Quantifying Azure RBAC Wildcard Overreach

Christophe Parisel*

June 13, 2025

Abstract

Azure's Role-Based Access Control (RBAC) leverages wildcard permissions to simplify policy authoring, but this abstraction often obscures the actual set of allowed operations and undermines least-privilege guarantees. We introduce Belshazaar, a two-stage framework that targets both (1) the effective permission set problem—deriving the exact list of actions granted by arbitrary wildcard specifications—and (2) the evaluation of wildcards permissions "spread". First, we formalize Azure action syntax via a context-free grammar and implement a compiler that expands any wildcard into its explicit action set. Second, we define an ultrametric diameter metric to quantify semantic overreach in wildcard scenarios. Applied to Microsoft's official catalog of 15,481 actions, Belshazaar reveals that about 39% of actions admit a cross Resource Provider reach when associated with non obvious wildcards, and that effective permissions sets are effectively computable. These findings demonstrate that wildcard patterns can introduce substantial privilege bloat, and that our approach offers a scalable, semantics-driven path toward tighter, least-privilege RBAC policies in Azure environments.

1 Introduction

Microsoft Azure's Role-Based Access Control (RBAC) system governs access to cloud resources through permission policies (e.g. Microsoft.Compute/* or */read) that specify allowed and denied operations. To simplify authoring of Built-in roles, Microsoft uses wildcard patterns in both `Actions` and `NotActions` clauses, enabling administrators to grant broad categories of Control Plane permissions with concise expressions. A similar approach is followed in the Data Plane, featuring wildcards in both `dataActions` and `NotDataActions` clauses.

Our research also unravels that, through the design of Custom Roles, customers are given more flexibility in wildcards positioning than what Microsoft itself uses for making Built-in roles: they are entitled to place wildcards at infix locations, yielding a wide-ranging, hard to predict, permission reach. A review of all Azure `Actions` shows that, when combined with ill-placed wildcards, 39% have a potential to get an extreme (cross Resource Provider) reach.

Wildcards usage should be restricted to a very limited number of users (break glass accounts) and service principals (CD/CI pipeline). The vast majority of principals should be granted explicit `Actions` because implicit permissions introduce significant security risks: a single asterisk ($\star$) can inadvertently grant access to hundreds of operations, violating the principle of least privilege [1].

1.1 Wildcards expansion in Azure

Current Azure RBAC APIs and Portal tools provide limited visibility into wildcard expansion, leaving security teams unable to:

1. Precisely determine which atomic operations are granted by wildcard patterns
2. Quantify the security impact of over-approximated permissions
3. Detect when Microsoft's service evolution expands existing wildcards
4. Substitute wildcard expressions with explicit permissions while preserving intended functionality

*Email: ch.parisel@gmail.com

1.2 Our Approach

We address these challenges through a formal language-theoretic framework that treats Azure RBAC actions as a grammar generating nodes and leaves in a structured hierarchy tree. Azure's dot-and-slash-delimited naming convention populates a natural tree structure over which we can define precise distance ultrametrics. By reverse-engineering Azure RBAC's wildcards expansion logic, we can generate an arbitrary number of valid wildcard actions that we treat as leaves of this tree. We can enslave this generation process to a genetic algorithm to find the most wide ranging actions with regards their respective locations in the tree.

1.3 Contributions

This paper makes the following contributions:

- **Azure RBAC Actions Grammar**: We propose a simple grammar specification of Azure's actions syntax
- **Wildcards Compiler**: We present Belshazaar, a wildcards expansions compiler based on this grammar
- **Wildcard Risks Quantification**: We define a mathematically principled ultrametric diameter for quantifying permission over-approximation granted by wildcards
- **Automated Identification of Extreme Actions**: We identify the most risky permissions by using a genetic algorithm trained to minimize the ultrametric diameter
- **Open-Source Implementation**: We release practical tooling for enterprise adoption

2 Background and Related Work

2.1 Azure RBAC Architecture

Azure RBAC operates on a hierarchical namespace where segments are delimited by slash. Each operation follows the pattern: `ResourceProvider/OperationsLevel1/.../OperationsLevelN/ActionVerb`

The first and last segments have a special meaning: the first segment declares the resource provider, whereas the last segment holds the actual action verb (read, write, delete, action). All other segments define intermediate levels of nesting. All segments are optional

For example, `Microsoft.Compute/virtualMachines/start/action` represents the action to start a virtual machine within Microsoft's Compute resource provider. Wildcards can appear in any segment except in the last one unless they are the only character in the segment. Here are some example of valid wildcard placements:

- `Microsoft.Compute/*` - All Compute operations
- `Microsoft.*/read` - All read operations across Microsoft Resource Providers
- `*/delete` - All delete operations
- `*` - All operations

2.2 Related Work

Access control analysis has been extensively studied in security literature. Sandhu et al. [2] formalized the RBAC model, while Li et al. [3] identified fundamental limitations in role-based systems.

Cloud-specific access control has received growing attention. AWS has developed Zelkova [10], an internal service that uses automated reasoning to analyze IAM policies. This approach allows AWS to verify that policies meet security requirements and do not grant unintended access. They also open-sourced Cedar [11], which lets Cloud customers perform a similar analysis on their custom IAM policies. Recently, David Kerber released IAM len [12], a tool for evaluating AWS IAM policies offline. While AWS tooling is well covered in the literature, Azure has received little dedicated treatment. One of our recent works [8] focused on assessing lateral motion in Azure Data Plane leveraging a similar ultrametric in Azure Tenant containers hierarchy.

Formal methods in security policy analysis have been explored by Fisler et al. [4]. Jackson [5] introduced the notion of Machine Diameter. We build upon this foundation by applying language theory specifically to hierarchical permission systems.

3 Azure Actions Grammar

Through analysis of what Azure's Custom Roles Definition permits and denies, we reverse-engineered the grammar governing action strings to build our own PLY[9] wildcards compiler.

3.1 Lexical Analysis

We found that modeling azure permissions with only 3 tokens is enough for our needs: WILDCARD, SLASH and TEXT.

- **SLASH**: single / character.
- **WILDCARD**: single asterisk $\star$ character, can appear only once.
- **TEXT**: matches sequences of alphanumerial characters (including dot, dash, underscore, dollar, the curly brackets, excluding all other characters).

Here is the PLY tokens formulation:

```
tokens = ('TEXT', 'WILDCARD', 'SLASH')

t_WILDCARD = r'\*'
t_SLASH    = r'/'

def t_TEXT(t):
    r'[a-zA-Z0-9.-_{}$]+'
    return t

t_ignore = ' \t\n'
```

3.2 Syntactic Analysis

- **SLASH**: acts as segments separator. Can be placed anywhere[1], except as a trailing character.
- **WILDCARD**: can be placed anywhere, except in the last segment unless it is the only character in the last segment. Spans one or more segments.
- **TEXT**: can be placed anywhere. The non-alphanumerical characters are not metacharacters, they can be treated like alphanumerical ones.

3.2.1 Preprocessing rules

To make parsing straightfoward, before running the compiler we ensure that:

1. no more than one wildcard shows up in the input string
2. a wildcard doesn't show up in the last segment, except if is the only character
3. the last segment must be one of read, action, write, delete or a wildcard
4. we ignore input strings without any segments

3.2.2 Syntax rules

```
pattern: segment_list
segment_list : segment
segment_list : segment_list SLASH segment
segment : TEXT
        | WILDCARD
        | TEXT WILDCARD
        | WILDCARD TEXT
        | TEXT WILDCARD TEXT
```

[1] although two slashes in a row and two dots in a row are permitted by Azure, we do not integrate them in our model for simplicity.

3.2.3 Grammar Properties

This grammar exhibits several important properties:

- **Grammar Completeness** Our specification accepts all valid Azure permission strings pulled from Azure Resource Provider API. These permissions are all explicit: they contain no wildcards. At of the time of writing, we validated completeness by parsing all 15,481 control plane actions without syntax errors or illegal characters.

  Our specification accepts wildcards from common built-in Owner and Contributor roles.

- **Wildcard Expressiveness** Through testing of wildcard patterns at various locations, we confirmed that Azure supports:
  - Prefix wildcards: `Microsoft.Stor*` matches `Microsoft.Storage/locations/usages/read`
  - Suffix wildcards: `*Machines/reimage/action` matches `Microsoft.Compute/virtualMachines/reimage/action`
  - Infix wildcards: `Micr*ft.AAD/Operations/read` matches `Microsoft.AAD/operations/read`
  - Complete wildcards: `*` matches any single segment

3.2.4 Production rules

The semantics rules of our compiler are available in appendix.

4 Wildcards Analysis Framework

We now present our framework for analyzing wildcards in Azure RBAC: we treat each wildcard expression as a regular language[6] over the Azure actions alphabet. Using all permissions strings from the Azure Resource Provider API, we are able to infer this alphabet exhaustively. With the alphabet now formulated, we can generate explicitely all `Actions` for each of these languages.

We do the same for `NotActions`. Substracting `NotActions` from an `Action` yields the effective permission set of this `Action`.

4.1 Mathematical Foundations

Azure Operation Universe

Let $\mathcal{U}$ denote the universe of all Azure operations, where each operation $u \in \mathcal{U}$ follows the canonical form:

$$u = \text{segment}/.../\text{segment}/.../\text{actionVerb}$$

Wildcard Language

Given a wildcard expression w, define language $L(w) \subseteq \mathcal{U}$ as the set of all operations matched by w. Formally:

$$L(w) = \{u \in \mathcal{U} : u \text{ matches glob pattern } w\}$$

If u is an Azure permission without a wildcard, $L(u) = u$.

Permission Set

In Azure RBAC, permission sets are defined at the role definition level. A role definition encompasses a set of `Actions` and a set of `NotActions`. Since both sets are entirely independant, to determine the permission set of a single Azure action in the `Actions` set of a role definition, we need to substract all the `NotActions`.

For an `Action` A and `NotActions` $N = \{n_1, n_2, \ldots, n_m\}$, the permission set is:

$$P(A, N) = A \setminus \left(\bigcup_{j=1}^{m} n_j \right)$$

Since P may contain wildcards in A and n_i, P is not effective. We expand all wildcards treating each A and n_i as a sentence of Azure's permissions grammar.

The effective, computable permission set becomes:

$$P_{eff}(A, N) = L(A) \setminus \left(\bigcup_{j=1}^{m} L(n_j) \right)$$

4.2 Ultrametric Distance

To quantify permission over-approximation induced by wildcards, we leverage Azure's hierarchical namespace structure and define its ultrametric distance.

Hierarchy Tree

Model Azure's permission namespace as a tree T where:

- Root (level 0) represents the global namespace
- Level 1 nodes represent Microsoft resource providers (e.g., `Microsoft`)
- Level 2 nodes represent the sub providers (e.g., `Storage`)
- Level n nodes represent resource types and operations groups (e.g., `storageAccounts`)
- Leafs represent action verbs (e.g., `read`)
- Levels are segmented using a separator[2]

Ultrametric Distance

For operations $u, v \in \mathcal{U}$, define:

$$d(u, v) = \text{depth}(\text{LCA}(u, v))$$

where $\text{LCA}(u, v)$ is the lowest common ancestor of u and v in tree T. Root has depth 0, Microsoft resource providers have depth 1, etc.

Readers familiar with[8] will notice that the ultrametric is linear in the present case.

Here is a first example:

Let $u = Microsoft.ApiCenter/services/workspaces/analyzerConfig/analysisExecutions/read$
and $v = Microsoft.ApiCenter/deletedServices/delete$

The depth of u in T is 7, the depth of v is 4. Their LCA is `Microsoft.ApiCenter` which sits at depth 2. Hence, their distance is 2.

Here is another example:

Let $u = Microsoft.BotService/botServices/channels/providers/Microsoft.Insights/diagnosticSettings/read$
and $v = Microsoft.BotService/botServices/channels/providers/Microsoft.Insights/logDefinitions/read$

The depth of u and v in T is 9 (notice the two dots in u and v, each dot delimitate a segment).

Their LCA is `Microsoft.BotService/botServices/channels/providers/Microsoft.Insights` which sits at depth 7. Hence, their distance is 7.

[2]We use both the slash and the dot to delimitate segments. While using dot is completely optional, it provides a finer granularity than using slashes alone.

4.3 Quantificatying over-approximations

Diameter

For a permission set P, define its diameter as:

$$Diam(P) = \min_{u \neq v \in P} d(u, v)$$

This metric quantifies the "spread" of actions across Azure's Resource Providers hierarchy. Large diameter (small distance between pair of actions) indicates tightly scoped permissions, while small diameter (large distance between pairs) suggests over-approximation.

The reason why we define diameters with a min and not a more conventional max is that we use a linear ultrametric distance.

5 Experimental Evaluation

5.1 Compiler

The python compiler we made for the Azure permissions grammar is called Belshazaar[13]. When provided with an `Action` and a list of `NotActions`, Belshazaar reads a cache containing all Azure actions and expands the `Action` and the `NotActions`. It returns the effective permission set of the action.

Here is an example of a run with $A = Microsoft.AAD/*$ and $N = \{Microsoft.AAD/*/read, Microsoft.AAD/*/delete\}$

```
belshazaar.py --action 'Microsoft.AAD/*' --notActions 'Microsoft.AAD/*/read,Microsoft.AAD/*/delete'

  Microsoft.AAD/domainServices/oucontainer/write
  Microsoft.AAD/domainServices/write
  Microsoft.AAD/register/action
  Microsoft.AAD/domainServices/providers/Microsoft.Insights/diagnosticSettings/write
  Microsoft.AAD/unregister/action
```

The effective permission set of $A = Microsoft.AAD/*$ accounting for `NotActions` is made of 5 explicit permissions, which Belshazaar enumerates.

5.2 Wildcards Generation

We wrote a wildcard insertion function which replaces a random subsequence of any (wildcard-action) azure action in the official Resource Provider list with a wildcard, abiding to the rules of the actions grammar. The exact extent of pattern globbing is identified by the first and last position of the subsequence within the sequence.

Here are a few examples of wildcard generated permissions:

Original action	Generated action	first	last
Microsoft.Blueprint/blueprintAssignments/write	Microsoft.Blueprint/bluepr*s/write	26	38
Microsoft.OperationalInsights/clusters/operationresults/read	Microsoft.Operati*s/read	17	53
Microsoft.Network/networkManagers/routingConfigurations/ruleCollections/rules/delete	Microsoft.Network/networkMana*llections/rules/delete	28	61

Table 1: Examples of valid wildcard actions (second column) produced by random generation from valid actions (first column)

5.3 Extreme Pairs Generation

To identify highly permissive wildcard patterns in Azure's action space, we developed a wildcard generation pipeline that combines randomized sampling with evolutionary optimization.

5.3.1 Initial Population

For each official Azure action string, we first constructed an initial population of 50 candidate wildcard patterns. Wildcards were inserted at random positions within the action string, subject to syntactic constraints designed to avoid trivial global patterns such as `*` or `Microsoft.*`.

The following rules were enforced:

- The wildcard insertion point must occur at least 3 characters after the Resource Provider dot (the `.` separator following `Microsoft`).
- Wildcards may not split the final segment of the action unless they fully replace it.

This ensures the generation of structurally meaningful wildcards that reflect nontrivial permission generalizations.

Examples:

1. `Microsoft.Net*/...` is valid because the wildcard begins at least 3 characters after the dot.
2. `Microsoft.O*ions/...` is invalid because the wildcard occurs only 2 characters after the dot.

5.3.2 Genetic Algorithm Optimization

After generating the initial population, we applied a Genetic Algorithm (GA) to evolve these wildcard populations over 50 generations. Each population consisted of 50 individuals associated with one original Azure action string.

At each generation, the following steps were performed:

1. **Expansion and Fitness Evaluation:** Each wildcard candidate was expanded using Belshazaar's grammar-based expansion engine, producing the full set of concrete Azure actions it matches. The *Ultrametric Diameter* of each expansion was computed using the linear distance model introduced earlier. The diameter served as the fitness function, where smaller diameters indicate broader wildcard reach (i.e., expansions containing semantically more distant actions).
2. **Selection:** The population was ranked by diameter (ascending), and the top 50% of candidates were selected as survivors for reproduction.
3. **Mutation and Reproduction:** New candidates were produced by randomly mutating either the start or end position of the wildcard insertion interval. Mutations involved shifting positions by a random offset. Every mutated candidate was again validated against the original syntactic constraints before inclusion in the population.

This iterative process allowed the algorithm to explore the search space of wildcard placements while avoiding trivial or degenerate cases.

Our evaluation covered all 15,481 Azure actions retrieved using `az provider operation list` on 06 June, 2025.

The full evolutionary search was executed across all official Azure actions, requiring approximately 5 hours of computation on a standard desktop machine. The output consists of, for each Azure action, one or more wildcard patterns whose expansions exhibit the smallest ultrametric diameters. These extreme wildcard patterns represent dangerous generalizations in terms of permission breadth.

5.3.3 Distribution of Diameters

Median Diameter Computation Because ultrametric diameters take on discrete integer values, we report the median via linear interpolation on the empirical cumulative distribution. After sorting the diameter counts and computing the cumulative percentages, we locate the two adjacent diameter values that straddle the 50% threshold and interpolate between them. This yields a more precise median of 1.26 rather than an integer.

This median value of Diameters (Figure 1) is extremely low. It is surprising since we took provisions to avoid common obvious wildcards. Recall that a diameter of 1 is indicative of significant over-permissioning wildcard configurations.

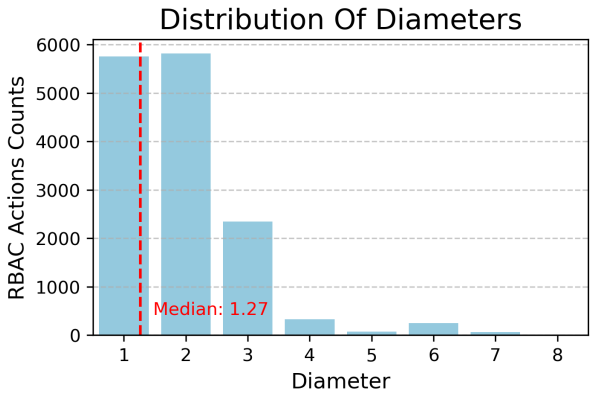

Figure 1: Distribution of Diameters

Wildcard permission	Left pair	Right pair
Microsoft.Api*/write	Microsoft.ApiCenter/services/apis/versions/securityRequirements/write	Microsoft.ApiManagement/gateways/configConnections/write
Microsoft.Aut*s/delete	Microsoft.Authorization/classicAdministrators/delete	Microsoft.Automation/automationAccounts/certificates/delete
Microsoft.Azur*er/action	Microsoft.AzureActiveDirectory/register/action	Microsoft.AzureArcData/register/action
Microsoft.Azure*n/action	Microsoft.AzureDataTransfer/pipelines/approveConnection/action	Microsoft.AzureLargeInstance/AzureLargeInstances/shutdown/action
Microsoft.Cer*ister/action	Microsoft.CertificateRegistration/register/action	Microsoft.Certify/register/action
Microsoft.Compu*egister/action	Microsoft.Compute/register/action	Microsoft.ComputeSchedule/register/action
Microsoft.Contai*start/action	Microsoft.ContainerInstance/containerGroups/restart/action	Microsoft.ContainerService/fleets/updateRuns/start/action
Microsoft.Con*ts/delete	Microsoft.Confluent/organizations/environments/delete	Microsoft.ConnectedOpenStack/networkPorts/delete
Microsoft.Containe*s/delete	Microsoft.ContainerInstance/containerGroupProfiles/delete	Microsoft.ContainerRegistry/registries/agentpools/delete
Microsoft.Dev*finitions/read	Microsoft.DevCenter/devcenters/catalogs/environmentDefinitions/read	Microsoft.Devices/IotHubs/logDefinitions/read
Microsoft.Data*es/write	Microsoft.DataBoxEdge/dataBoxEdgeDevices/bandwidthSchedules/write	Microsoft.DataFactory/datafactories/datapipelines/write
Microsoft.Edg*/write	Microsoft.Edge/capabilityLists/write	Microsoft.EdgeMarketplace/locations/operationStatuses/write
Microsoft.Hardwar*ers/delete	Microsoft.Hardware/orders/delete	Microsoft.HardwareSecurityModules/cloudHsmClusters/delete
Microsoft.Kubern*s/write	Microsoft.Kubernetes/locations/operationstatuses/write	Microsoft.KubernetesConfiguration/extensions/write
Microsoft.IoT*/write	Microsoft.IoTCentral/IoTApps/privateEndpointConnectionProxies/write	Microsoft.IoTFirmwareDefense/firmwareGroups/firmwares/write
Microsoft.Man*oups/write	Microsoft.ManagedNetworkFabric/neighborGroups/write	Microsoft.Management/managementGroups/write
Microsoft.Net*ps/delete	Microsoft.NetApp/netAppAccounts/accountBackups/delete	Microsoft.Network/adminNetworkSecurityGroups/delete
Microsoft.Operat*s/read	Microsoft.OperationalInsights/clusters/operationresults/read	Microsoft.OperationsManagement/managementassociations/read
Microsoft.Netw*nces/delete	Microsoft.Network/networkSecurityPerimeters/linkReferences/delete	Microsoft.NetworkCloud/storageAppliances/delete

Table 2: A sample of extreme (diameter 1) pairs produced by our generator

5.3.4 Wide-ranging wildcards

Tables 2 and 3 provide a small excerpt of extreme pairs generated by wildcards expansion. Table 2 shows the most extreme pairs, featuring a diameter of 1. Table 3 shows pairs featuring diameter 2.

The full list is available on github[7].

6 Limitations and Future Work

6.1 Current Limitations

- **Scope**: Limited to Azure RBAC; it does not analyze conditional access or resource-level permissions
- **Semantics**: Belshazaar focuses on syntactic analysis; does not model operational security impact

6.2 Future Directions

- **Multi-Cloud**: Extend framework to AWS IAM and Google Cloud IAM
- **Policy Synthesis**: Automatically generate least-privilege roles from usage data

Wildcard permission	Left pair	Right pair
Microsoft.AAD/*tions/read	Microsoft.AAD/Operations/read	Microsoft.AAD/domainServices/providers/Microsoft/Insights/logDefinitions/read
Microsoft.AVS/*/action	Microsoft.AVS/privateClouds/listAdminCredentials/action	Microsoft/AVS/register/action
Microsoft.ApiCenter*s/delete	Microsoft.ApiCenter/deletedServices/delete	Microsoft.ApiCenter/services /apis/versions/securityRequirements/delete
Microsoft.Authorization/policy*s/write	Microsoft.Authorization/policyAssignments /privateLinkAssociations/write	Microsoft.Authorization/policyDefinitions/versions/write
Microsoft.Blueprint/bl*/write	Microsoft.Blueprint/blueprintAssignments/write	Microsoft.Blueprint/blueprints/artifacts/write
Microsoft.Cdn/*cies/delete	Microsoft.Cdn/cdnwebapplicationfirewallpolicies/delete	Microsoft.Cdn/profiles/securitypolicies/delete
Microsoft.Cog*s/write	Microsoft.CognitiveServices/accounts/capabilityHosts/write	Microsoft.CognitiveServices/attestations/write
Microsoft.Comp*s/action	Microsoft.Compute/disks/beginGetAccess/action	Microsoft.ComputeSchedule/autoActions/attachResources/action
Microsoft.ContainerSer*snapshots/write	Microsoft.ContainerService/managedclustersnapshots/write	Microsoft.ContainerService/snapshots/write
Microsoft.DBforPostgr*n/action	Microsoft.DBforPostgreSQL/assessForMigration/action	Microsoft/DBforPostgreSQL/ flexibleServers/tuningOptions/startSession/action

Table 3: A sample of diameter 2 pairs produced by our generator

7 Conclusion

In this paper, we presented **Belshazaar**, a semantics–aware framework for rigorously expanding and evaluating Azure RBAC wildcards. By formalizing the complete Azure action language as a context-free grammar and building a compiler to derive minimal explicit permission sets, Belshazaar delivers an accurate "effective permissions" view for any wildcard specification. Our linear ultrametric diameter model then quantifies the semantic overreach of these wildcards, showing that approximately 39% of patterns span more than two grammar levels.

We empirically validated Belshazaar on Microsoft's catalog of 15,481 actions, demonstrating that it scales to real-world workloads and yields actionable insights into privilege bloat.

Looking forward, integrating Belshazaar into policy-authoring and continuous-monitoring pipelines will enable administrators to detect and remediate over-privileged roles before deployment. We may extend our ultrametric analysis with user-guided refinement strategies that automatically suggest least-privilege replacements for low-diameter patterns. We believe Belshazaar paves the way toward bridging the gap between policy expressiveness and security assurance in large-scale cloud environments.

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of IEEE*, vol. 63, no. 9, pp. 1278-1308, 1975.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *Computer*, vol. 29, no. 2, pp. 38-47, 1996.

[3] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust-management framework," *Proceedings IEEE Symposium on Security and Privacy*, pp. 114-130, 2002.

[4] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," *Proceedings International Conference on Software Engineering*, pp. 196-205, 2005.

[5] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

[6] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Pearson, 2006.

[7] Christophe Parisel, A dump of all Azure action diameters as of June 06, 2025 `https://github.com/labyrinthinesecurity/silhouette/blob/2.1/formal/azureDiameters.txt`

[8] Christophe Parisel, Scoring Azure permissions with metric spaces `https://arxiv.org/abs/2504.13747`

[9] PLY (PYthon Lex-Yacc), `https://ply.readthedocs.io/en/latest/`

[10] How AWS uses automated reasoning to help you achieve security at scale, `https://aws.amazon.com/blogs/security/protect-sensitive-data-in-the-cloud-with-automated-reasoning-zelkova/`

[11] Cedar, a policy language and evaluation engine, `https://github.com/cedar-policy`

[12] David Kerber, IAM Lens, https://github.com/cloud-copilot/iam-lens

[13] Christophe Parisel, Belshazaar, an Azure RBAC actions compiler, https://github.com/labyrinthinesecurity/silhouette/blob/2.1/formal/README.md

Appendix

Production rules

```
pattern : segment_list
        p[0] = p[1]

segment_list : segment
        p[0] = p[1]

segment_list : segment_list SLASH segment
        p[0] = p[1] + "/" + p[3]

segment : TEXT
        p[0] = re.escape(p[1])

        | WILDCARD
        p[0] = ".*"

        | TEXT WILDCARD
        p[0] = re.escape(p[1]) + ".*"

        | WILDCARD TEXT
        p[0] = ".*" + re.escape(p[2])

        | TEXT WILDCARD TEXT
        0] = re.escape(p[1]) + ".*" + re.escape(p[3])
```