r_{check} to the solution set $\pi_R(p)$ and avoid a satisfiability query.

Algorithm 1 presents finite-domain model enumeration with model generalization. For a given formula $\psi(x, V)$, the algorithm maintains a set $\mathcal{C}$ of previously discovered satisfying assignments to the variables V. For each domain element d in the domain D, the algorithm first attempts to reuse each context in $\mathcal{C}$ by checking if the formula remains satisfied when d is substituted for the original domain value. If any context reuse succeeds, d is added to the result via simple evaluation rather than an SMT solver call. Otherwise, the solver is invoked to find a new satisfying context; if satisfiable, this new context is added to $\mathcal{C}$ for future reuse.

Theorem 2 (Correctness of Model Generalization)**.** *Algorithm 1 correctly computes the set $\{d \in D \mid \exists V.\ \psi(d, V)\}$.*

Proof. **Soundness:** Every element d added to S satisfies the condition $\exists V.\ \psi(d, V)$. This holds because d is added only if either (1) a reused context c satisfies $\psi(d, c)$, or (2) the SMT solver finds $\psi(d, V)$ satisfiable with a new context.

Completeness: Every element $d \in D$ such that $\exists V.\ \psi(d, V)$ is included in S. If no context reuse succeeds for d, the algorithm explicitly checks satisfiability of $\psi(d, V)$, ensuring no satisfiable elements are missed. □

Algorithm 1 Finite-Domain Model Enumeration with Generalization

Require: Formula $\psi(x, V)$, finite domain D
Ensure: Set $S = \{d \in D \mid \exists V.\ \psi(d, V)\}$

```
S ← ∅, C ← ∅                          ▷ Store results and discovered contexts
for d ∈ D do
    found ← false
    for c ∈ C do                       ▷ Try to reuse existing contexts
        if ψ(d, c) is true then
            S ← S ∪ {d}, found ← true
            break
    if ¬found then                     ▷ No existing context, call SMT
        if ψ(d, V) is satisfiable with context c_new then
            S ← S ∪ {d}, C ← C ∪ {c_new}
return S
```

Model generalization is effective when many domain elements are satisfied under similar assignments to the variables V. This is common in access control policies, where wildcards like `records-*` (the predicate accepting strings that start with "records-") match multiple resources under identical request contexts. Notably, the two techniques are complementary: filtering is most effective in environments with restrictive policies, where many principal-resource pairs can be ruled out; model generalization is most beneficial when policies grant broad permissions, since one satisfying context (the model) is more likely to transfer across similar queries.

Returning to the running example, consider computing $\pi_R(\text{Bob})$, the set of resources to which Bob's identity policy grants access. The SMT solver determines that the formula $f_{\text{Bob}}(\text{records-2024}, V)$ is satisfiable by finding the context

$$c = \{\text{action} \mapsto \texttt{s3:GetObject}, \text{time} \mapsto 12\}.$$

To check records-2025, we simply evaluate the formula $f_{\text{Bob}}(\text{records-2025}, c)$ under this existing context. Since both resources match the `records-*` pattern and the time condition remains valid, the formula evaluates to true; therefore records-2025 $\in \pi_R(\text{Bob})$ without requiring an additional SMT solver call. This single context discovery enables us to determine Bob's access to both records buckets with only one satisfiability query instead of two.

D. Access Enumeration Algorithm

Algorithm 2 combines decomposition and model generalization to solve access enumeration efficiently. The algorithm first uses model generalization to compute the projections $\pi_R(p)$ (for each principal p) and $\pi_P(r) \subseteq P$ (for each resource r) and only checks satisfiability of the pairs that are not filtered by Theorem 1.

Algorithm 2 Access-Admitting Pair Enumeration

Require: Principal set P, resource set R
Ensure: Access-admitting pairs $\mathcal{A} = \{(p, r) \mid \varphi_{(p,r)}(V)$ is satisfiable$\}$

1: **for** $p \in P$ **do** ▷ Compute resource projections
2: $\quad \pi_R(p) \leftarrow$ MODELGENERALIZATION$(f_p(r, V),\ R)$
3: **for** $r \in R$ **do** ▷ Compute principal projections
4: $\quad \pi_P(r) \leftarrow$ MODELGENERALIZATION$(f_r(p, V),\ P)$
5: $\mathcal{C} \leftarrow \{(p, r) \in P \times R \mid (r \in \pi_R(p) \vee p \in \pi_P(r)) \wedge ([\otimes = \vee] \vee (r \in \pi_R(p) \wedge p \in \pi_P(r)))\}$
6: $\mathcal{A} \leftarrow \emptyset$
7: **for** $(p, r) \in \mathcal{C}$ **do** ▷ Verify remaining candidates
8: $\quad$ **if** $\varphi_{(p,r)}(V)$ is satisfiable **then**
9: $\quad\quad \mathcal{A} \leftarrow \mathcal{A} \cup \{(p, r)\}$
10: **return** $\mathcal{A}$

Example 1 (Running Example Trace)**.** *Per our example, let* $P = \{\textit{Alice}, \textit{Bob}, \textit{Carol}\}$ *and* $R = \{\textit{records-2024}, \textit{records-2025}, \textit{contractor-data}\}$*. First, the projection of Bob is* $\pi_R(\textit{Bob}) = \{\textit{records-2024}, \textit{records-2025}\}$ *using one SMT call plus context reuse, while* $\pi_R(\textit{Alice}) = R$ *due to her permissive administrator-access policy. Second,* $\pi_P(\textit{contractor-data}) = \{\textit{Alice}\}$ *and as previously explained,* $\pi_P(\textit{records-2024}) = \pi_P(\textit{records-2025}) = \emptyset$ *as these resources have no policies which could grant access. Line 5 filters out* $(\textit{Bob}, \textit{contractor-data})$ *since* $\textit{contractor-data} \notin \pi_R(\textit{Bob})$ *and* $\textit{Bob} \notin \pi_P(\textit{contractor-data})$*. Lines 7–9 verify the remaining candidates, confirming that principal-resource pairs such as* $(\textit{Alice}, \textit{contractor-data})$ *and* $(\textit{Bob}, \textit{records-2024})$ *are satisfiable.*

The following corollary establishes soundness and completeness of Algorithm 2 and follows from Theorem 1 and Theorem 2.

Corollary 1 (Correctness of Access-Admitting Pair Enumeration)**.** *Algorithm 2 computes exactly the set of access-admitting pairs:*

$$\mathcal{A} = \{(p, r) \in P \times R \mid \varphi_{(p,r)}(V) \textit{ is satisfiable}\}$$

The computational complexity of our algorithm can be analyzed in terms of SMT solver calls. Computing projections naively would require $|P| \times |R|$ calls for resource projections and $|R| \times |P|$ calls for principal projections, totaling $2|P| \times |R|$ SMT queries (though on the smaller formulas $f_p(r, V)$ and $f_r(p, V)$ rather than $\varphi_{(p,r)}$). However, model generalization reduces this cost: in the best case where contexts generalize perfectly across domain elements, we require only $|P| + |R|$ SMT calls (one per principal and one per resource). In practice, access control policies with wildcards like `records-*` and `s3:Get*` enable significant context reuse. Combined with aggressive filtering that eliminates most candidate pairs, our approach reduces total computational cost compared to the naive $|P| \times |R|$ joint satisfiability checks, as demonstrated in Section V. The number of solver calls in the verification loop may be further reduced by reusing satisfying contexts discovered during the projection loops, for example by grouping candidate pairs that share a principal or a resource. However, during verification both the principal and resource components of $\phi_{p,r}(V)$ vary across iterations, limiting the potential of generalization compared to the projection phase. We leave exploration of this optimization to future work.

E. Action Enumeration

Algorithm 2 computes which principal-resource pairs admit access. For compliance and auditing purposes, it is often useful to also determine the specific actions each principal can perform on each resource, as shown in Table I.

Let Act denote the finite set of actions relevant to a given resource type. For example, S3 buckets support more than a hundred individual actions, such as `s3:GetObject` and `s3:PutObject`.

Definition 5 (Action Enumeration Problem)**.** *The* action enumeration problem *is to compute all triples* (p, r, a) *such that principal* p *can perform action* a *on resource* r*:*

$$\{(p, r, a) \mid a \in \mathsf{Act}_r \wedge \exists V'.\ \varphi_{(p,r)}(a, V') \textit{ is satisfiable}\}$$

For each access-admitting pair $(p, r) \in \mathcal{A}$, determining the permitted actions is precisely an instance of the finite-domain model enumeration problem (Definition 4) with domain $D = \mathsf{Act}_r$ and formula $\psi(a, V') = \varphi_{(p,r)}(a, V')$.

The key insight is that our model generalization technique (Algorithm 1) applies directly to this problem. Access control policies commonly use action wildcards like `s3:Get*` or `s3:*`, meaning that a single satisfying context often generalizes across multiple actions. For example, if a specific authorization request context $c = \{\text{time} \mapsto 12, \text{IP} \mapsto \text{internal}\}$ satisfies the formula for action `s3:GetObject`, it likely also satisfies the formula for `s3:GetObjectVersion`, `s3:GetObjectAcl`, and other actions beginning with "`s3:Get`."

Therefore, action enumeration for each pair $(p, r) \in \mathcal{A}$ can be computed efficiently using the same model generalization approach, avoiding the naive approach of issuing $|\mathsf{Act}_r|$ independent SMT queries. The extent of this benefit depends on the contents of the policies involved. Policies using broad action wildcards such as `s3:*` may let a single context cover many actions, whereas policies that enumerate individual actions offer fewer opportunities for reuse.

IV. IMPLEMENTATION

We implemented our approach in a tool for performing access enumeration for AWS environments. The input to our tool is a description of a target AWS environment to be analyzed, such as an account or an organization, including principals P, resources R, and all relevant access control policies. The output is the set of access-admitting pairs $\mathcal{A} \subseteq P \times R$. Each $(p, r) \in \mathcal{A}$ can then be analyzed further to provide the administrators of the analyzed environment more thorough information about the access control posture

of their AWS estate, such as enumerating the actions that p could be authorized to perform on r (as in Section III-E). One application built using our tool applies the *stratified abstraction* algorithm to each $(p, r) \in \mathcal{A}$ to produce a full listing of possible *access paths* from p to r, describing all combinations of actions and condition values that could be allowed [6]. We provide statistics about this application in Section V-B.

Our tool is itself implemented as a cloud application on AWS. The workflow of Algorithm 2 is orchestrated using AWS Step Functions [14]. Projection computation for each principal and resource (lines 1–4) is performed using AWS Lambda Functions configured with approximately 5 GB of memory [15], as are the computation of candidate pairs (line 5) and the final verification step for each pair (lines 7–9). The encoding produces constraints over strings together with other theories such as bitvectors and integers. For computing satisfiability results, each Lambda Function therefore uses a portfolio of SMT solvers, including cvc5 [16] and z3 [17], taking the first result returned. Running multiple solvers improves robustness, as their relative performance varies across the range of queries our encoding generates.

Our implementation is also designed with the practical considerations of running production workloads in mind. For example, we use parallelism to process the **for** loops in Algorithm 2, taking advantage of the *Map State* feature of AWS Step Functions. While in theory it may be possible to simply run all iterations of each loop at the same time, such an approach would lead to separate workloads contending for the same compute resources and could significantly lower the overall throughput of our application. Passing intermediate results between Lambda Functions as part of the workflow also incurs I/O overhead, and it is important to avoid and manage contention here as well. These constraints highlight another advantage of our approach: reducing total work, rather than simply exploiting parallelism, is essential for scalable, cost-effective access enumeration.

V. Evaluation

We evaluated our approach using the following research questions.

RQ1: Algorithmic Component Analysis. How do decomposition and model generalization individually contribute to performance compared to the naive baseline?

RQ2: Production Deployment Effectiveness. How does our tool perform when deployed in real-world applications with practical policy configurations?

Experiments designed to answer these questions are described below. Please note that performing these experiments requires access to the proprietary IAM-to-SMT encoding, which we are unable to release for independent reproduction.

A. RQ1: Algorithmic Component Analysis

To understand the contribution of each algorithmic component in our approach, we ran an ablation experiment to compare the following three configurations.

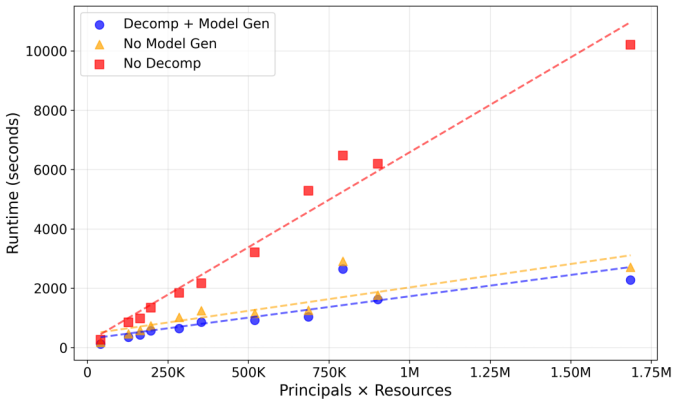

Fig. 2. Runtime scaling across configurations. **Decomp + Model Gen** achieves 2.2–5.1× speedup over **No Decomp**, with the gap generally widening at larger scales.

- **Decomp + Model Gen**: Decomposition via projection-based filtering, applying model generalization when computing the projections (i.e., the complete approach described in Section III).
- **No Model Gen**: Decomposition via projection-based filtering, but naive enumeration for projections (i.e., the technique in Section III-B is enabled but the one in Section III-C is disabled).
- **No Decomp**: No decomposition, leading to an independent satisfiability check for each $(p, r) \in P \times R$ (i.e., without any of the techniques presented in this paper).

To perform this experiment and determine the contributions of each method, we deployed the implementation described in Section IV to a *non-production environment* with additional input options for switching between the different configurations (the production version only runs the full algorithm). **Decomp + Model Gen** is our approach as given in Algorithm 2, and the environment matches the production deployment of our tool. For **No Model Gen**, the AWS Lambda Functions computing the per-principal and per-resource projections were altered to disable the model generalization optimization (Section III-C). For **No Decomp**, the AWS Step Functions workflow was configured to skip projection computation and candidate filtering, and proceed with joint access analysis for each $(p, r) \in P \times R$. This last configuration represents the baseline: directly applying the approach from prior work [11] to each principal-resource pair, without decomposition or model generalization. Note that model generalization as an optimization is enabled by decomposition: different principal-resource pairs involve different principal policies, different resource policies, and potentially different operators ($\wedge$ vs $\vee$), obscuring the shared structure for contexts to generalize across. For this reason there is no experiment configuration with model generalization but without decomposition.

a) Benchmarks: We generated synthetic benchmarks by scaling real AWS account configurations owned by our organization. We use synthetic benchmarks here rather than real

production inputs because the latter include content (access control policies) authored by AWS customers. These cannot be analyzed or accessed outside the production setting, where only the full algorithm runs and the **No Decomp** baseline would in any case be prohibitively expensive at scale. Our goal was to create a collection of inputs scaling to larger sizes while preserving the structures of realistic policy documents, which are difficult to synthesize from scratch. We extracted IAM principals (roles and users), Amazon S3 Buckets [13], and DynamoDB Tables [18], along with their associated identity and resource policies, from a few separate AWS accounts. To produce each benchmark, we expanded a source account by randomly selecting 10 principals and 10 resources and duplicating them along with their policy documents. Identifiers for duplicated entities are added to other policies referencing their original principal or resource with low probability (0.25), producing sparse access patterns consistent with least-privilege best practices. We generated 10 benchmark inputs with $|P| \times |R|$ ranging from 41,173 to 901,582; to validate scaling behavior, we also evaluated on a larger input with $|P| \times |R| = 1,685,583$.

For each benchmark, we ran each configuration three times and report median runtimes. Because our tool runs on AWS cloud infrastructure, duration can vary across identical runs due to factors such as AWS Lambda cold starts and scaling delays, or variable polling latencies when transitioning between states in AWS Step Functions. The resulting median variance observed across identical runs was 3.4%.

b) Results: Figure 2 shows runtime scaling across configurations for our experiment. The **Decomp + Model Gen** configuration achieves 2.2–5.1× speedup over **No Decomp**, with the improvement generally increasing at larger scales: from 2.2× at 41K pairs to 4.5× at 1.69M pairs (with the largest observed speedup of 5.1× at 686K pairs). Model generalization contributes 1.1–1.6× additional speedup over **No Model Gen**. At 901K pairs, **Decomp + Model Gen** completes in 27.0 minutes compared to 103.2 minutes for **No Decomp** (3.8× speedup). For the largest input (1.69M pairs), **Decomp + Model Gen** completed in 38.0 minutes versus 170.1 minutes for **No Decomp** (4.5× speedup). The speedup is driven primarily by filtering effectiveness: projection-based filtering (applied in both the **Decomp + Model Gen** and **No Model Gen** configurations) eliminates 49.6–79.1% of candidate pairs before joint analysis, with effectiveness increasing at larger scales.

c) Findings: To answer **RQ1**: decomposition provides the primary performance benefit (2.2–5.1× speedup), while model generalization contributes a more modest additional improvement (1.1–1.6×). Filtering effectiveness increases at larger scales, eliminating up to 79% of candidate pairs before joint analysis.

B. RQ2: Production Deployment Effectiveness

The synthetic benchmarks in RQ1 were designed to reflect least-privilege practices, producing sparse access patterns.

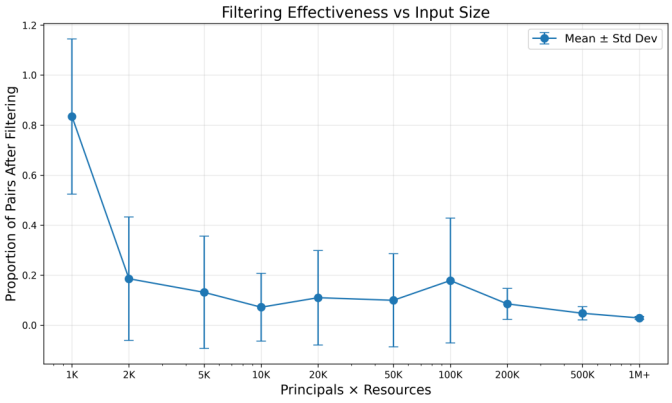

Fig. 3. Filtering effectiveness vs. input size over six months of production inputs in a large region (log scale). Each point shows the mean proportion of pairs requiring joint analysis after filtering, aggregated into buckets (e.g., the point at 2K includes production requests where $1,000 < |P \times R| \leq 2,000$). Error bars show standard deviation.

While least-privilege is a security best practice, real-world environments may vary in how strictly they follow it. To validate our approach under realistic conditions, we report aggregate metrics from production deployment over a six-month period in a major region. Our tool has been deployed since 2025 as part of a commercial cloud security feature that analyzes internal access to critical resources. The feature allows security teams to specify critical resources (such as S3 buckets or DynamoDB tables containing sensitive data) and identifies which IAM roles and users within their organization have access to those resources. Due to confidentiality constraints, we report averages and ranges rather than exact values or counts.

a) Input Size: Target AWS environments to be analyzed varied widely, ranging from hundreds to tens of thousands of principals. The number of resources was typically smaller, reflecting the design of the security feature built using our tool, which requires customers to specify particular "critical" resources for analysis. The largest analyzed environments involved several million principal-resource pairs.

b) Runtime and Throughput: The daily p99 (99th percentile) end-to-end latency of our tool, per target environment analysis, ranged from 3.5 minutes to over 50 minutes. The daily count of successful analyses ranged from a few thousand up to tens of thousands. The total number of analyses per day that exceeded the 4 hour analysis time limit ranged from 0 to 5.

c) Filtering Effectiveness: When analyzing real-world AWS environments, decomposition and filtering is generally more effective at larger scales. Figure 3 shows the average proportion of (p, r) pairs remaining after filtering that required joint analysis (i.e., $|\mathcal{C}|/|P \times R|$) across different sizes of real-world inputs analyzed in production. The data shows that filtering effectiveness tends to increase with input size. For small inputs with $|P \times R| < 1000$, the average proportion of remaining pairs was 0.84, meaning filtering removed only

around 16%, while for inputs with $|P \times R|$ greater than 500K the average proportion of remaining pairs was 0.03; that is, filtering removes 97% of all pairs. This pattern suggests that larger AWS environments tend toward sparser access patterns, consistent with least-privilege practices, making our approach most effective precisely where naive enumeration would be most costly. More generally, the gains from our approach depend on the density of allowed accesses. Filtering only removes pairs for which no access is possible, so environments where a large fraction of pairs admit access offer less to filter. Model generalization behaves in the opposite way, becoming more effective as policies grant broader permissions and a single satisfying context transfers across more pairs. The two techniques are therefore complementary across the range of access densities.

d) Findings: To answer **RQ2**, our tool performs well in production, with effectiveness of our approach increasing at scale. When analyzing large environments ($|P \times R| > 500$K), filtering eliminates 97% of pairs before joint analysis, while smaller environments see more modest benefits (16% elimination for $|P \times R| < 1000$). This confirms that our approach provides the greatest benefit for the large-scale analyses where naive enumeration would be prohibitively expensive. The system handles production workloads with p99 latency under 50 minutes with rare timeouts, and supports thousands of analyses per day.

VI. Related Work

Access control policies can be formalized using various approaches, each with distinct trade-offs in expressiveness and complexity [2, 3, 19, 20]. In terms of industrial impact, the most successful technique was established by Zelkova [1], which provided a foundational approach for encoding AWS access control policy semantics into SMT formulas, enabling automated reasoning. This work has spawned several extensions that build on its SMT-based foundations to address different aspects of policy analysis [5, 6, 11].

Backes et al. [6] developed stratified abstraction on top of Zelkova to produce human-readable summaries of the permissions granted by IAM policies. This approach powers AWS's IAM Access Analyzer service and handles policy explanation when complete policy information is unavailable (e.g., analyzing cross-account access where external account policies are unknown). In such scenarios, the abstraction technique assumes conservative bounds on external permissions. However, when all relevant policies are available for analysis (as is common in comprehensive organizational audits), this approach faces a quadratic explosion in complexity. Each principal-resource pair requires independent analysis across all applicable policies, leading to computational costs that grow quadratically with the number of entities. Our decomposition approach specifically addresses this scalability challenge by exploiting the structure of complete policy sets to avoid redundant computation, making comprehensive enumeration manageable even when precise analysis of all policies is required.

Barnett et al. [11] extend the Zelkova foundation through IAM-MultiPolicyAnalyzer, which models the complete AWS authorization engine by handling multiple interacting policy types. Their modular formalization addresses correctness verification across different policy types using a domain-specific language for composing individual policy analyses. Our work builds on top of IAM-MultiPolicyAnalyzer's comprehensive policy modeling capabilities, leveraging its sound multi-policy formalization as the foundation for our enumeration approach. While IAM-MultiPolicyAnalyzer focuses on correctness verification of individual queries, we extend this foundation to address the enumeration scalability problem across large sets of principals and resources.

Access enumeration, and the finite-domain model enumeration problem, share similarities to solution enumeration problems as studied in propositional logic such as All-SAT [21, 22] and All-SMT [23, 24]. However, there are key differences that make techniques for those problems difficult to apply. For the access enumeration problem (Definition 1), there is not one singular SMT formula whose satisfying assignments correspond to the solution; instead, each access control formula $\varphi_{(p,r)}(V)$ is constructed from separate principal and resource policies. While in principle one could construct a singular formula, it would require encoding individual constraints inside the same formula for each principal, resource, and policy in the AWS environment to be analyzed, resulting in formulas whose size grows with $P \times R$ and whose structure obscures the relationships and domain knowledge about access control that our decomposition exploits.

The finite-domain model enumeration problem (Definition 4) is similar to the problem of projected model enumeration [25, 26]. However, in our setting the variables do not range over domains that can be implicitly defined such as integers or bitvectors, but rather explicit sets of concrete values that lack succinct logical characterization (for example, the precise identifiers of all principals or resources in an AWS environment). Adapting existing techniques to solve the finite-domain enumeration problem would require encoding in the formula each element of the finite domain, again resulting in a large formula whose size grows with $P \times R$ and makes it difficult to take advantage of the similarities between how different domain elements are allowed access. Model sampling techniques in SAT and SMT (for example, [27–29]) offer efficient methods for generating representative subsets of satisfying assignments for formulas, but security audits and compliance verification require completeness: every access-admitting pair must be identified, not just a statistically representative sample. Closely related to model generalization, Reynolds et al. [30] reuse counterexamples and unsatisfiability cores across related SMT queries to reduce solver calls in enumerative syntax-guided synthesis. However, their setting involves synthesizing an unknown formula, whereas in our setting the formulas are known and correspond to the SMT encoding of access control policies [1]. Therefore, we reuse satisfying assignments to confirm results across a finite domain

rather than pruning a candidate search space.

The idea of decomposition presented in this paper is similar to the notion of *query planning* performed by database engines when optimizing queries [8]. Specifically, the idea of pushing selects before joins—a fundamental optimization known as predicate pushdown [9]—can be seen as analogous to what our work does when computing principal-resource relations by first *selecting* the candidates based on principal policies and resource policies alone, and then computing the more expensive *"join"* only on the selected candidate pairs. This optimization mirrors how database query optimizers reduce intermediate result sizes by applying filters early in the execution plan [31], thereby minimizing the data that flows through expensive operations.

VII. Conclusion

We have presented a decomposition-based approach for enumerating principal-resource access in cloud environments. Our key techniques, projection-based filtering and model generalization, exploit the structural independence of access control formulas to reduce the number of SMT solver calls required for comprehensive enumeration. Evaluation on synthetic benchmarks shows 2.2–5.1× speedup over the baseline, with filtering effectiveness increasing at larger scales. Production deployment powering a commercial cloud security feature confirms these results: for large environments, our approach eliminates up to 97% of candidate pairs before joint analysis, making comprehensive access enumeration practical for security auditing and compliance verification. We believe model generalization may apply more broadly to settings where many related satisfiability queries share structure, and that further analysis of its effectiveness and the finite-domain model enumeration problem more generally are promising directions for future work. In particular, our current technique reuses concrete satisfying assignments; reusing a more general, symbolic characterization of satisfying assignments is a natural next step. Another direction is incremental analysis. Our system analyzes static snapshots and re-runs analysis periodically to reflect changes, but when only a single policy changes much of the prior computation remains valid. Reusing cached projections and satisfying contexts to avoid recomputation from scratch could substantially reduce the cost of keeping analyses current.

References

[1] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–9.

[2] A. Armando and S. Ranise, "Scalable automated symbolic analysis of administrative role-based access control policies by SMT solving," *Journal of Computer Security*, vol. 20, no. 4, pp. 309–352, 2012.

[3] E. Zahoor, Z. Asma, and O. Perrin, "A formal approach for the verification of AWS IAM access control policies," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 59–74.

[4] A. Chakarov, J. Geldenhuys, M. Heck, M. Hicks, S. Huang, G.-A. Jaloyan, A. Joshi, K. R. M. Leino, M. Mayer, S. McLaughlin, A. Mritunjai, C. Pit-Claudel, S. Porncharoenwase, F. Rabe, M. Rapoport, G. Reger, C. Roux, N. Rungta, R. Salkeld, M. Schlaipfer, D. Schoepe, J. Schwartzentruber, S. Tasiran, A. Tomb, E. Torlak, J.-B. Tristan, L. Wagner, M. W. Whalen, R. Willems, T. Xiang, T. J. Byun, J. Cohen, R. Fang, J. Jang, J. Rath, H. T. Syeda, D. Wagner, and Y. Yuan, "Formally verified cloud-scale authorization," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 2508–2521.

[5] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta *et al.*, "Block public access: trust safety verification of access control policies," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 281–291.

[6] J. Backes, U. Berrueco, T. Bray, D. Brim, B. Cook, A. Gacek, R. Jhala, K. Luckow, S. McLaughlin, M. Menon, D. Peebles, U. Pugalia, N. Rungta, C. Schlesinger, A. Schodde, A. Tanuku, C. Varming, and D. Viswanathan, "Stratified abstraction of access control policies," in *Computer Aided Verification (CAV)*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 165–176.

[7] "Using AWS identity and access management access analyzer," https://docs.aws.amazon.com/IAM/latest/UserGuide/what-is-access-analyzer.html, accessed: January 2026.

[8] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79. New York, NY, USA: Association for Computing Machinery, 1979, p. 23–34.

[9] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 34–43.

[10] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 08 2007.

[11] L. Barnett, L. D'Antoni, A. Goel, R. G. Kıcı, N. Rungta, M. Southern, and C. Sung, "Modeling the AWS authorization engine," in *Formal Methods in Computer-Aided Design (FMCAD)*, A. Irfan and D. Kaufmann, Eds. Vienna: TU Wien Academic Press, 2025, pp. 188–197.

[12] "How AWS enforcement code logic evaluates requests to allow or deny access," accessed: January 2026.

[Online]. Available: https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_evaluation-logic_policy-eval-denyallow.html
[13] "Amazon simple storage service documentation," https://docs.aws.amazon.com/s3/, accessed: January 2026.
[14] "AWS Step Functions documentation," https://docs.aws.amazon.com/step-functions/, accessed: January 2026.
[15] "AWS Lambda documentation," https://docs.aws.amazon.com/lambda/, accessed: January 2026.
[16] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442.
[17] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.
[18] "Amazon DynamoDB documentation," https://docs.aws.amazon.com/dynamodb/, accessed: January 2026.
[19] P. Biswas, R. Sandhu, and R. Krishnan, "A comparison of logical-formula and enumerated authorization policy ABAC models," in *Data and Applications Security and Privacy XXX*, S. Ranise and V. Swarup, Eds. Cham: Springer International Publishing, 2016, pp. 122–129.
[20] T. Kosiyatrakul, S. Older, and S.-K. Chin, "A modal logic for role-based access control," in *Computer Network Security*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 179–193.
[21] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Computer Aided Verification (CAV)*. Berlin, Heidelberg: Springer-Verlag, 2002, p. 250–264.
[22] T. Toda and T. Soh, "Implementing efficient all solutions SAT solvers," *ACM J. Exp. Algorithmics*, vol. 21, Nov. 2016. [Online]. Available: https://doi.org/10.1145/2975585
[23] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT techniques for fast predicate abstraction," in *Computer Aided Verification (CAV)*, T. Ball and R. B. Jones, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 424–437.
[24] Q.-S. Phan and P. Malacaria, "All-solution satisfiability modulo theories: Applications, algorithms and benchmarks," in *2015 10th International Conference on Availability, Reliability and Security*, 2015, pp. 100–109.
[25] J.-M. Lagniez and E. Lonca, "Leveraging decision-DNNF compilation for enumerating disjoint partial models," in *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning*, 2024.
[26] S. Möhle, R. Sebastiani, and A. Biere, "On enumerating short projected models," *Discrete Applied Mathematics*, vol. 361, pp. 412–439, 2025.
[27] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015, pp. 304–319.
[28] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. Association for Computing Machinery, 2018, p. 549–559.
[29] R. Dutra, J. Bachrach, and K. Sen, "GUIDEDSAMPLER: Coverage-guided sampling of SMT solutions," in *2019 Formal Methods in Computer Aided Design (FMCAD)*, 2019, pp. 203–211.
[30] A. Reynolds, H. Barbosa, D. Larraz, and C. Tinelli, "Scalable algorithms for abduction via enumerative syntax-guided synthesis," in *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 141–160. [Online]. Available: https://doi.org/10.1007/978-3-030-51074-9_9
[31] G. Graefe, "The cascades framework for query optimization," *IEEE Data Eng. Bull.*, vol. 18, no. 3, pp. 19–29, 1995.