UNIVERSITY OF BUCHAREST

FACULTY OF
MATHEMATICS AND
COMPUTER SCIENCE

SECURITY & APPLIED LOGICS

Dissertation Thesis

GRAPH ANALYSIS OF AWS IAM USING NEO4J

Graduate
Richițeanu Mihai-Sebastian

Scientific coordinator
Conf. Univ. Dr. Ing. Paul Irofti

Bucharest, September 2024

Rezumat

Această lucrare are ca obiectiv principal dezvoltarea unui model de graf destinat mapării complete a unei configurații AWS IAM (Identity and Access Management). Modelul propus urmărește să ofere o reprezentare detaliată și coerentă a tuturor entităților IAM, inclusiv utilizatori, grupuri, roluri și politici, precum și relațiile dintre acestea.

În plus, lucrarea explorează modul în care acest model grafic poate fi utilizat pentru a efectua verificări și teste împotriva posibilelor breșe de securitate care ar putea compromite integritatea și confidențialitatea resurselor. Se analizează, de asemenea, modul în care modelul poate contribui la identificarea și implementarea de îmbunătățiri în configurația IAM, optimizând astfel securitatea și eficiența gestionării accesului.

Un alt aspect esențial al lucrării este prezentarea unui instrument specializat, denumit *IAM-Sentinel*, care are ca scop integrarea între datele de configurație IAM și baza de date Neo4j utilizată pentru stocare și analiză.

Abstract

This paper's main objective is to develop a graph model designed for the comprehensive mapping of an AWS IAM (Identity and Access Management) configuration. The proposed model aims to provide a detailed and coherent representation of all IAM entities, including users, groups, roles, and policies, as well as the relationships between them.

Furthermore, the paper explores how this graph model can be used to perform checks and tests against potential security breaches that could compromise the integrity and confidentiality of resources. It also examines how the model can aid in identifying and implementing improvements in IAM configuration, thereby optimizing the security and efficiency of access management.

Another essential aspect of the paper is the presentation of a specialized tool, named *IAM-Sentinel*, which is intended to integrate IAM configuration data with the Neo4j database used for storage and analysis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

By offering profound insights into relationships and access patterns inside an organization's infrastructure, graph analytics on AWS IAM can significantly increase security and operational efficiency. Organizations can visually represent complicated access restrictions, see possible security threats, and find misconfigurations or improper privileges that might not be obvious through typical analytical approaches by applying graph analytics to IAM data. Graph analytics, for example, can map out user roles, privileges, and interactions in order to find excessive permissions or strange access patterns, improving audits and compliance checks. By identifying duplicate or conflicting permissions, it also helps to optimize access policies, which ultimately contributes to a more secure and efficient IAM environment.

1.2 Contributions

In this research, I propose a graph model which maps complete information about all IAM users, groups, roles and policies in an Amazon Web Services account, including their relationships to one another. The model is designed to detect anomalies and conduct graph analytics.

Moreover, it includes the development of an automated tool, called *IAM-Sentinel*, that integrates the AWS CLI *get-account-authorization-details* command's output to efficiently and reliably structure and map the data into Neo4j. The source code can be found on GitHub: https://github.com/SebastianRichiteanu/IAM-Sentinel.

Lastly, I provide a set of analytics made possible by such model and demonstrate how it can be utilized for advanced graph analysis.

1.3 Structure

The paper is structured into three distinct sections to provide a comprehensive overview of the research and contributions. The first section introduces the proposed model, detailing its design and theoretical framework for analyzing AWS IAM configurations. The second section focuses on the development and implementation of a tool based on this model, highlighting the technical aspects and practical considerations of bringing the model to operational use. Finally, the third section explores the application of graph analytics, demonstrating how these advanced techniques are employed to enhance the analysis of IAM data, uncovering insights and identifying potential security vulnerabilities. Moreover, in the last phase of this section, I present the results obtained and the limitations of the proposed model, alongside a retrospective analysis and suggestions for potential improvements that could enhance the effectiveness and applicability.

Chapter 2

Preliminary

2.1 AWS

Amazon Web Services (AWS) was launched in 2006, and brought high-performance computing resources to the IT world, making them accessible to a broad audience. This innovation has enabled companies of all sizes to grow and innovate at an accelerated pace.

AWS offers a diverse range of services, including storage, networking, and security features, all accessible via the internet but one of its most notable security offerings is AWS Identity and Access Management (IAM).

2.2 AWS IAM

"AWS Identity and Access Management (IAM) is a powerful security tool within Amazon Web Services (AWS) that allows you to manage access to AWS resources securely. IAM enables you to create and control user accounts, groups, and roles and also to assign specific permissions to each based on their need to access and manage resources." [20]

The key components of AWS IAM are:

- Users: They represent individuals which can interact with AWS services. Each user has its own set of permissions and credentials.
- Groups: They are collections of users and can help manage permissions for multiple users at once.
- Roles: They are identities which Users can assume. They come with their own set of permissions and are useful for granting temporary access.
- Policies: Policies are JSON objects that define permissions for actions on AWS resources. They can specify which actions are permitted or denied and to which resources it applies to, alongside which conditions that have to be met.

- Actions: Actions refer to operations that can be performed on AWS resources, such as S3, EC2, RDS, and others. Access can be granted for full control on resources, but it can also be granted granularly, for a specific operation.

Format of Actions

According to AWS Documentation [2], the format of actions is designed to clearly define the specific operations that can be performed on resources. Actions are specified as strings and follow the next standardized format:

1. Service Namespace: The AWS service which is the subject of the action
2. Action Name: The specific operation that is being granted or denied, a verb that describes the actions' functionality.
3. Separator: Both of the above parts are separated by a colon.

Moreover, the wildcard *(asterisk) is used to represent all possible actions for a service, or across all services. This facilitates allowing full access to resources.

2.3 AWS ARN

Amazon Resource Name (ARN) is a unique identifier to refer to specific resources. "ARNs follow a standardized format and provide a way to uniquely identify and access resources across AWS services. The format typically includes the service, region, account ID, resource type, and resource identifier." [2]

2.4 get-account-authorization-detail

get-account-authorization-details is an AWS CLI command used to retrieve detailed information about the IAM resources in an AWS account. This command provides a comprehensive snapshot of the account's IAM configuration, including information about users, groups, roles, policies, and their associated permissions. The output of this command is a JSON object, which contains the IAM details of the entire account.

2.5 Neo4j

Neo4j is a graph database management system designed to handle and query connected data, it is open-source and java based.

"Neo4j is considered the reference software in the area of graph structured storage, but also in the area of graph analytics. It served purpose for multiple areas of science

and research, such as health, government, automotive production, military area, among others." [10]

Cypher is the query language used in Neo4j and it was designed specifically for working with graphs. Originally it was developed as the query language for Neo4j but it has been adopted beyond Neo4j due to its ability to handle complex graph queries.

Neo4j also introduces labels which are used to categorize nodes. They make it easier to query and manage graph data, but also to organize nodes that are distinct entities. For example, you might use labels like User for nodes representing people and Group for nodes representing groups of people.

Libraries like Graph Data Science (GDS) and Awesome Procedures on Cypher (APOC) extend the capabilities of Neo4j. GDS focuses on analytics, offering graph algorithms for pattern recognition, network analysis. APOC provides utilities to simplify data integration, transformation but also improves the querying capabilities.

2.6 Neo4j GDS

"The GDS (Graph Data Science) library was first released in 2020. It is the successor to the Graph Algorithm plugin, and, it contains tools to be used in data science projects." [19]

GDS includes a wide range of tools, featuring path-related algorithms like A* and Dijkstra, machine learning models and pipelines. It offers support for, but is not limited to, the following:

2.6.1 Centrality algorithms

Centrality algorithms analyze the position and connections of nodes in a graph to determine the importance of each node. Common metrics include Closeness centrality, which evaluates direct connections to other nodes; Betweeness Centrality, which measure how important a node is in all the shortest paths between other nodes; and Closeness Centrality which evaluates how close to the "center" of the graph the nodes is, this is accomplished by calculating the distance to all other nodes. These metrics help identify key nodes in graphs, such as critical nodes within communictaion networks or influential individuals within social networks.

2.6.2 Community detection algorithms

To identify clusters or groups of nodes within a graph one can use Community detection algorithms. These algorithms separate nodes into communities by using their connections and aid in revealing the structure of the network. Some algorithms that are

frequently used are Label Propagation and the Louvain Method. This algorithms are useful to understand the modular organization of complex systems such as social networks and organizational structures.

2.6.3 Graph Projections

In Neo4j, the database itself contains a "native" graph, where nodes and relationships are stored. However, this graph might contain more data than you need for specific analyses. Graph projections allow users to create simplified and optimized versions of the graph (or sub-graph) that only include the relevant nodes, relationships and properties used for a specific analysis. By doing so, the computational cost is reduced and algorithms run more efficiently, resulting in better results and shorter waiting times.

2.7 Graph algorithms

2.7.1 Degree Centrality

Degree Centrality (C_D) is a network analysis metric that measures the importance of a node based on the number of direct connections it has. Due to its simple formula, it's one of the simplest centrality measures and it is useful for identifying the most connected nodes in a network.

Definition 2.7.1 (Degree Centrality [9])**.** Degree centrality measures the number of edges (connections) a node has to other nodes in the network. In an undirected graph, this is simply the count of all edges connected to the node. In a directed graph, degree centrality is split into two types:

- In-Degree Centrality: The number of incoming edges to a node.
- Out-Degree Centrality: The number of outgoing edges from a node.

The degree centrality of a node v within a graph $G = (V, E)$ can be calculated with the simple formula $C_D(V) = deg(v)$. Where V denotes the nodes (vertices), E the edges within the graph and $deg(v)$ represents the number of edges connected to node v.

■

The above formula can be modified to use the in-degree or out-degree to calculate the respective centrality, depending on the context.

2.7.2 Betweenness Centrality

"Betweenness centrality (C_B) is a key metric in network analysis that measures a node's significance based on its role as an intermediary in the shortest paths between other nodes." [9]

This measure is very effective in identifying nodes that control the flow of information or resources within the network, making them critical to maintaining network connectivity and communication.

Definition 2.7.2 (Betweenness Centrality [9])**.** The betweenness centrality of a node can be computed as: $C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$

In the above formula, σ_{st} represents the total number of shortest paths between the two nodes s and t and $\sigma_{st(v)}$ denotes the number of such shortest paths that pass through the node v.

■

Shortest paths in a graph are calculated using algorithms that determine the minimum distance between pairs of nodes. The goal is to find the path between two nodes that traverses the fewest edges. Several algorithms are commonly used to compute these shortest paths, but according to the Neo4j documentation [15], the Graph Data Science (GDS) library does "not compute the exact betweenness centrality for all nodes due to the high computational cost associated with this task. Instead, it utilizes *Brandes' approximate algorithm* to provide an efficient estimation of betweenness centrality values."

Brandes' approximate algorithm

"The approximate algorithm reduces the computation by calculating betweenness centrality based on a subset of all possible shortest paths rather than all-pairs shortest paths. By selecting a random sample of source nodes, or more deterministic ways like MaxMin, MaxSum, etc., from the graph and computing the shortest paths from these nodes to all others, the algorithm approximates the betweenness centrality scores." [5]

This method offers a trade-off between accuracy and computational efficiency, both of which are influenced by the number of sample nodes that are part of the source subset.

2.7.3 Closeness Centrality

Closeness centrality (C_C) is a measure used to evaluate the proximity of a node to all other nodes in a graph. It is based on the concept that nodes which are closer to other nodes, in terms of shortest path distance, are more central or influential.

Definition 2.7.3 (Closeness Centrality [9])**.** The closeness centrality of a node v is defined as the inverse of the average shortest path distance from v to all other reachable nodes,

and can be calculated with the formula: $C_C(v) = \frac{1}{\sum_{t \in V} d(v,t)}$, whereas $d(v,t)$ represents the shortest path distance between node v and t, where the sum is over all nodes t that can be reached from the node v.

■

However, according to Neo4j documentation [14], "the implementation uses a normalized version of this centrality, namely *Wasserman-Faust normalized closeness* for undirected graphs, where for directed ones the implementation is defined alternatively".

Definition 2.7.4 (Wasserman-Faust Normalized Closeness [14])**.** Instead of using the distance from v to every other node, the distance from every other node to v is taken into account.

$C_{C,\text{norm}}(v) = \frac{(n-1)^2}{(|V|-1) \times \sum_{u \in V \setminus \{v\}} d(v,u)}$

■

2.7.4 PageRank

"PageRank is an algorithm used to rank the importance of nodes in a graph, most famously applied by Google to rank web pages in search engine results." [6]

PageRank calculates a value for each node, the value representing the likelihood that a random walker would reach that node.

Definition 2.7.5 (PageRank [6])**.** In a directed graph $G = (V, E)$ the PageRank of a node can be calculated with: $PR(v) = (1-d) + d \sum \frac{PR(u)}{C(u)}$, having:

- d as the dumping factor. It is a number between 0 and 1, but is usually set to 0.85
- u as all the pages(nodes) that point towards v
- $C(u)$ as the number of outgoing links (out-degree) of node u

■

2.7.5 Louvain

"The Louvain method is an algorithm to detect communities in large networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities." [4]

The algorithm works in two phases that are repeated:

- Modularity Optimization: Each node is associated with its own community and then, for each node, the algorithms considers "moving" it to the communities of its neighbours, calculating the modularity gain for each. The node is then "moved" to the community where the modularity gain was the highest.

- Community Aggregation: After all nodes have been handled, the communities are aggregated into supernodes. In other words, the network is now a new, smaller network where each node represents an entire community from the previous phase, afterwards, the first phase is repeated on this new network.

As stated above, the Louvain algorithm aims to optimize the modularity (Q) of the partition.

Definition 2.7.6 (Modularity [4])**.** The modularity of a node is given by the formula: $Q = \frac{1}{2m} \sum_i^N \sum_j^N \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j)$, where:

- m: is the sum of all weights or the total number of edges in the graph, in an unweighted graph
- N: is the number of nodes in the graph
- A: is the adjency matrix of the graph, i.e. $A_{ij} = 1$ if there is an edge between nodes i and j, 0 otherwise
- k_i: is the sum of the weights attached to the edges of node i, or the degree of node i in a unweighted graph
- c_i: is the community to which node i belongs to
- $\delta(c_i, c_j)$: is the Kronecker delta, 1 if nodes are in the same community, 0 otherwise

■

The approach discussed above primarily applies to undirected graphs. For directed graphs, modifications to the modularity formula are necessary. Specifically,

Definition 2.7.7 (Modularity for Directed Graphs [4])**.** The function k is divided into two functions: k_{out} (out-degree) and k_{in} (in-degree). Additionally, the division in the modularity formula is adjusted to account for the total number of edges, resulting in the new formula: $Q = \frac{1}{m} \sum_i^N \sum_j^N \left[A_{ij} - \frac{k_i^{\text{out}} k_j^{\text{in}}}{m} \right] \delta(c_i, c_j)$

■

2.7.6 Label Propagation

"Label Propagation Algorithm (LPA) is an iterative algorithm that detects communities by propagating labels through the network. Each node starts with a unique label, and during each iteration, it updates its label to the most frequent label among its neighbors, choosing randomly if a tie occurs. The process continues until the labels stabilize, i.e. no more changes happen." [16]

2.7.7 Weakly Connected Components

A Weakly Connected Component of a directed graph is a maximal sub-graph where all nodes are reachable from one another when treating the graph as undirected. This can be useful to understand how nodes are linked, even if the connections are indirect, and it also aids in finding out communities of nodes that can not reach each other.

2.7.8 Triangle Count and Local Clustering Coefficient

"The triangle count refers to the number of triangles present in a graph. A triangle in a graph is a set of three vertices that are all connected by edges, forming a closed loop or cycle of length three." [1]

"The Local Clustering Coefficient is a measure that quantifies the likelihood that a node's neighbors are also neighbors with each other." [18] In other words, it reflects how tightly a node's neighborhood is interconnected.

Definition 2.7.8 (Clustering Coefficient [18])**.** For a node v, that is part of an undirected graph $G = (V, E)$, the local clustering coefficient $C(v)$ can be calculated with the following formula: $C(v) = \frac{2T_v}{d_v(d_v-1)}$, where T_v is the Triangle Count of the node v and d_v is the degree of the node. ■

High clustering could indicate redundant or overlapping permissions among users or roles, which could be optimized for better security and efficiency.

2.8 Go

"Go, created by Google, is an open-source, statically typed, compiled programming language designed for simplicity and efficiency." [12] Go offers built-in support for concurrent programming through go routines, which makes the language well-suited forr developing scalable and high-performance applications. Go is highly regarded for its speed and simplicity in deployment, making it a popular choice for cloud services, distributed systems, and server-side applications.

2.9 Docker

"Docker is a platform that simplifies application deployment by using containers to encapsulate applications and their dependencies. Docker images serve as the blueprint for these containers, containing everything needed to run an application. By leveraging Docker images, users can ensure that applications run consistently across different environments." [7]

Chapter 3

Proposed Model

In the proposed graph model for AWS Identity and Access Management (IAM), the primary entities are represented as nodes: users, groups, roles, policies, and actions. This model utilizes directed edges to capture the relationships between these nodes. Each node encapsulates properties like ARN, creation date, path, attachable status, etc. derived from the detailed IAM data presented in the section 2.4.

3.1 Nodes

Different types of nodes encapsulate different types of properties that are proprietary to their type, but ARNs are utilized as unique identifiers. Each node, with the exception of action type nodes, is indexed by its ARN to ensure precise and consistent identification of all resources within the graph database. Actions are indexed by the action string.

3.1.1 Nodes attributes

Beside the ARN identifier, all nodes share two more common properties, *Path* and *CreateDate*.

- Path: Indicates the hierarchical structure or path of the IAM user
- CreateDate: Specifies the timestamp when the IAM user was created

The subsequent subsections will delve into detailed descriptions of these fields, explaining their specific roles and how they contribute to the overall model.

Users

- UserName: Denotes the name of the IAM user
- UserId: Represents the unique identifier for the IAM user
- GroupsList: Specifies the groups that the user is being part of

Groups

- GroupName: The name of the IAM group
- GroupId: The unique identifier for the IAM group

Roles

- RoleName: The name of the IAM role.
- RoleId: The unique identifier for the IAM role.

Policies

- PolicyName: The name of the managed policy.
- PolicyId: The unique identifier for the managed policy.
- AttachementCount: The number of entities (users, groups, roles) to which the policy is attached.
- IsAttachable: Indicates whether the policy can be attached to entities
- DefaultVersionId: The ID of the default version of the policy.
- UpdatedDate: The date and time when the policy was last updated.

Actions

- Action: The action string under the standardized AWS format, as described in section 2.2.

3.2 Relationships

Relationships in the IAM graph model show how nodes interact and are connected with each other within the access control framework. These relationships establish the connections between nodes, illustrating how different components are related.

Because the focus is on Action nodes, they occupy the lowest level in the ”hierarchy”, with no outgoing edges. All relationships flow ”downwards” towards them, starting from higher-level entities like groups at the top, followed by roles and users, then policies, and finally ending at actions.

In the proposed model, there are four different type of relationships:

HAS_MEMBER

A group is associated with an user by the HAS_MEMBER relationship, which indicates that the user belongs to the designated group. Since it creates the link between specific users and their groups, this connection is essential for managing and organizing people inside the IAM ecosystem. The HAS_MEMBER relationship defines this link and helps in enforcing group-based rules, organizing access restrictions, and making it easier to manage user rights based on group memberships.

MANAGES

The MANAGES relationship establishes a connection between a user and a policy or between a role and a policy, indicating that the user or role is capable of using the specific policy. This relationship also highlights the capability of the user or role to create, modify, or apply the policy. By defining this relationship, it becomes clear who is responsible for handling policies and which entities are capable of using them, thus facilitating better control and oversight of policy administration and ensuring that permissions and access controls are effectively managed within the system.

CAN_ASSUME

A role and an action are linked by the CAN_ASSUME connection, which shows that the role is authorized to carry out the specific action. This relationship defines the range of activities that a certain position is permitted to perform, therefore defining the role's capabilities inside the system. Through the establishment of this link, the CAN_ASSUME relationship guarantees that roles are authorized to do different tasks, facilitating efficient administration and implementation of access controls.

TRUSTS

By using the TRUSTS relationship to link a policy or an user to an action, it signifies that the policy or user has been granted trust or authorization to carry out the specified action. This relationship highlights the level of trust or permission assigned, highlighting which actions are permissible based on the established trust between the entities. By defining this connection, the relationship clarifies the scope of authority and access granted, thus assuring that actions are carried out in compliance with established trust and permission policies as well as assisting with the implementation of fine-grained access control.

3.2.1 Relationships attributes

Relationships linking to nodes of type Action will always include an attribute named effect. This attribute is limited to two possible values: *Allow* or *Deny*, depending on the nature of the action performed. [3] As previously stated, the relationships that can connect to nodes of type Action are TRUSTS and CAN_ASSUME.

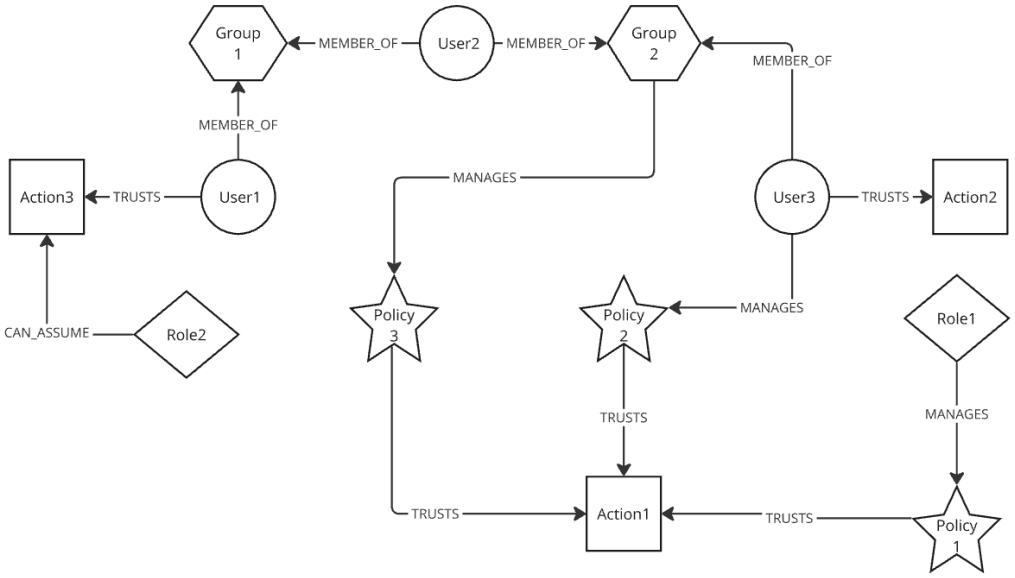
3.2.2 Manipulating Relationships

Despite the unidirectionality of the edges, Neo4j provides functionalities that allow for their manipulation and exploration. Specifically, the "relationship inversion" feature enables querying and interpreting relationships in the reverse direction, while "quantified relationships" serve as placeholders or conceptual links to indicate the absence of direct connections. [13] These features facilitate a more flexible and comprehensive analysis of relationships within the graph database.

3.3 Graph Visualization Example

The below figure showcases the full spectrum of node types and relationships mapped from IAM data to Neo4j. It includes nodes representing users, roles, groups, roles, policies and actions, all interconnected by various relationship types such as HAS_MEMBER, MANAGES, CAN_ASSUME and TRUSTS. This visualization highlights the complex structure of permissions, group memberships, and role assignments, providing a clear, intuitive understanding of the access control structure.

Figure 3.1: Graph Structure of IAM Data mapped into the Model



Chapter 4

IAM-Sentinel

The newly developed Go tool offers seamless integration of IAM data with Neo4j, enabling users to visualize and analyze their access controls in a graph database. By mapping IAM entities such as users, roles, and permissions directly to Neo4j, this tool simplifies the process of understanding complex relationships and dependencies within an organization's access management framework. This enhances security audits, compliance checks, and overall governance, providing a powerful way to identify potential vulnerabilities or inefficiencies in IAM configurations.

4.1 Architecture

The tool's architecture is built around several key components, each with a distinct role, adhering to the Single Responsibility Principle [17]. All components are included within the main package to simplify usage.

At its core is the Sentinel, the primary orchestrator that manages the flow of data through the system and the configuration parameters, embedding all other components.

The Neo4jConnector handles connections to the Neo4j database, ensuring secure and efficient data ingestion and retrieval.

The raw data is read from JSON files by the Parser component, which also processes and structures it, making it ready for mapping.

The ResourceMapper then takes over, translating IAM entities into the corresponding nodes and relationships for Neo4j.

The Analyzer provides tools for querying and analyzing the mapped data, allowing users to extract meaningful insights and perform detailed evaluations of their IAM configurations. This component only serves the purpose of demonstration, for more complex analysis, one should use neo4j directly.

Finally, the Exporter component facilitates the transfer of the analyzed data into files or console logs.

This modular architecture ensures flexibility, scalability, and ease of maintenance.

4.2 Data mapping

In the system, entities such as users, groups, roles, policies and actions are represented as nodes in the Neo4j graph database. Each node is assigned a specific label corresponding to its entity type and is populated with proprietary fields as detailed in section 3.1.1.

The mapping of relationships within the graph is more complex. The mapper extracts actions from policy documents, like the user policy list, and creates relationships of type TRUSTS between users and these actions. This process is similarly applied to groups, with relationships of type TRUSTS established between groups and actions. For policies, the mapper identifies all known versions listed in the version list and maps these policies to actions using the TRUSTS relationship. The same approach is used for roles, with relationships of type TRUSTS linking roles to actions.

Additionally, roles may include assume policy documents associated with the role or the instance profile list. In this context, the mapper utilizes CAN_ASSUME relationships to connect roles to actions defined in these assume policies.

Furthermore, all policies are extracted from entities that can contain them(users, groups, roles) and a MANAGES relationship is created between each entity and its associated policies.

Lastly, the mapper establishes relationships linking users to their respective groups, as specified in the GroupList, thereby reflecting group memberships in the graph database.

4.3 Integration

There are two primary integrations within the application: AWS for data extraction and Neo4j for data visualization and analysis.

The tool operates independently of the AWS SDK and does not establish any direct connection to AWS whatsoever. Instead, it processes IAM data provided in the form of JSON files, which are outputs from the relevant AWS command. This design choice enables versioning, allowing users to visualize and analyze historical IAM configurations and track their evolution over time.

The integration with Neo4j is accomplished through the use of the Neo4j Go driver [15], which utilizes session, read, write, and transaction functions to facilitate data interactions. Additionally, the application provides a Dockerfile that enables users to quickly deploy a local instance of Neo4j, facilitating rapid and convenient access for usage.

4.4 Use cases

The tool's mapping capabilities offer significant benefits across various use cases and applications. For example, in security audits, it enables detailed visualization of IAM

structures, helping to identify and address potential vulnerabilities, such as excessive permissions or misconfigurations; for compliance and governance, it provides clear, visual evidence of access controls and policies, facilitating easier audits and adherence to regulatory requirements.

4.5 Querying & Visualization

The Neo4j User Interface, accessible through the Docker image on port 7474, offers a comprehensive UI for visualizing data and can also be utilized to further enhance this visualization. [11] Within this interface, users can perform a variety of tasks to enhance the visualization of the data. These tasks include querying the graph to extract specific information, applying filters to narrow down the displayed data, and utilizing color coding to distinguish between different types of nodes or relationships. Additionally, users can adjust the size of nodes to emphasize particular entities or relationships, thereby improving the clarity of the visualization.

For demonstration purposes, simple example queries are provided below to illustrate the types of insights that can be obtained from the mapped IAM data. These examples serve to showcase the flexibility and effectiveness of the Neo4j UI in analyzing and understanding complex IAM configurations.

Figure 4.1: Query to get Groups with associated Users

```
1  MATCH (group:Group)-[:HAS_MEMBER]->(user:User)
2  RETURN user, group
```

Figure 4.2: Groups with associated Users

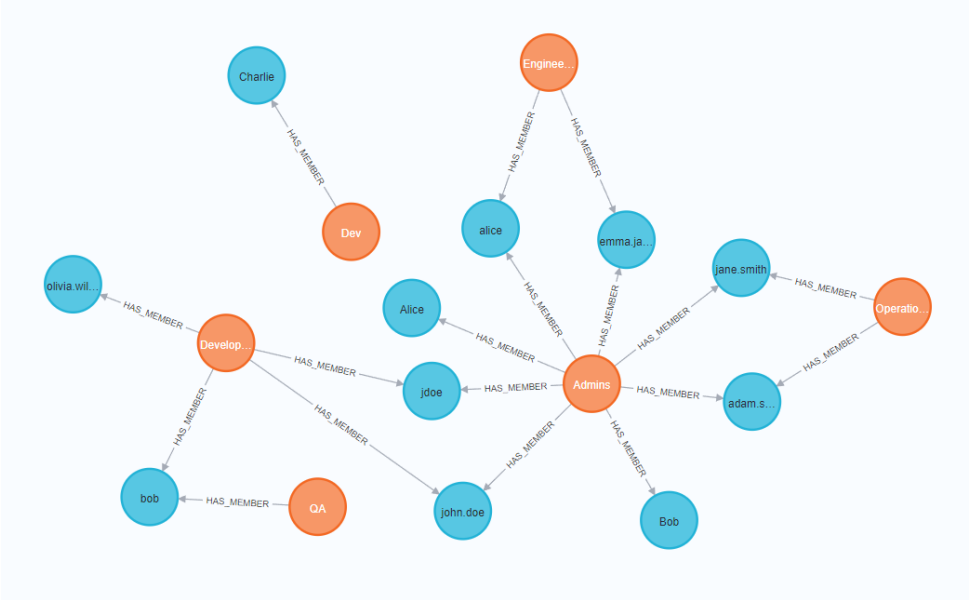


Figure 4.3: Query to get allowed actions for users

```
MATCH(u:User)-[r]-(a:Action) WHERE r.effect = "Allow"
RETURN u,a
```

Figure 4.4: Allowed actions for Users

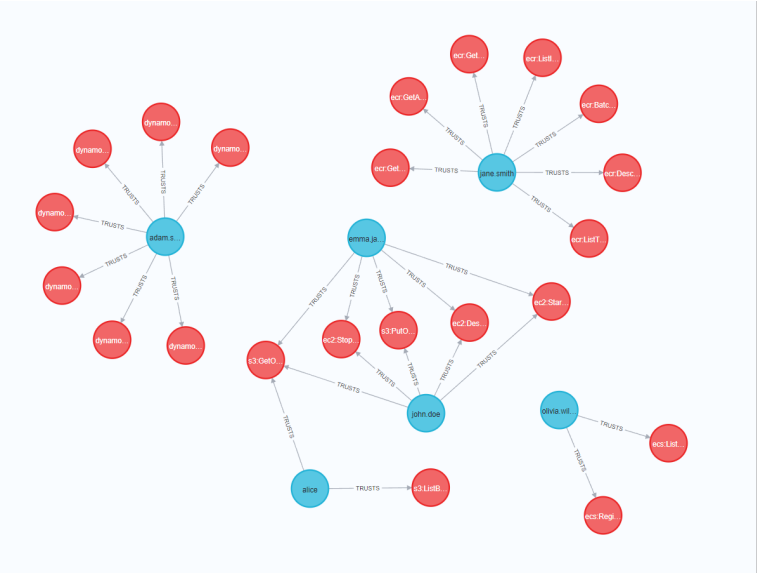


Figure 4.5: Query to fetch all users with any S3 permissions

```
MATCH(a:Action)<-[*..3]-(u:User) WHERE a.action =~ "ec2.*"
RETURN DISTINCT u.arn As ARNs
```

Figure 4.6: Users with permission to perform any operation on S3

	ARNs
1	"arn:aws:iam::123456789012:user/adam.smith"
2	"arn:aws:iam::123456789012:user/jane.smith"
3	"arn:aws:iam::123456789012:user/bob"
4	"arn:aws:iam::123456789012:user/emma.jackson"
5	"arn:aws:iam::123456789012:user/john.doe"

Chapter 5

Analysis & Results

5.1 Data set

As discussed in Chapter 4, the data is sourced from local files and conforms to the format of the *get-account-authorization-detail* output. Due to security concerns, actual data could not be used; therefore, the example data utilized in this study was generated by a large language model (LLM) and then subject to manual inspection and verification.

Given the strongly typed nature of the Go programming language, it was necessary to modify the data to ensure compatibility. This involved adjusting the data format to align with Go's strict type requirements, which demand that variables and data structures adhere to specific types.

These transformations were applied uniformly across all fields that could represent either a single value or an array. To ensure consistency, all such fields were converted into arrays, with single-value fields being transformed into arrays containing a single element. An example of this transformation is the *AttachedManagedPolicies* field. When only a single attached policy is present, the command's output, in the JSON format, represents this as a single object rather than an array of objects. To accommodate the strongly typed nature of Go, such fields are converted into arrays, even when they contain only one element. The tool could be extended to cover such cases, but this does not make the scope of this research.

5.2 Graph Projection

Given that the dataset used in this study is relatively small, there shouldn't be an issue with performance. Consequently, the *"fullGraph"* graph projection is actually the entire graph, including all nodes, relationships, and properties.

5.3 Centrality

Centrality measures in graphs are critical for understanding the importance of nodes within a network. From a security perspective, these measures help identify key nodes that could be crucial for maintaining or compromising the integrity of the system.

5.3.1 Degree Centrality

In this analysis, our focus is solely on nodes of type Group and User. To enhance clarity and optimize performance, I have created a projection that includes only these node types. This projection is named *groupsWithUsers* and was generated using the query provided below:

Figure 5.1: Query for creating groupsWithUsers projection

```
1  MATCH (g:Group)-[:HAS_MEMBER]->(u:User)
2  RETURN gds.graph.project('groupsWithUsers', g, u, {})
```

This projection encompasses all users and groups, establishing connections between them. By utilizing this projection, Degree centrality can be applied to identify the most influential and prominent groups, defined by the highest number of associated users. An example of such a query, along with its output, is provided below:

Figure 5.2: Query to calculate Degree Centrality for Groups

```
1  CALL gds.degree.stream('groupsWithUsers')
2  YIELD nodeId, score
3  RETURN gds.util.asNode(nodeId).GroupName AS groupName,
      score AS members
4  ORDER BY members DESC
```

The default orientation for the *gds.degree.stream* function is set to NATURAL, meaning it evaluates the out-degree of the nodes. However, the query can be reversed to consider the in-degree, allowing to determine which users belong to the most groups.

The results of the query can be visualized in the next figure 5.3.1. This analysis can help in identifying which groups are the most popular, and thus helping with audits and with improving the structure of the AWS IAM.

Figure 5.3: Degree Centrality for Groups

	groupName	members
1	"Admins"	8.0
2	"Developers"	4.0
3	"Engineering"	2.0
4	"Operations"	2.0
5	"Dev"	1.0
6	"QA"	1.0

5.3.2 Betweenness Centrality

An example of identifying the most significant Policy in the graph can be achieved using betweenness centrality. As presented in 2.7.2, this metric measures the influence of a node on the flow of information within the graph by calculating the shortest paths between all pairs of nodes.

The query below analyzes the entire graph and returns the Policy node with the highest betweenness centrality. This Policy is identified as the most critical and should be prioritized for protection against potential attacks.

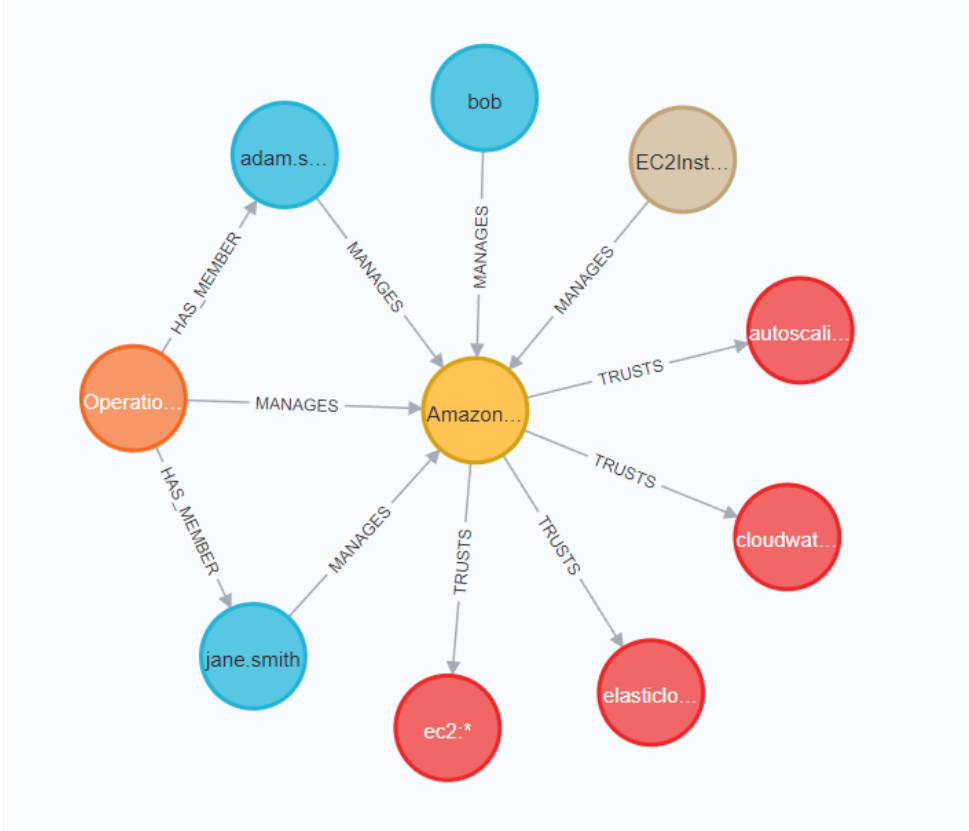
Figure 5.4: Query to calculate Betweenness Centrality for Policies

```
CALL gds.betweenness.stream('fullGraph')
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
WHERE 'Policy' in labels(node)
RETURN node.PolicyName, score
ORDER BY score DESC
LIMIT 1
```

Figure 5.5: Betweenness Centrality for Policies

	node.PolicyName	score
1	"AmazonEC2FullAccess"	32.0

Figure 5.6: AmazonEC2FullAccess Policy Neighborhood



5.4 PageRank

PageRank measures the relative importance of a node in a graph by evaluating the likelihood of a random walker visiting that node. Nodes with high PageRank scores are considered more important or influential because they are likely to be visited frequently, either due to having many incoming links or being linked to by other important nodes.

The query below analyzes the graph and will return Policy nodes that are the most important for the flow of permissions in the graph. This metric can help identify policies that, due to their connections, are likely to be crucial in maintaining the integrity and functionality of the IAM graph.

Figure 5.7: Query to calculate PageRank for Policies

```
CALL gds.pageRank.stream("fullGraph", {})
YIELD nodeId, score
WITH gds.util.asNode(nodeId) as node, score
WHERE 'Policy' in labels(node)
RETURN node.PolicyName as policyName, score
ORDER BY score DESC
```

Figure 5.8: PageRank for Policies

	policyName	score
1	"AmazonEC2FullAccess"	0.4815667467948718
2	"AdministratorAccess"	0.4302154342185593
3	"AmazonS3FullAccess"	0.28060576923076924
4	"AmazonS3ReadOnlyAccess"	0.22339786324786326
5	"AWSLambdaBasicExecutionRole"	0.21501682692307697

5.4.1 Closeness Centrality

Closeness centrality measures a node's average "farness", or inverse distance, to all other nodes in the graph. Nodes with a low closeness score have the longest distances to all other nodes, which can help identify actions or nodes that are the most challenging to "reach".

An application of this concept involves identifying the actions that are the most challenging to obtain or access within an environment. Specifically, this query aims to return actions that are considered most critical for maintaining and managing the security and functionality of the environment. For instance, it should include actions with broad or powerful permissions, such as *iam:**.

If actions that give the most critical permissions are not found in such an analysis, they should be further analyzed and IAM maintainers should improve the granting of such actions and also reduce the paths that they can be obtained through.

Figure 5.9: Query to calculate Closeness Centrality for Policies

```
CALL gds.closeness.stream('fullGraph')
YIELD nodeId, score
WITH gds.util.asNode(nodeId) AS node, score
WHERE 'Action' in labels(node)
RETURN node.action as Action, score
ORDER BY score ASC
```

Figure 5.10: Closeness Centrality for Actions

	Action	score
1	"ec2:*"	0.45
2	"elasticloadbalancing:*"	0.45
3	"cloudwatch:*"	0.45
4	"autoscaling:*"	0.45
5	"ecr:DescribeRepositories"	0.6
6	"ecr:GetAuthorizationToken"	0.6
7	"ecr:GetDownloadUrlForLayer"	0.6
8	"ecr:GetRepositoryPolicy"	0.6
9	"ecr:ListImages"	0.6
10	"ecr:ListTagsForResource"	0.6
11	"ecr:BatchGetImage"	0.6
12	"iam:*"	0.6666666666666666
13	"s3:PutObject"	0.6666666666666666
14	"ec2:StartInstances"	0.6666666666666666
15	"ec2:StopInstances"	0.6666666666666666

5.5 Community detection

In GDS, certain algorithms, such as Triangle Count, require undirected relationships, while others, like Louvain, perform better with an undirected graph. To accommodate this, we can generate a new projection of the entire graph in which all relationships are treated as undirected. The new projection is called *undirectedFullGraph* and it was created with the following query:

Figure 5.11: Query to create undirected full graph projection

```
CALL gds.graph.project(
   'undirectedFullGraph',
   ['*'],
   {
     HAS_MEMBER: {
       type: 'HAS_MEMBER',
       orientation: 'UNDIRECTED'
     },
     MANAGES: {
       type: 'MANAGES',
       orientation: 'UNDIRECTED'
     },
     CAN_ASSUME: {
       type: 'CAN_ASSUME',
       orientation: 'UNDIRECTED'
     },
     TRUSTS: {
       type: 'TRUSTS',
       orientation: 'UNDIRECTED'
     }
   }
 )
```

5.5.1 Louvain

By identifying clusters of tightly connected nodes, the Louvain method can reveal groups of IAM roles, users, and permissions that are closely related or frequently interacting. This community detection can help in understanding and managing access control more effectively, uncovering potential security risks, and optimizing permissions. For example, if certain roles or users frequently operate together, it might indicate a need for specialized access controls or highlight areas where permissions could be adjusted for better efficiency.

The Louvain method was applied to both the *fullGraph* and the *undirectedFullGraph* to illustrate the influence of graph directionality on the community detection process. This comparative analysis demonstrates how the presence of directional edges can affect the outcomes of community detection.

The following query was used for both executions, with the respective projection name used, returning the ARNs for all nodes, except Actions, where the action was used instead.

Figure 5.12: Query for the Louvain Method

```
CALL gds.louvain.stream("fullGraph", {})
YIELD nodeId, communityId
RETURN communityId as communityId, count(distinct nodeId)
   as count, collect(coalesce(gds.util.asNode(nodeId).arn,
   gds.util.asNode(nodeId).action)) as ARNs
```

Figure 5.13: Louvain method for fullGraph

	communityId	count	ARNs
1	53	29	["arn:aws:iam::aws:policy/AdministratorAccess", "arn:aws:iam::123456789012:group/Admins", "arn:aws:iam::123456789012:user/Alice", "a
2	11	3	["arn:aws:iam::aws:policy/PowerUserAccess", "arn:aws:iam::123456789012:group/Dev", "arn:aws:iam::123456789012:user/Charlie"]
3	8	6	["policygen-201310141157", "arn:aws:iam::123456789012:group/Finance", "aws-portal:*", "DenyBillingAndIAMPolicy", "arn:aws:iam::12345
4	85	10	["arn:aws:iam::aws:policy/AmazonS3FullAccess", "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess", "arn:aws:iam::123456789012:r
5	59	12	["arn:aws:iam::123456789012:policy/create-update-delete-set-managed-policies", "iam:CreatePolicy", "iam:CreatePolicyVersion", "iam:Dele
6	28	3	["arn:aws:iam::123456789012:policy/S3-read-only-specific-bucket", "s3:Get*", "s3:List*"]
7	79	20	["arn:aws:iam::aws:policy/AmazonEC2FullAccess", "ec2:*", "elasticloadbalancing:*", "cloudwatch:*", "autoscaling:*", "arn:aws:iam::1234567
8	90	13	["arn:aws:iam::aws:policy/AmazonDynamoDBReadOnlyAccess", "QAPolicy", "arn:aws:iam::123456789012:group/QA", "dynamodb:Scan", "
9	55	1	["logs:CreateLogGroup"]
10	56	1	["logs:CreateLogStream"]
11	57	1	["logs:PutLogEvents"]
12	86	1	["DeveloperECSAccessPolicy"]
13	97	1	["ECSClusterPolicy"]

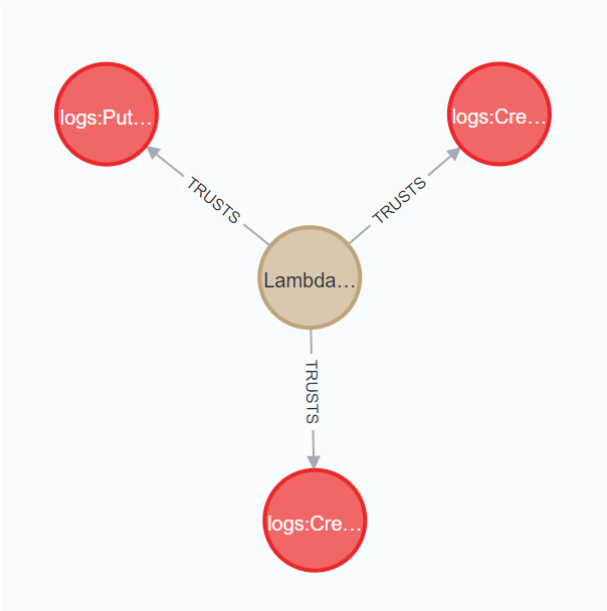
The figures display noticeable differences due to the relationships differences. The execution on the directed graph struggles with grouping actions that should belong together, particularly evident in the *logs* actions, where CreateLogGroup, CreateLogStream, and PutLogEvents are all granted by the same Role LambdaExecutionRole.

The execution on the undirected graph successfully assigns the actions to the same community, which enhances the accuracy and coherence of the analysis. This approach ensures that related actions are grouped together, which enhances the overall comprehension of the system's structure. It can also provide analysts with insights into sensitive points within the network and identify potential security breaches, particularly if nodes that normally should be separated are found within the same communities.

Figure 5.14: Louvain method for undirectedFullGraph

	communityId	count	ARNs
1	40	23	["arn:aws:iam::aws:policy/AdministratorAccess", "arn:aws:iam::123456789012:group/Admins", "arn:aws:iam::123456789012:user/Alice", "arn:a
2	3	3	["arn:aws:iam::aws:policy/PowerUserAccess", "arn:aws:iam::123456789012:group/Dev", "arn:aws:iam::123456789012:user/Charlie"]
3	9	6	["policygen-201310141157", "arn:aws:iam::123456789012:group/Finance", "aws-portal:*", "DenyBillingAndIAMPolicy", "arn:aws:iam::1234567
4	54	17	["arn:aws:iam::aws:policy/AmazonS3FullAccess", "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess", "arn:aws:iam::123456789012:role
5	17	12	["arn:aws:iam::123456789012:policy/create-update-delete-set-managed-policies", "iam:CreatePolicy", "iam:CreatePolicyVersion", "iam:DeleteF
6	27	3	["arn:aws:iam::123456789012:policy/S3-read-only-specific-bucket", "s3:Get*", "s3:List*"]
7	72	15	["arn:aws:iam::aws:policy/AmazonEC2FullAccess", "ec2:*", "elasticloadbalancing:*", "cloudwatch:*", "autoscaling:*", "arn:aws:iam::1234567890
8	100	8	["arn:aws:iam::123456789012:group/Developers", "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole", "DeveloperECSAc
9	99	14	["arn:aws:iam::aws:policy/AmazonDynamoDBReadOnlyAccess", "QAPolicy", "arn:aws:iam::123456789012:group/QA", "dynamodb:Scan", "dyr

Figure 5.15: Logs actions that are part of LambdaExecutionRole



5.5.2 Label Propagation

Similarly, the Label Propagation algorithm is capable of identifying communities within the AWS IAM graph, offering valuable insights into the relationships and interactions between roles, users, and permissions. Since label propagation is more effective in undirected graphs, the following example utilizes the undirected projection for the analysis.

Figure 5.16: Query for the Label Propagation

```
CALL gds.labelPropagation.stream("undirectedFullGraph", {})
YIELD nodeId, communityId
RETURN communityId as communityId, count(distinct nodeId)
   as count, collect(coalesce(gds.util.asNode(nodeId).arn,
   gds.util.asNode(nodeId).action)) as ARNs
```

Figure 5.17: Label propagation for undirectedFullGraph

	communityId	count	ARNs
1	38	67	["arn:aws:iam::aws:policy/AdministratorAccess", "arn:aws:iam::123456789012:group/Admins", "policygen-201310141157", "arn:aws:iam::12345678
2	40	3	["arn:aws:iam::aws:policy/PowerUserAccess", "arn:aws:iam::123456789012:group/Dev", "arn:aws:iam::123456789012:user/Charlie"]
3	83	12	["arn:aws:iam::123456789012:policy/create-update-delete-set-managed-policies", "iam:CreatePolicy", "iam:CreatePolicyVersion", "iam:DeletePolicy
4	93	3	["arn:aws:iam::123456789012:policy/S3-read-only-specific-bucket", "s3:Get*", "s3:List*"]
5	96	6	["arn:aws:iam::aws:policy/AmazonEC2FullAccess", "ec2:*", "elasticloadbalancing:*", "cloudwatch:*", "autoscaling:*", "arn:aws:iam::123456789012:r
6	51	5	["LambdaBasicExecution", "arn:aws:iam::123456789012:role/service-role/LambdaExecutionRole", "logs:CreateLogGroup", "logs:CreateLogStream
7	120	5	["ecs:RegisterTaskDefinition", "ecs:ListClusters", "ECSClusterPolicy", "arn:aws:iam::123456789012:user/olivia.wilson", "arn:aws:iam::12345678901

The results of this analysis are similar to the Louvain ones (refer to 5.5.1), the difference here being that the label propagation method will not create clear distinctions between communities because it's a non-deterministic algorithm and it relies on local information to assign labels to nodes, unlike the Louvain method, which optimizes modularity to partition the graph into communities that maximize internal connectivity and minimize external connections. This approach may result in less distinct community boundaries, as nodes are influenced by their local neighborhood, potentially leading to overlapping or ambiguous community assignments. Although it provides a similar analytical perspective to the Louvain method, it does so with reduced accuracy.

5.5.3 Weakly Connected Components

WCC is often used early in an analysis to understand the structure of a graph. Using WCC to understand the graph structure enables running other algorithms independently on an identified cluster, but WCC can also be useful for understanding how permissions and roles might be linked across the organization, even if the connections are indirect. It helps in identifying potential vulnerabilities or pathways for privilege escalation.

Figure 5.18: Query for Weakly Connected Components

```
CALL gds.wcc.stream("fullGraph", {})
YIELD nodeId, componentId
RETURN gds.util.asNode(nodeId).arn as node, componentId
```

Figure 5.19: Weakly Connected Components for fullGraph

	componentId	count	ARNs
1	0	83	["arn:aws:iam::aws:policy/AdministratorAccess", "arn:aws:iam::123456789012:group/Admins", "policygen-201310141157", "arn:aws:iam::12345678901
2	2	3	["arn:aws:iam::aws:policy/PowerUserAccess", "arn:aws:iam::123456789012:group/Dev", "arn:aws:iam::123456789012:user/Charlie"]
3	16	12	["arn:aws:iam::123456789012:policy/create-update-delete-set-managed-policies", "iam:CreatePolicy", "iam:CreatePolicyVersion", "iam:DeletePolicy", "
4	26	3	["arn:aws:iam::123456789012:policy/S3-read-only-specific-bucket", "s3:Get*", "s3:List*"]

From this analysis, by inspecting the component with ID 2, it can be confidently concluded that only the user Charlie is part of the Dev group and possesses PowerUserAccess, no other user is able to attain those permissions. Moreover, this analysis is instrumental in identifying critical actions and permissions that should be tightly controlled and not easily accessible.

5.5.4 Triangle Count and Clustering Coefficient

Triangle Count and Clustering Coefficient are critical metrics in an AWS IAM graph because they provide valuable insights into the ”interconnectness” of roles, users, and permissions. A high triangle count in an IAM graph can reveal tightly connected groups where roles or users frequently collaborate, which might suggest areas of high trust or interdependency. Similarly, a high clustering coefficient can highlight dense clusters of permissions or roles that are closely associated, which can be crucial for detecting security risks, understanding access patterns, and optimizing permission management. Together, these metrics help in identifying critical relationships and potential vulnerabilities within your cloud infrastructure.

Figure 5.20: Query for Triangle Count

```
CALL gds.triangleCount.stream("undirectedFullGraph", {})
YIELD nodeId, triangleCount
RETURN coalesce(gds.util.asNode(nodeId).arn, gds.util.
    asNode(nodeId).action) as ARN, triangleCount as
    triangleCount
ORDER BY triangleCount DESC
```

Figure 5.21: Triangle Count for undirectedFullGraph

	ARN	triangleCount
1	"arn:aws:iam::aws:policy/AdministratorAccess"	8
2	"arn:aws:iam::123456789012:group/Admins"	6
3	"arn:aws:iam::123456789012:group/Developers"	5
4	"arn:aws:iam::123456789012:group/Engineering"	4
5	"arn:aws:iam::123456789012:group/Operations"	4
6	"arn:aws:iam::123456789012:user/alice"	3
7	"s3:GetObject"	3
8	"arn:aws:iam::123456789012:user/john.doe"	3
9	"arn:aws:iam::123456789012:user/jane.smith"	3
10	"arn:aws:iam::123456789012:user/emma.jackson"	3

The Triangle Count is highly effective in identifying redundant permissions, aiding in their optimization. A good example is the *s3:GetObject* action. As illustrated in the figure below, Alice and Emma belong to the Engineering Group, while John is part of the Developers group. All three users are granted the GetObject action, but their respective groups also have this action assigned, leading to redundant permissions.

Figure 5.22: Neighbourhood of Action s3:GetObject

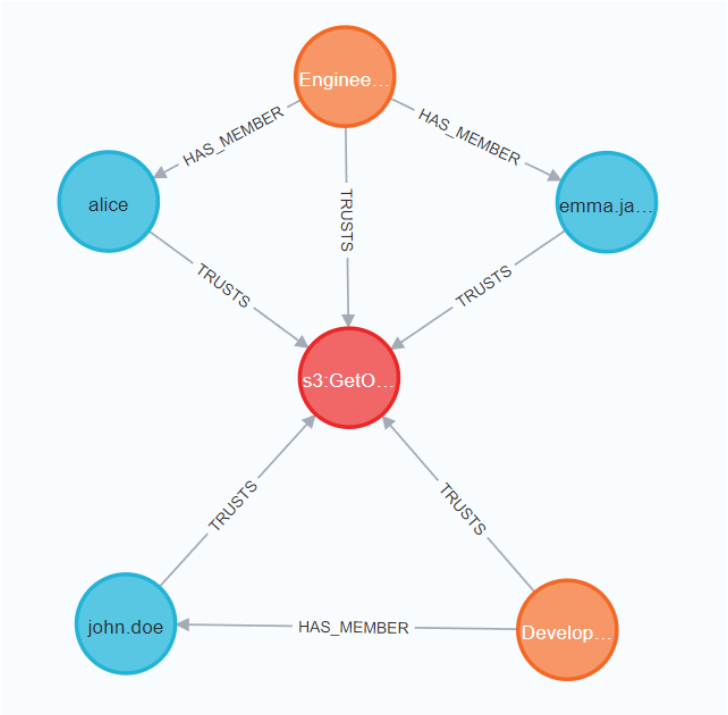


Figure 5.23: Query for Local Clustering Coefficient

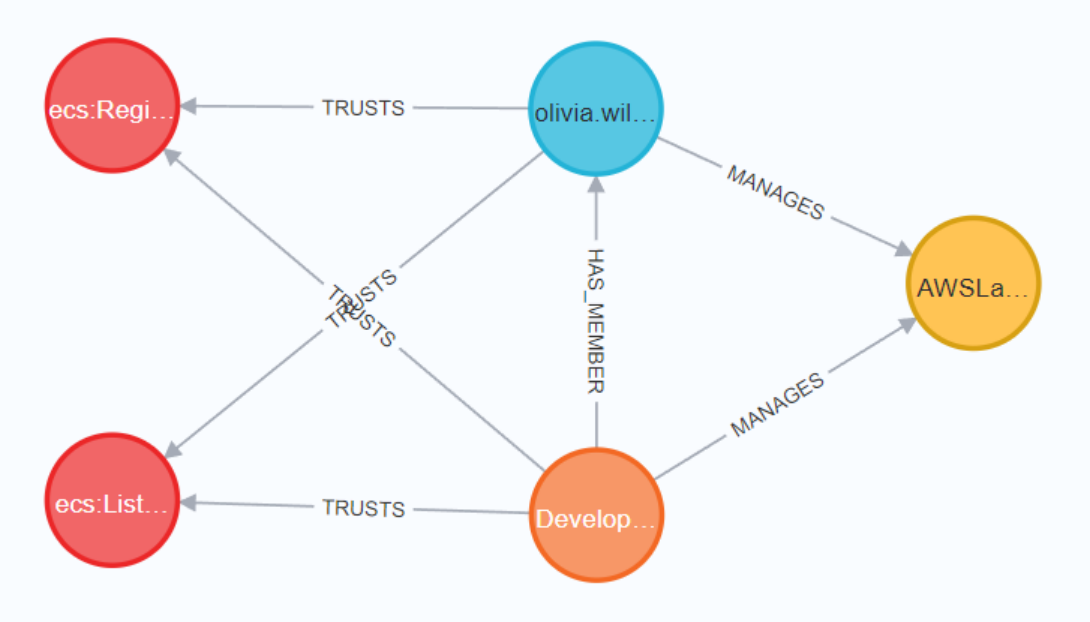
```
CALL gds.localClusteringCoefficient.stream("
   undirectedFullGraph", {})
YIELD nodeId, localClusteringCoefficient
RETURN coalesce(gds.util.asNode(nodeId).arn, gds.util.
   asNode(nodeId).action) as ARN,
   localClusteringCoefficient as localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Figure 5.24: Local Clustering Coefficient for undirectedFullGraph

	ARN	localClusteringCoefficient
1	"arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"	0.3333333333333333
2	"ecs:RegisterTaskDefinition"	0.3333333333333333
3	"ecs:ListClusters"	0.3333333333333333
4	"arn:aws:iam::123456789012:user/olivia.wilson"	0.3
5	"s3:GetObject"	0.2
6	"arn:aws:iam::aws:policy/AdministratorAccess"	0.17777777777777778
7	"arn:aws:iam::123456789012:group/Admins"	0.16666666666666666
8	"arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess"	0.16666666666666666
9	"arn:aws:iam::123456789012:user/jdoe"	0.16666666666666666
10	"s3:ListBucket"	0.16666666666666666

Similar to the Triangle Count, the local clustering coefficient helps identify nodes with potentially redundant relationships due to their neighboring connections. In the example above, the user olivia.wilson has a clustering coefficient of 0.3. Upon closer inspection, we can safely assume that this user has redundant permissions for both EC2 actions and redundant management over the AWSLambda policy, due to her membership in the Delevopers group.

Figure 5.25: Neighbourhood of User olivia.wilson



5.6 Results

As demonstrated in this chapter, the proposed model and associated tool offer significant support in detecting anomalies and misconfigurations. The model facilitates the application of multiple graph analysis algorithms on the subsequent graph, with each algorithm yielding relevant results.

The proposed model and its implementation can be compared to other existing models, implementations, and research papers in the field. For instance:

- *Sophos Cloud Optix Anomaly Detection Service*: This service utilizes AI models to detect anomalous patterns, requiring specific resources and learning periods to establish a baseline of normal behavior, which then enables the identification of unusual activity. Thus, the methodology and underlying principles it employs differ significantly from those of our approach. Additionally, this service is not free and requires a paid subscription.

- AWS tools: AWS provides tools for anomaly detection, primarily based on CloudWatch and log analysis. However, these tools are limited to analyzing anomalies within the current environment and detecting changes over time. In contrast, our proposed model can load and analyze any AWS configuration, regardless of its time of ”existance”.

- *Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies* [8]: The paper is an excellent study but focuses solely on policies, actions,

and their direct connections. It does not account for the propagation of permissions through roles and groups.

In summary, although the presented model might achieve lower accuracy in certain specific areas compared to other methods, it compensates for this by providing more extensive coverage across a wide range of scenarios. This broader scope enables it to address a greater variety of use cases and potential issues, thereby offering a more comprehensive analytical tool overall.

Chapter 6

Conclusions

In conclusion, the research presented highlights the significant potential of graph analytics in enhancing the security and operational efficiency of AWS IAM environments. By enabling a comprehensive visualization of access patterns and relationships within an organization's infrastructure, the proposed graph model and the accompanying, *IAM-Sentinel*, tool provide a robust framework for detecting anomalies, optimizing access policies, and improving compliance processes. The integration of IAM data into a graph database like Neo4j allows for sophisticated analysis, uncovering insights that traditional methods might overlook. This approach not only strengthens security by identifying excessive permissions and unusual access patterns but also streamlines IAM management by resolving redundant or conflicting privileges. The practical application of this model and tool, as demonstrated through various analytics, highlights its value in cultivating a more secure and efficient IAM environment.

6.1 Further developments

While the model and the accompanying tool effectively facilitate mapping and analysis, they have limitations. This subsection highlights several shortcomings of the proposed model and suggests potential areas for improvement:

- Unmarshalling Intervention: As outlined in Section 5.1, the strongly typed nature of the Go programming language necessitates data modifications for compatibility. This issue could be addressed by utilizing the any type in Go and performing type casting during the parsing stage.
- Transitive Edges: Incorporating intermediary edges, which represent implicit connections inferred through existing paths, could enhance various aspects of graph analysis. These transitive edges simplify direct path analysis, reachability queries, and contribute to overall analytical efficiency. They can be utilized for different types of graph projections depending on the specific analysis.

- Weighted Edges: The inclusion of weights in the graph can profoundly affect the outcomes of graph analysis by influencing algorithmic performance and the interpretation of relationships. Weights could be incorporated into the model by considering the length of ”hops” created by transitive edges or by using node properties to derive alternative weighting methods.

- Relationship Properties: Although the model captures the effects of actions on relationships, these properties are currently not utilized in the analysis. Leveraging these properties could potentially enhance the performance and depth of the analysis.

- Time series: Treating the graph as a time series involves capturing and storing snapshots of the IAM configuration at different points in time. This can be achieved by regularly mapping IAM data and indexing each snapshot with a timestamp. This approach could enable historical analysis, allowing for the identification of trends, assessment of policy changes, and detection of anomalies or unauthorized access.

Bibliography

[1] Mohammad Al Hasan and Vachik S Dave. "Triangle counting in large networks: a review." In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.2 (2018), e1226.

[2] Amazon. *Official AWS IAM Documentation*. URL: https://docs.aws.amazon.com/iam/.

[3] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based automated reasoning for AWS access policies using SMT." In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–9.

[4] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. "Fast unfolding of communities in large networks." In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[5] Ulrik Brandes and Christian Pich. "Centrality estimation in large networks." In: *International Journal of Bifurcation and Chaos* 17.07 (2007), pp. 2303–2318.

[6] Sergey Brin and Lawrence Page. "The anatomy of a large-scale hypertextual web search engine." In: *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117.

[7] Inc Docker. "Docker." In: *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker* (2020).

[8] Thijs van Ede, Niek Khasuntsev, Bas Steen, and Andrea Continella. "Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies." In: *Proceedings of the 2022 on Cloud Computing Security Workshop*. 2022, pp. 63–74.

[9] Linton C Freeman et al. "Centrality in social networks: Conceptual clarification." In: *Social network: critical concepts in sociology. Londres: Routledge* 1 (2002), pp. 238–263.

[10] José Guia, Valéria Gonçalves Soares, and Jorge Bernardino. "Graph Databases: Neo4j Analysis." In: *ICEIS (1)*. 2017, pp. 351–356.

[11] Sumit Gupta. *Neo4j Essentials*. Packt Publishing, 2015.

[12] Mike McGrath. *GO Programming in easy steps: Discover Google's Go language (golang)*. In Easy Steps Limited, 2020.

[13] Mark Needham and Amy E Hodler. "A comprehensive guide to graph algorithms in neo4j." In: *Neo4j. com* (2018).

[14] Neo4j. *Betweenness Centrality in Neo4j GDS*. URL: https://neo4j.com/docs/graph-data-science/current/algorithms/closeness-centrality/.

[15] Neo4j. *Official Neo4j Go Driver*. URL: https://pkg.go.dev/github.com/neo4j/neo4j-go-driver/.

[16] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. "Near linear time algorithm to detect community structures in large-scale networks." In: *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 76.3 (2007), p. 036106.

[17] Jyotiswarup Raiturkar. *Hands-On Software Architecture with Golang: Design and architect highly scalable and robust applications using Go*. Packt Publishing Ltd, 2018.

[18] Satu Elisa Schaeffer. "Graph clustering." In: *Computer science review* 1.1 (2007), pp. 27–64.

[19] Estelle Scifo. *Graph Data Science with Neo4j: Learn how to use Neo4j 5 with Graph Data Science library 2.0 and its Python driver for your project*. Packt Publishing Ltd, 2023.

[20] Andreas Wittig and Michael Wittig. *Amazon Web Services in Action: An in-depth guide to AWS*. Simon and Schuster, 2023.