Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies

Thijs van Ede
University of Twente
The Netherlands, Enschede
t.s.vanede@utwente.nl

Niek Khasuntsev
University of Twente
The Netherlands, Enschede
n.a.khasuntsev@alumnus.utwente.nl

Bas Steen
PwC Advisory N.V.
The Netherlands, Amsterdam
bas.steen@pwc.com

Andrea Continella
University of Twente
The Netherlands, Enschede
a.continella@utwente.nl

ABSTRACT

In recent years, misconfigurations of cloud services have led to major security incidents and large-scale data breaches. Due to the dynamic and complex nature of cloud environments, misconfigured (e.g., overly permissive) access policies can be easily introduced and often go undetected for a long period of time. Therefore, it is critical to identify any potential misconfigurations before they can be abused. In this paper, we present a novel misconfiguration detection approach for identity and access management policies in AWS. We base our approach on the observation that policies can be modeled as permissions between entities and objects in the form of a graph. Our key idea is that misconfigurations can be effectively detected as anomalies in such a graph representation. We evaluate our approach on real-world identity and access management policy data from three enterprise cloud environments. We investigate the effectiveness of our approach to detect misconfigurations, showing that it has a slightly lower precision compared to rule-based systems, but it is able to correctly detect between 3.7 and 6.4 times as many misconfigurations.

CCS CONCEPTS

• **Security and privacy → Access control**; **Intrusion/anomaly detection and malware mitigation**; • **Networks → Cloud computing**.

KEYWORDS

Identity and access management, Anomaly detection, Cloud computing

ACM Reference Format:
Thijs van Ede, Niek Khasuntsev, Bas Steen, and Andrea Continella. 2022. Detecting Anomalous Misconfigurations in AWS Identity and Access Management Policies. In *Proceedings of the 2022 Cloud Computing Security Workshop (CCSW '22), November 7, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3560810.3564264

CCSW '22, November 7, 2022, Los Angeles, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9875-6/22/11.
https://doi.org/10.1145/3560810.3564264

1 INTRODUCTION

Data breaches are a dangerous threat to our society [34]. In 2019, Capital One, an American bank holding, suffered a data breach where data of over a hundred of million people was stolen [18]. More recently, the credit card details of more than hundred million hotel guests were stolen [32], personal data of over 10 million church-goers was leaked [31], and inmate records were leaked from a prison system [2]. All these data breaches have two common characteristics: the data was hosted in the cloud; and the cloud access policies were misconfigured.

Since its introduction in 2006, the use of cloud computing solutions, such as Amazon Simple Storage Service (S3), Microsoft Azure, and Google Cloud Platform increased. While the adoption of cloud services offers a wide range of benefits, it also brings a number of security challenges [11]. As the aforementioned data breaches already demonstrate, security misconfigurations, such as publicly accessible private data, are a considerable threat in cloud security. Attackers even collect publically available lists of misconfigured to find common vulnerabilities in cloud systems [26]. Moreover, the majority of misconfigurations are reported only when they lead to security incidents [16, 23, 24], making the problem even larger than it first appears. As most cloud environments are exposed to the Internet, misconfigurations create a large attack surface, making it easy for attackers to scan for misconfigured services and exploit them.

To prevent breaches, cloud solutions offer identity and access management (IAM). IAM allows cloud operators to define and manage the roles and access privileges of network users and systems, offloading responsibility to cloud administrators [9, 12]. When configured correctly, IAM systems prevent unauthorized access to protected resources, ensuring that only specified users get access to the specified resources. However, for each newly introduced or modified resource, role or user, these IAM policies must be reconfigured. This (re)configuration is challenging and may unintentionally introduce incorrect rules, which allow access to resources that are undesirable (e.g., guest users accessing sensitive data), potentially leading to security issues. We refer to these undesirable rules as *misconfigurations* and further explain them in Section 2. As cloud environments are often large, dynamic, and complex, the configuration of security services becomes difficult and error prone. Therefore, there is a need for systems to flag potential misconfigurations *as soon as* they are introduced and *before* such misconfigurations can be abused.

Existing solutions such as Cloud Custodian [19] use a rule-based approach to prevent the introduction of security misconfigurations in cloud environments. Before deployment, such systems compare IAM policies against existing rules to detect misconfigurations. However, rule-based approaches are limited by the fact that rules need to be created and maintained to adhere to security policies for each specific organization. A rule-based system requires constant effort to keep rules up-to-date, has different requirements per organization, and can be error prone due to manual rule creation. Other approaches, such as P-DIFF [35], monitor access and control behavior to detect misconfigurations. However, this is a reactive approach, meaning that misconfigurations are detected only when they are abused. Unfortunately, at this point, it is often too late, as data is already leaked.

To overcome the limitations of existing solutions, we propose a novel misconfiguration detection approach that is *proactive* in detection, but is able to take into account the *context* of a specific organization and does not require *high maintenance* like rule-based detection. We achieve this by collecting all identity and access management policies from cloud environments before they are rolled out. Next, we model identity and access management policies as a graph where rules are represented by edges between nodes that model entities (users, groups, and roles) and resources. In such graphs, we observed that similar policies within the same environment also have similar graph representations. This means that, correctly configured policies are similar to each other, while misconfigurations show up as anomalies. Leveraging our observation, we apply anomaly detection to spot outliers and raise real-time alerts about potential misconfigurations.

To validate our proposed approach, we collected real-world identity and access management policy data of AWS cloud environments from three different companies. We manually labelled the data as correct policies and potential misconfigurations. On these datasets, our proposed approach correctly detected between 3.7 and 6.4 times as many misconfigurations as rule-based approaches.

In summary, our paper makes the following contributions:

- We introduce a novel approach to model AWS identity and access management policies in the form of a graph model;
- We present a novel system that uses anomaly detection techniques to identify potential misconfigurations in Amazon AWS environments;
- We show that our approach correctly detects between 3.7 and 6.4 times as many misconfigurations than state-of-the-art rule-based approaches on a real-world dataset of IAM policies from three AWS cloud environments.

To foster future research on the automatic identification of misconfigurations in cloud environments, we release our prototype open-source: https://github.com/utwente-scs/misdet-code.

2 BACKGROUND AND MOTIVATION

We first detail the main concepts used in IAM and then focus on better understanding when access to cloud resources is misconfigured. While we focus on AWS, the concepts discussed here can be applied to any IAM system.

Cloud Identity and Access Management. Identity and access management allows administrators to limit access to their cloud services and resources. Using IAM, the administrator can create and manage different *entities* that can consist of individual *users*, *groups* of multiple users, and *roles* that can be linked users, groups or even other systems. Each entity has certain *permissions* that allow or deny certain *actions* on cloud resources, such as reading or writing. Sets of permission rules are captured in *policies* that allow administrators a more high level view of related permissions. Cloud providers have different ways of implementing IAM policies, Figure 1 gives an example of an AWS' IAM policy [6] in JSON format, which describes an AdministratorAccess policy that allows an administrator to perform any action (*, wildcard) on any resource (*). In this example, the policy itself is related to a *resource*, which means that it still needs to be attached to an entity, i.e. either a user, group, or role. The entity is then granted the permissions specified in the policy.

Misconfigurations. While cloud providers often include IAM systems, it is the responsibility of the customer to specify IAM policies for their organization [12]. As organizations differ in their needs, so do the policies they need in place for access and restriction to cloud resources. Keeping a balance between access for maintaining company workflow, and restrictions to improve security is difficult, especially for large organizations where policies often change depending on new needs. This can introduce misconfigurations into IAM policies in several ways. We identify three types of misconfigurations:

1. **Overly permissive policies** allow entities actions on resources that they should not be able to perform.
2. **Overly restrictive policies** denies entities actions on resources that they should be able to perform.
3. **Incorrectly attached policies** specifies correct actions on resources, but for the incorrect entity.

Misconfigured policies open up vulnerabilities that can potentially be exploited by attackers. While there exist some approaches to detect misconfigurations [19, 35], they are either (1) rule-based, requiring a large effort to maintain rules, or (2) reactive, thus detecting misconfigurations only after they are abused. Our approach tackles the following challenges: (i) **Proactive**, misconfigurations need be detected *before* they lead to security incidents; (ii) **Context-specific**, misconfigurations depend on the organization for which they are detected. Some organizations may require stricter or more permissive policies, depending on their workflow; (iii) **Low maintenance**, administrators should not have to maintain rules or perform manual fine-tuning for detecting misconfigurations.

```
{
   "Version":"2012-10-17",
   "name":"AdministratorAccess",
   "Statement":{
      "Effect":"Allow",
      "Action":"*",
      "Resource":"*"
   }
}
```

Figure 1: AdministratorAccess policy that allows all actions (*) on all resources (*).

3 APPROACH

We aim to detect potential misconfigurations in IAM policies based on the cloud environment for which they are defined. We recall that IAM policies define allowed/disallowed actions on resources and link them as permissions to entities. We propose to model these connections in a graph to represent the connected nature of policies, providing advantages for analyzing and visualizing the policies.

By modelling IAM policies as a graph, we observe that policies permissiveness level is naturally represented through the number of connections (i.e., degree) of each node. Intuitively, an overly permissive policy will have many direct connections to allowed actions, whereas overly restrictive policies will have many direct connections to disallowed actions. We use this observation and leverage anomaly detection techniques [28] to detect misconfigurations. The idea is that we can automatically learn an expected permissiveness level for each resource of an organization given that most policies will be correctly configured. Finally we use these learned models to detect deviations representing overly permissive or restrictive policies and alert operators.

Figure 2 shows a high-level overview of the three phases of our approach:

(1) **Graph Creation** builds a graph model from the identity and access management policies.
(2) **Graph Embedding** extracts relevant features from the graph to apply anomaly detection methods.
(3) **Anomaly Detection** trains on past, verified policy data and analyzes new, unverified policies for anomalous misconfigurations. Upon detection we alert an operator for manual verification.

3.1 Graph Creation

For the creation of the graph, we transform the policies of a cloud instance into a graph representation. The nodes of the graph represent one of the following types: policy, action, resource, or entity, where entity represents either a user, group, or role. Policy nodes represent the identity and access management policies specified in the environment. Action nodes represent the specified actions within the permissions in the policy, while resource nodes represent the resources on which the actions apply. Entity nodes are the identities that are present within the environment and can make use of the policies. The relationships between the nodes are defined as edges in the graph as follows:

```
(Entity)-[ATTACHED_TO]->(Policy)
(Policy)-[ALLOWS | DENIES]->(Action)
(Action)-[WORKS_ON]->(Resource)
```

3.2 Graph Embedding

While our graph model accurately represents IAM policies, they cannot directly be used by existing anomaly detection models. This is because anomaly detection models rely on features that indicate normal behavior and find deviations in those features. Therefore, we first transform the graph into a feature representation that anomaly detection models *can* interpret. We recall that intuitively, a large number of allowed actions and low number of disallowed actions attached to a resource may indicate overly permissive policies. Vice versa, a low number of allowed actions and high number of disallowed actions may indicate overly restrictive policies. However, these features may have different values depending on the attached resource, entities or even the type of organization implementing these policies. Therefore, we require method that captures the contextual differences of nodes attached to policies, i.e., type of attached entities, actions and resources. To this end, we use Node2Vec [21], an algorithm that uses random path sampling to create embeddings for each node in the graph [20]. The idea behind Node2Vec is that we select a target node in the graph and perform various random walks, starting from that node. During these walks, we record the types of other nodes we encounter and encode them into a low-dimensional feature space (we use the current best-practice of 128 features). By performing multiple walks for each node in the graph, we capture both information about the number of allowed and disallowed actions, but also of the attached resources and entities. If other nodes have a similar number of allowed and disallowed actions for similar resources and entities, the resulting vector representations will also be similar. We note that there exist other methods that capture similar information, such as graph2vec [27]. However, graph2vec embeds a graph as a whole, making it more difficult to identify which specific policy is misconfigured. As Node2Vec allows us to identify the individual policy node, we argue that it is a better graph embedding for our purpose.

3.3 Anomaly Detection

Now that we have embedded policy nodes of the graph, we can use anomaly detection methods to identify potential misconfigurations. We recall that similar policies will also have similar graph representations. Therefore, embeddings of properly configured policies will be similar to each other, and misconfigurations will be different. Anomaly detection techniques will be able to spot these outliers and thereby detect potential misconfigurations.

The anomaly detection model should ideally be trained on properly configured policies (we discuss misconfigurations in training data in Section 6.3). These can be policies that follow security policies, adhere to industry best practices, or are created through the use of existing tools to ensure proper configuration. Then, when policies change or when new policies are introduced, the anomaly detection model checks them for outliers and marks them as potential misconfigurations. This continuous monitoring of changes in cloud environments enables our approach to detect potential misconfigurations immediately when there is a change in policies. This quick reaction time minimizes the time a cloud environment is exposed, and therefore minimizes the possibility for abuse.

Based on our empirical evaluation in Section 6, we use Local Outlier Factor [10] (LOF) as our anomaly detection model. LOF is a density-based model, where a data point is considered an outlier if it has a lower local density compared to its neighbors. The model does not make any assumption on the probability distribution of the data and it provides some tolerance in case of outliers in the training set, making it a generic solution suitable for our approach.

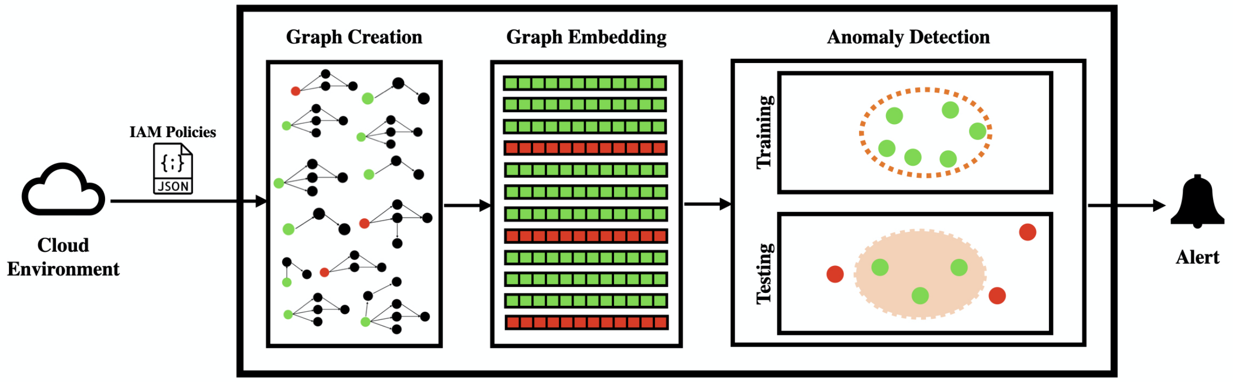


Figure 2: Overview of our proposed approach to detect potential misconfigurations. First, we create graph model representations of the IAM policies in the cloud environment (Graph Creation); then, we transform the graph model into vector representations (Graph Embedding), and finally, we apply anomaly detection to spot outliers and thereby detect potential misconfigurations.

4 IMPLEMENTATION

While we described our approach as a generic cloud IAM misconfiguration detection system, our implementation focuses on the AWS cloud platform, as it is currently the largest cloud provider [33].

Graph Creation. To create our graph, we pull the IAM policies from the monitored AWS environment in JSON format and transform the file to a Neo4j graph database [4]. We recall from Section 2 and Figure 1 that policies contain multiple statements that specify actions and resources. We create nodes for each policy, action and resource in the JSON file, and connect them with edges according to their relationship defined in the statements. We perform a similar operation for the entities defined in the JSON file and connect them to their corresponding policies. Figure 3 shows an example of a resulting graph, where yellow nodes represents the policy names, blue nodes represent actions, and purple nodes represent resources. Existing graphs can easily be updated if changes are made to the policies in the environment. We define a change as adding, removing or modifying nodes, edges or any of their properties. Changes in policies can be automatically detected from differing lines in IAM JSON files and can be applied as updates to the graph. This mechanism updates the graph rather than recreating it entirely, making it more efficient (see Section 6.4). Moreover, because the updates are triggered automatically, there is minimal overhead for security operators as they only have to focus on verifying detected misconfigurations.

Graph Embedding. Next, we transform a graph to vectors representing policies which we can then use in anomaly detection models. We run the algorithm Node2vec [21] included in Neo4j on each policy node in the graph to produce an embedded vector of 128 features (128 features are currently considered best practice). We then store the resulting embedded feature vectors as properties in the policy nodes. This way, we can access the embedded vectors by querying the database, enabling us to easily extract the vectors for anomaly detection.

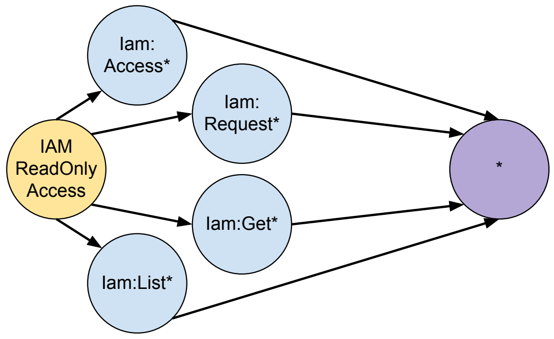


Figure 3: Example of a graph representation of a policy in the environment. Policies (yellow) allow actions (blue) on resources (purple).

Anomaly Detection. Finally, we use the scikit-learn Python library [29]. This library implements a wide range of machine learning algorithms, including the Local Outlier Factor algorithm and other algorithms used in our evaluation. More details on the implementation of the anomaly detection will be discussed in Section 6.

5 DATASET

To evaluate our prototype, we need real IAM policies containing labelled misconfigurations. We collected a total of 2480 different policies from three organizations that use AWS cloud environments over a period of several weeks. These datasets are detailed in Table 1. The first two datasets were collected from two different financial organizations with approximately 12,000 (dataset 1) and 130 (dataset 2) employees. The third dataset belongs to a smaller tech/software development company with 4 employees. These datasets represent small, medium and large enterprises, giving us a variety of real-world deployments. While this dataset is relatively small (three organizations), we use all the deployed policies from real-world environments, minimizing potential sampling bias. A further advantage is that we have a realistic ratio of misconfigurations (which are relatively scarce), minimizing any base rate fallacy.

We collected this data with a script that uses the AWS CLI [1] to periodically pull IAM policies from the cloud environments of these organizations[1]. We collected all IAM policies, as well as all entities (user, groups, and roles) from each environment and store it locally in a spreadsheet. Please note that collecting AWS IAM policies can contain sensitive data about the resources used by an organization and should therefore be treated confidently or be sanitized (e.g., replacing ARNs with generic identifiers) when shared with third parties.

As shown in Table 1, the organizations implemented between 812 and 842 policies. This relatively high number of policies is due to the fact that all companies used the 515 default IAM policies provided by AWS themselves. Each company added between 297 and 327 custom policies for their environment. The first two datasets do not contain any users or user groups because those environments authenticate users with a separate identity provider that issues temporary cloud credentials. This automatically ensures users assume roles rather than having permissions on their own user account, a common industry best-practice. We note that this does not impact our approach as we treat users, groups and roles all as entities. Furthermore, we collected IAM policies of each organization at regular intervals during a period of multiple weeks (least 2 as shown in Table 1). This allows us to evaluate the performance of our approach when misconfigurations are introduced after the initial configuration.

Data Labeling. To evaluate our proposed approach, the collected policies were manually labelled into correct and misconfigured policies. Each policy has been manually reviewed with respect to the level of permissiveness. Policies that contain a high number of allowed actions, on a high number of resources, were considered with extra care. Each policy has been compared against current industry best practices [8], and, by following these guidelines, we attempted to minimize label inaccuracy. An example of a potential misconfiguration is the previously mentioned AdministratorAccess policy (see Figure 1). This policy has a high level of permissiveness since it grants permission to perform all the actions on all the resources. When not attached to the proper entities this policy could be a potential misconfiguration. All three datasets were labeled, and contain 12, 11, and 6 misconfigurations, respectively. During our evaluation, we analyze misconfigurations that were introduced after initial configuration. However, our labelling process showed that no new misconfigurations were introduced in the modifications. Therefore, in those sections, we simulate the temporal aspect by manually introducing misconfigurations after the initial setup.

6 EVALUATION

To evaluate our prototype, we take care to follow good practices in machine learning detailed by Arp et al. [5] and TESSERACT [30]. In each experiment, we briefly discuss how we tried to avoid relevant common pitfalls.

We evaluate our prototype by comparing it against Cloud Custodian [19], one of the main rule-based tools used for validating many AWS IAM policies, which can be seen as our experimental baseline. Cloud Custodian allows operators define custom rules to ensure deployed IAM policies adhere to organizational requirements. The organizations represented in our dataset did not have any predefined rules in place. Thus, we instead compared our approach against Cloud Custodian using open-source IAM rules[2]. We note that open-source rules such as those used for this experiment are not tailored towards a specific setting and will therefore likely not give the best possible results. To make the comparison more fair, we also performed the same experiment where we manually selected open-source rules that are applicable to the organizations in our dataset, removing rules producing incorrect detections. This emphasizes the fundamental limitation of rule-based systems, i.e., there are no generic rules that can be applied to all cloud environments. Hence, an anomaly-based solution like ours can help reduce the manual effort required for detecting IAM misconfigurations while potentially detecting more misconfigurations.

Table 2 shows the performance of both our approach using the LOF anomaly detector (see Section 6.1), and the two rulesets of Cloud Custodian on all three datasets split randomly into 90/10 training and testing sets. While Cloud Custodian performs better over all policies, we find that for misconfigurations, our approach performs better, correctly detecting between 3.7 and 6.4 times as many misconfigurations. The reason for this is that rule-based systems such as Cloud Custodian are very precise in their detection, i.e. if there exists a rule for a misconfiguration, it will only trigger for the misconfiguration and not for other rules. Our approach on the other hand is anomaly-based, meaning that it may incorrectly flag some correct configurations as misconfigurations, but it is also able to detect misconfigurations not captured by rules. Therefore, our approach shows a promising direction for detecting additional misconfigurations using anomaly detection, improving the overall cloud security.

Misclassifications. During our evaluation, we found some occurences of false positives (correct policies flagged as misconfigurations) and false negatives (undetected misconfigurations). The common characteristics of our false positives are high levels of permissiveness, which in many scenarios are misconfigurations, but under certain circumstances can be allowed. An example of such policy is the *ReadOnly* policy, which, in dataset 1 permits 762 read-only actions, making it very permissive. However, since in this dataset, all read-only permissions are for on non-critical resources the policy was not considered a misconfiguration. As our anomaly detector does not have a sense of which resources are critical or not, our approach plays it safe and classifies these policies as potential misconfigurations.

Conversely, false negatives can occur when a policy does not seem permissive but actually allows certain high-impact actions. A policy such as the *PowerUserAccess* allows only a small number of actions, for a limited number of entities, but works on critical resources and can have a high impact. Therefore, to reduce the number of misclassifications, further research is needed into methods that take into account the impact of policies.

[1]Code available at https://github.com/utwente-scs/misdet-code

[2]https://github.com/davidclin/cloudcustodian-policies

Table 1: Overview of three datasets used for the evaluation. We show the size of the datasets in terms of number of employees working for each organization as well as the number of policies, users, groups, roles defined in the IAM policies. To evaluate changes in datasets, we collected policies multiple times over a period of multiple weeks indicated by *Number of collections.*

Dataset	Total number of employees	policies	users	groups	roles	Number of collections
1	12,000	842	0	0	55	8
2	130	812	0	0	34	2
3	4	826	2	1	10	12

Table 2: Overall evaluation. Performance of our approach compared with Cloud Custodian using all rules, and using rules specifically selected for our dataset. We show the performance for detecting misconfigurations (top) and the overall performance (bottom). While Cloud Custodian seems to perform better overall, its recall is low, meaning the majority of misconfigurations go undetected.

		Our approach			Cloud Custodian All rules			Cloud Custodian Selected rules		
	DS	Prec.	Recall	F1-score	Prec.	Recall	F1-score	Prec.	Recall	F1-score
Misconf.	1	66.67%	66.67%	66.67%	7.89%	10.34%	4.48%	100.00%	10.34%	9.37%
	2	70.00%	63.34%	66.67%	13.73%	17.07%	7.61%	100.00%	17.07%	14.58%
	3	75.00%	50.00%	60.00%	15.38%	11.32%	6.52%	100.00%	11.32%	10.17%
Overall	1	91.58%	91.58%	91.58%	97.93%	97.60%	97.76%	98.99%	98.98%	98.57%
	2	92.03%	92.31%	92.15%	97.40%	97.09%	97.24%	98.75%	98.73%	98.28%
	3	94.97%	95.45%	95.03%	98.93%	97.88%	96.87%	98.12%	98.08%	97.33%

6.1 Anomaly Detectors

We have shown that our approach correctly detects more misconfigurations than rule-based approaches using LOF as our anomaly detector. The only requirement for our detector is that it can detect anomalies based on a vector representation. Therefore, before choosing LOF, we compared four different anomaly detection techniques to empirically find the best performing technique for our approach: 1) One-Class Support Vector Machine; 2) Local Outlier Factor; 3) Isolation Forest; 4) Robust Covariance.

For this experiment, we limit ourselves to dataset 1 to minimize the data snooping bias that the result of this experiment may introduce to our overall evaluation. Ideally, such evaluation would be performed on a separate dataset to exclude bias completely. However, due the limited availability of data we chose to still report the performance of dataset 1 in Table 2, but show the results separately for each dataset, demonstrating that the performance generalizes accross datasets. We follow our proposed approach as described in Section 3. First, we create a graph model representation from the first collected IAM policies. Second, we apply node2vec to embed the policy nodes into vector representations. The vector representations are the same for all four tested anomaly detectors. Next, we split dataset 1 into a 90/10 training and testing set, following machine learning best practices [5, 30]. Dataset 1 contains a total of 842 policies conatining 12 misconfigurations. To ensure that the anomaly detection model is solely trained on correctly configured policies, we temporarily remove the misconfigurations from the dataset, leaving us with 830 correct policies. Using a 90/10 split, we create a training set consisting of 747 (90%) correct policies without any misconfiguration. The test set consists of the remaining 83 (10%) correct policies, and the 12 misconfigurations, for a total test size of 95 policies. We choose a 90/10 training testing split as IAM policy changes are often relatively small compared to the existing policies. In these environments, the majority of policies are created initially when the environment is setup. New policies are added through the life-cycle of the environment, but comprise a smaller part of the total amount. Additionally, the test set is imbalanced as there are considerably more correct policies than misconfigurations which is also representative of real-world deployments, since misconfigurations are introduced much less frequently than correct modifications [5, 30]. In this setting, the training set can be seen as the initial deployment of a cloud environment. The test set then simulates the changes made to the policies in the environment, both additions of policies as well as modifications, and can therefore be considered new observations.

Each anomaly detector goes through the same process outlined in Section 3.3. We first train the model using the correct training policies. Then, we evaluate the performance of the model using the test set, containing both correct policies and misconfigurations. We evaluate the performance using the following metrics: precision, recall, F1-score, and ROC Area Under Curve (ROC AUC). The precision measures the proportion of correctly identified misconfigurations in all detected misconfigurations. The recall measures the proportion of correctly identified misconfigurations in all actual misconfigurations. The F1-score is the harmonic mean of precision

Table 3: Comparison of anomaly detectors. Performance of anomaly detectors after parameter optimization (dataset 1).

Algorithm	Precision	Recall	F1-Score	AUC
One-Class SVM	88.78%	89.47%	89.06%	0.70
Local Outlier Factor	91.58%	91.58%	91.58%	0.66
Isolation Forest	84.08%	87.37%	84.50%	0.60
Robust Covariance	87.94%	89.47%	87.90%	0.65

and recall, and conveys the balance between the precision and the recall. The ROC AUC measures the relationship between the True Positive and False Positive Rate of the anomaly detector.

Parameter optimization. The performance of each anomaly detector depends on the use of specific parameters that determine how each algorithm separates misconfigurations from valid configurations. Therefore, we performed a grid-search on dataset 1 to find the optimal parameters for each of the four anomaly detection algorithms. The effect of the parameters on the performance metrics can be found in Figure 4. For each parameter, we choose the best performing according to the ROC AUC metric as a high ROC AUC will not only give a high precision, but will also be robust against variations in the data. Using our grid-search, we found the following optimal parameters for each anomaly detector:

- One-Class SVM: gamma = 0.001, nu = 0.5
- Local Outlier Factor: n_neighbours = 5
- Isolation Forest: n_estimators = 30
- Robust Covariance: contamination = 0.1

Results. With the aforementioned selected optimal parameters, we evaluate the performances of the four anomaly detection techniques. Table 3 shows the obtained overall performance for each anomaly detector for their optimal parameters. Our first observation is that all four techniques have relatively high precision and recall. Both metrics are important since precision measures whether detected misconfigurations are *actual* misconfigurations, while recall measures the proportion of *detected* misconfigurations. In our approach, however, we prefer a high recall above a high precision because potentially missed misconfigurations have a considerably larger impact to cloud security than a false positive detection, which can be manually filtered out by an operator.

From the results in Table 3 we find that the Local Outlier Factor (LOF) has the best performance in precision, recall and F1-score. The ROC AUC of the One-Class SVM is slightly higher, meaning that the choice of decision boundary is slightly more robust and will therefore depend less on the chosen parameters. Nevertheless, the ROC AUC of the LOF algorithm is the second highest and gives a better overall performance, making it our algorithm of choice. Additionally, LOF has the great advantage of being more resilient in cases where the training set contains anomalous data points (see Section 6.3). In fact, by measuring the local deviation of data points with respect to their neighbors, LOF can identify local outliers in the training set, providing some tolerance for misconfigurations already present during the training.

6.2 Parameter Transferability

Our approach shows promising results for detecting more misconfigurations than rule-based approaches. However, with large manual effort, rules may be tweaked to the extent that they cover nearly every edge case. Therefore, we want to evaluate whether our anomaly-based approach requires a similar level of tweaking, or whether our internal parameters are transferable accross settings. To this end, we take our optimal LOF parameter (n_neighbours = 5) for dataset 1 as found in Section 6.1 and apply them to datasets 2 and 3 to see how well they generalize.

Analogous to our approach, we create separate graph models for the two new datasets and embed the policy nodes into vector representations. We then apply the same 90/10 training testing split as in the previous experiment and obtain 737 correct policies for training and 11 misconfigurations and 64 correct policies for testing dataset 2. For dataset 3, the training set contains 743 correct policies and the testing set contains 6 misconfigurations, and 77 correct policies. We create separate LOF models for the training sets, using the n_neighbours = 5 parameter and evaluate the performance using the test sets.

Table 4 shows the results of this experiment. We observe that all metrics are quite similar compared to the baseline, and even slightly improved. The reason that parameters are transferable is the relatively high similarity between AWS IAM policies. By default, there are 515 IAM policies managed and provided by AWS, meaning that there is a significant overlap (> 60%) of policies between the datasets. As these default policies are well-checked, our model is trained on properly configured policies that are available in nearly all real-world scenarios. The small variance between our baseline and the other datasets can be explained by the level of permissiveness for policies in the dataset. In dataset 1, policies were more permissive, without being classified as misconfigurations, meaning it is more difficult to distinct between correct and misconfigured policies. In conclusion, we find that that the parameter, n_neighbours = 5, seems to be transferable between datasets.

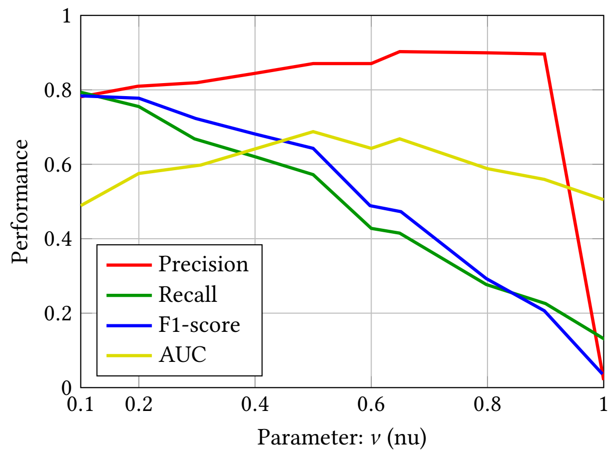
6.3 Misconfigurations in Training Data

As our approach is based on anomaly detection, misconfigurations in training data may lead to undetected misconfigurations during deployment. To test the resilience against misconfigurations in the training data, we introduced known misconfigurations during the training phase of all four algorithms and tested the performance with respect to the amount of introduced misconfigurations.

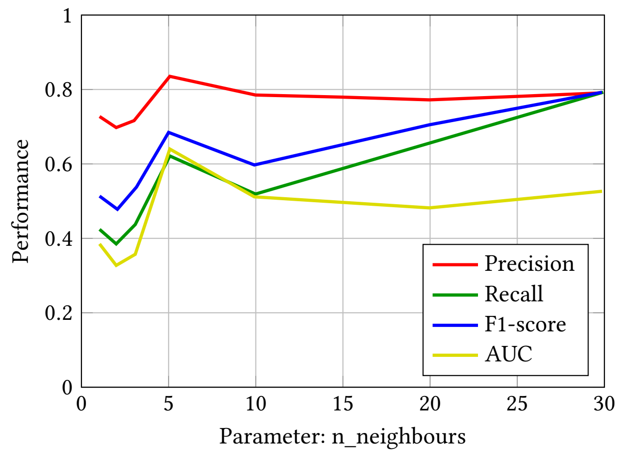
Table 5 shows the result of this experiment. We see that for the One-class SVM and Isolation Forest, the recall drops significantly with respect to the data in Table 3. The Local Outlier Factor and

Table 4: Parameter transferability. LOF detection performance on dataset 2 and 3, with parameter n_neighbours = 5.

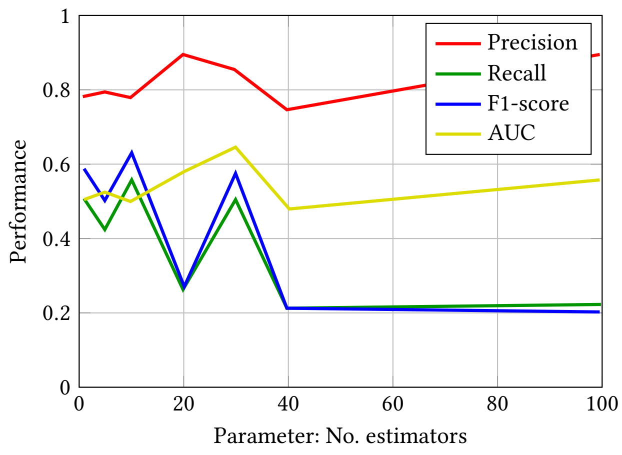
Dataset	Precision	Recall	F1-Score	ROC AUC
1 (baseline)	91.58%	91.58%	91.58%	0.66
2	92.03%	92.31%	92.15%	0.73
3	94.97%	95.45%	95.03%	0.72



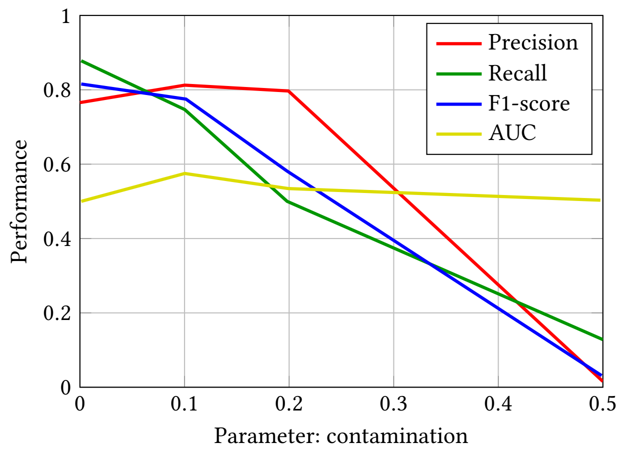
(a) One-Class SVM



(b) LOF



(c) Isolation Forest



(d) Robust Covariance

Figure 4: Parameter selection. Detection performance metrics during parameter optimization for the four anomaly detection techniques

Robust Covariance algorithms are much more resilient against misconfigurations in the training data. We also note that increasing the number of misconfigurations seems to increase the precision, recall and F1-score. However, this is due to fewer misconfigurations in the test data as they were included in the training data instead. From Table 2, we found that evaluation metrics over misconfigurations are relatively lower, hence having fewer misconfigurations slightly skews the evaluation upwards. Nevertheless, our experiments show that both the Local Outlier Factor and Robust Covariance algorithms are relatively robust against misconfigurations in the training data.

6.4 Runtime Performance

During our experiments, we measured the runtime performance of the graph creation and model training stage of our approach for various input sizes. All experiments were performed on an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz laptop running Ubuntu 20.04 LTS. We report the average runtime over 10 runs.

Graph Creation. During Graph Creation, the IAM policy data is transformed into a graph model representation. Figure 5 shows the runtime performance of the Graph Creation stage. The performance scales linearly, but larger datasets can take up a significant amount of time. For our largest dataset (1) with 842 policies, graph creation took 20 minutes and 55 seconds. It is worth mentioning that only the initial graph creation is this long, due to the fact that all nodes must be created. Graph updates are considerably faster since the number of new nodes is generally low. To evaluate this, we performed 10 experiments where we took the graph of Dataset 3 consisting of 826 nodes. Next, we added 10 random policies and added these to the graph, which took on average 0.581 seconds versus 833.986 seconds for recreating the graph. This shows that our graph update achieved a speedup of a factor 1091 versus recreating the graph from scratch. While the speedup depends on both the total size of the graph (influencing the recreation of the graph) and the number

Table 5: Misconfigurations in training data. We show the influence of including a certain number of misconfigurations in the training dataset.

Algorithm	No. Misconfigurations	Precision.	Recall	F1-score
One-class SVM	1 (0.12% of training)	90.68%	50.05%	63.15%
One-class SVM	5 (0.61% of training)	90.81%	48.20%	61.83%
One-class SVM	10 (1.21% of training)	92.09%	49.71%	63.49%
Local Outlier Factor	1 (0.12% of training)	91.07%	95.24%	93.10%
Local Outlier Factor	5 (0.61% of training)	91.78%	95.61%	93.65%
Local Outlier Factor	10 (1.21% of training)	92.67%	95.78%	94.20%
Isolation Forest	1 (0.12% of training)	88.67%	17.10%	23.94%
Isolation Forest	5 (0.61% of training)	91.08%	28.47%	39.81%
Isolation Forest	10 (1.21% of training)	91.86%	30.10%	42.84%
Robust Covariance	1 (0.12% of training)	90.64%	81.73%	85.91%
Robust Covariance	5 (0.61% of training)	91.57%	85.07%	88.17%
Robust Covariance	10 (1.21% of training)	92.37%	82.65%	87.19%

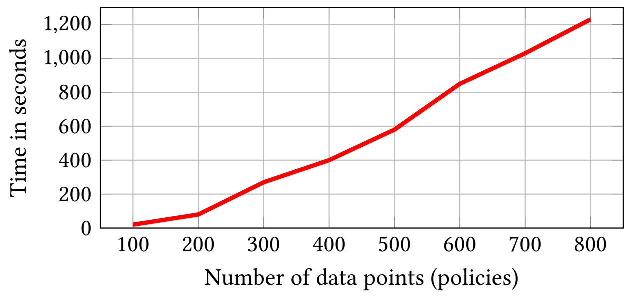


Figure 5: Overview of runtime performance for the creation of the graph model

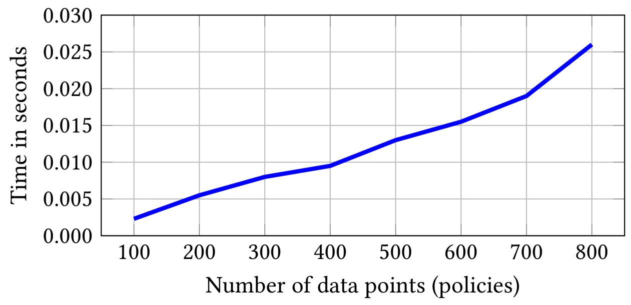


Figure 6: Overview of runtime performance for training the LOF anomaly detector.

of added nodes (influencing the average update time), we believe our experiment shows a considerable speedup in a realistic scenario.

Graph Embedding. After the graph has been created, we run the Node2vec algorithm on the policy nodes to transform nodes into vectors, which took on average 57 milliseconds for our entire dataset.

Model Training and Prediction. We also consider the model training overhead of the Local Outlier Factor anomaly detection model. The LOF model is trained on correct policies and then used to determine whether new observations are also correct or potential misconfigurations. The measured runtime performance of the LOF model can be found in Figure 6. We find that the runtime performance is negligible with respect to creating the graph model with 26 milliseconds for our largest dataset. For the prediction, we measured an average of 6 milliseconds for our largest test dataset of 86 policies and is therefore also negligible.

7 DISCUSSION AND FUTURE WORK

Challenges. We recall from Section 2 that our approach aims to be **proactive**, **context-specific**, and **low maintenance**. Our approach can be automatically triggered upon changes in IAM policies as explained in Section 4. Furthermore, our evaluation in Section 6.4 has shown that our approach can also be run in a reasonable timespan, allowing for **proactive** misconfiguration detection. Additionally, our evaluation in Table 2 has shown that our anomaly-based approach detects more misconfigurations, compared to generic rule-based approaches, meaning it is better able to take into account the **context-specific** (mis)configurations for an organization. And finaly, with our Parameter Transferability evaluation in Section 6.2, we have shown that our approach is **low maintenance**, showing promising results for all our criterea.

Misconfigurations in training data. Our approach assumed that we only train the anomaly detection model on correct policies. While our evaluation in Section 6.3 showed that Local Outlier Factor and Robust Covariance have some robustness with small numbers of misconfigurations, ideally the training data should not contain misconfigurations. The policy data collected for our experiments is from real-world cloud environments. Each dataset has been manually reviewed, and potential misconfigurations have been removed from the training set. This is a costly human operation that is error prone and may not be feasible in all use cases. Therefore, a possible approach could be to train only on the 515 default IAM policies provided by AWS. In this scenario, we can be sure that there are no misconfigurations in the training data. As a disadvantage, our approach will not be able to learn the context-specific environment and may flag policies that seem misconfigured with respect to IAM policies, but are permissible in certain cases. In short, we believe that with careful review, the number of misconfigurations that slip through manual detection is sufficiently low, and we have shown that for low numbers of misconfigurations, Local Outlier Factor and Robust Covariance still produce good results.

Advanced embedding and anomaly detection. In our approach, we have used the graph embedding technique Node2vec, which is currently the state-of-the-art of graph embedding. Node2vec in combination with LOF has already provided us with good results in detecting potential misconfigurations. There are however newer techniques in the making that might be able to transform the graph in a better and more efficient way. An example of such a new embedding technique is GraphSage [22], which uses inductive representation learning to also enable the embedding of node properties. Furthermore, we have only considered four anomaly detection algorithms in our approach. More complex machine learning techniques could further enhance the performance of our approach. Examples of such techniques are Graph Convolutional Networks [25], and One-Class Neural Networks [13].

Different policy types. In our current approach, we only consider identity-based policies. There are, however, more IAM policie types in cloud environments[3]:

- Resource-based policies,
- Permission boundaries,
- Organization service contol policies,
- Access control lists,
- Session policies.

These policies cover different scenarios and can be represented using other node types in a graph. E.g., a resource-based policy can set a storage service node to be either publicly or privately accessible. Such policies specify actions that are only allowed on that specific resource, regardless of who is performing the action. These different policy types can be created and stored in the same way as identity-based policies but may show different behaviour when analyzed using our approach. Therefore, we recommend future investigation into using our approach with different policy types.

Future Work. From our current approach, we see several potential future directions for improvement. First, a feedback loop may be added after a security operator verifies or discards flagged misconfigurations. The model can be updated with such new information and prevent other similar alerts in the future.

Second, our work focused on the AWS cloud platform. While we only evaluated our approach on AWS IAM datasets, the technique itself may be extended to other cloud platforms that provide IAM policies where entities can be linked to resources through policies. Besides AWS, other major cloud platforms such as Google Cloud Platform and Microsoft Azure support such policy structures. Our approach can be extended by adding collection services for additional cloud providers. While the rest of our approach should be independent of the cloud provider, future research should show whether our approach achieves similar results on those platforms.

[3]https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html

Finally, our approach is able to detect misconfigurations, but is not yet able to provide suggestsions for IAM policy modifications. To this end, we could explore link prediction techniques that indicate which links should be added or removed. While link prediction is technically possible to implement in our graph, we consider this out of scope for the current research.

8 RELATED WORK

Access control is a subfield of the broader area of identity and access management, and has been studied extensively. P-Diff [35] monitors access and control behavior by using decision tree algorithms. While effective, P-Diff has one major limitation: it learns access control policies from access logs. This means that detection is therefore limited to the information contained in the access logs, which makes the approach reactive. As a consequence, misconfigurations can only be detected after abuse, once anomalies show up in the access logs. Instead, our approach is proactive and aims at identifying misconfigurations as soon as they are introduced in the cloud environment. Baaz [15] infers permission misconfigurations in an enterprise network by monitoring updates made to the access control metadata, and looking for potential inconsistencies among peers. The major limitation of Baaz is that it relies on the definition of what should be considered as an inconsistency. This parameter can be tweaked by administrators, but could still cause problems and influence the performance of the system.

Rule-based solutions rely on predefined rules to which the newly created or modified cloud resources must adhere—these rules have to be created, monitored and maintained throughout the life cycle of the cloud environment. Cloud Custodian [19] is a widely used open-source rule-based system. Cloud Custodian enables users to be well managed in the cloud. It allows for the easy definition of rules to manage the cloud infrastructure. These rules are collected in policies. The policies can be as simple or as complex as the person creating them wants them to be. Examples of such policies can be the blocking of all the public access to S3 buckets or the detection of an account receiving admin privileges. AWS Remediation Framework [17] is another example of an open-source solution. As the name suggests, it is a project that identifies and remediates AWS security issues to ensure AWS usage is compliant with a set of rules. Although these rule-based solutions can be very powerful and have clear advantages, there are also a number of limitations. First of all, the rules need to be created and maintained to adhere to security policies. This has to be performed manually and can require a large effort. Furthermore, this process can be error prone and security policies can be insufficient to detect all misconfigurations. Secondly, cloud environments are generally extremely dynamic and change frequently. There are situations in which a certain action can be seen as a misconfiguration, while it is needed for a certain operation, predefined rules can therefore be too rigid too handle these quick changes, which will impact the performance of the system.

Cloud providers have also started offering solutions for detecting misconfigurations. AWS provides CloudTrail [3], which is an AWS service that enables governance, compliance, and auditing of the AWS account and all the corresponding resources. It provides logging and continuous monitoring of the AWS environment. Cloudtrail can be used in two ways to detect misconfigurations.

First, it can be used to log and raise alerts in case any changes are made to the identity and access management configurations of the cloud resources. Second, it can be used to detect unauthorized access if misconfigurations are abused. Both ways have limitations. Either over-alerting administrators on every change made, or reacting to already happened abuse, thus being too late. Besides, AWS has some mechanisms in place to prevent misconfigurations. For example, when overly permissive identity and access management roles are created, the system raises a warning. This already creates a first line of defense, however, it can be easily overridden by administrators and only identifies major and obvious errors.

Other studies, which are orthogonal to our research, focused on measuring misconfigurations. Continella et al. [14] investigated permission misconfigurations on Amazon Simple Storage Service (S3) buckets [7]. Another research has been performed on the cause of data leaks when cloud platforms are used as mobile app backends [38]. Finally, Zahoor et al. developed a formal method for detecting conflicting policies [36] and extended this method to work accross multiple cloud providers [37]. These conflicts cover cases where some policies allow entities access to a resource, whereas another policies deny overlapping entities from accessing to same resource. While such policies are also misconfigurations, they are orthogonal to the context-specific overly permissive or restrictive policies that our work focuses on.

9 CONCLUSION

In this paper, we presented a novel approach for detecting misconfigurations of AWS identity and access management policies. The goals for this approach were to be proactive, context-specific, and requiring low maintenance. To achieve these goals, we first created a graph model representing IAM policies from a given cloud environment. We then created context-specific representations of all policies using node2vec embeddings. Finally, we trained an anomaly detection model on correct policy embeddings and used it to detect potential misconfigurations in new policies. We have evaluated our approach on real-world IAM policies from three organizations and have shown that our approach correctly detects between 3.7 and 6.4 times more misconfigurations than state-of-the-art approaches at the cost of a slight decrease in precision. Furthermore, we have shown that the parameters for the anomaly detection algorithms are transferable between environments, while still maintaining a similar detection performance, ensuring low maintenance costs. As security misconfigurations in cloud environments have detrimental consequences, our approach performs an important step to reduce the risk of security misconfigurations.

REFERENCES

[1] 2006. AWS Command Line Interface. https://aws.amazon.com/cli/

[2] 2020. Misconfigured AWS S3 Bucket Leaks 36,000 Inmate Records. https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/misconfigured-aws-s3-bucket-leaks-36-000-inmate-records

[3] 2021. AWS CloudTrail. https://aws.amazon.com/cloudtrail/

[4] 2021. Graph Database Platform: Graph Database Management System: Neo4j. https://neo4j.com/

[5] 2022. Dos and Don'ts of Machine Learning in Computer Security. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/arp

[6] AmazonWebServices. 2003. AWS IAM. https://aws.amazon.com/iam/

[7] AWS. 2002. Amazon Simple Storage Service (Amazon S3). https://aws.amazon.com/s3/

[8] AWS. 2003. AWS IAM Best Practices. https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html

[9] Kelly W Bennett and James Robertson. 2019. Security in the Cloud: understanding your responsibility. In *Cyber Sensing 2019*, Vol. 11011. International Society for Optics and Photonics, 1101106.

[10] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. , 93–104 pages.

[11] JM Brook, A Getsin, G Jensen, L Jameson, M Roza, N Thethi, A Kurmi, S Levy, S Shamban, V Hargrave, et al. 2019. Top Threats to Cloud Computing: The Egregious Eleven. *Cloud Security Alliance* (2019).

[12] Herbert G. Buff. 2000. Compliance. https://aws.amazon.com/compliance/shared-responsibility-model/

[13] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. 2018. Anomaly detection using one-class neural networks. *arXiv preprint arXiv:1802.06360* (2018).

[14] Andrea Continella, Mario Polino, Marcello Pogliani, and Stefano Zanero. 2018. There's a Hole in that Bucket! A Large-scale Analysis of Misconfigured S3 Buckets. In *Proceedings of the ACM Annual Computer Security Applications Conference (ACSAC)*.

[15] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. 2010. Baaz: A System for Detecting Access Control Misconfigurations.. In *USENIX Security Symposium*. 161–176.

[16] Constanze Dietrich, Katharina Krombholz, Kevin Borgolte, and Tobias Fiebig. 2018. Investigating system operators' perspective on security misconfigurations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1272–1289.

[17] Flatironhealth. 2021. Aws Remediation Framework. https://github.com/flatironhealth/aws-remediation-framework

[18] Emily Flitter and Karen Weise. 2019. Capital One Data Breach Compromises Data of Over 100 Million. https://www.nytimes.com/2019/07/29/business/capital-one-data-breach-hacked.html

[19] Linux Foundation. 2020. Cloud Custodian. https://cloudcustodian.io/

[20] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.

[21] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 855–864.

[22] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216* (2017).

[23] Muhammad Kazim and Shao Ying Zhu. 2015. A survey on top security threats in cloud computing. (2015).

[24] Issa M Khalil, Abdallah Khreishah, Salah Bouktif, and Azeem Ahmad. 2013. Security concerns in cloud computing. In *2013 10th International Conference on Information Technology: New Generations*. IEEE, 411–416.

[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[26] Nag. 2021. S3-Leaks. https://github.com/nagwww/s3-leaks

[27] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv preprint arXiv:1707.05005* (2017).

[28] Salima Omar, Asri Ngadi, and Hamid H Jebur. 2013. Machine learning techniques for anomaly detection: an overview. *International Journal of Computer Applications* 79, 2 (2013).

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. , 2825–2830 pages.

[30] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th USENIX Security Symposium (USENIX Security 19)*. 729–746.

[31] Tara Seals. 2020. Good Heavens! 10M Impacted in Pray.com Data Exposure. https://threatpost.com/10m-impacted-pray-com-data-exposure/161459/

[32] Tara Seals. 2020. Millions of Hotel Guests Worldwide Caught Up in Mass Data Leak. https://threatpost.com/millions-hotel-guests-worldwide-data-leak/161044/

[33] Katy Stalcup. 2020. AWS vs Azure vs Google Cloud Market Share 2020: What the Latest Data Shows. https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/

[34] Lance Whitney. 2021. 2020 sees huge increase in records exposed in data breaches. https://www.techrepublic.com/article/2020-sees-huge-increase-in-records-exposed-in-data-breaches/

[35] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. 2019. Towards Continuous Access Control Validation and Forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 113–129.

[36] Ehtesham Zahoor, Zubaria Asma, and Olivier Perrin. 2017. A formal approach for the verification of AWS IAM access control policies. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 59–74.

[37] Ehtesham Zahoor, Asim Ikram, Sabina Akhtar, and Olivier Perrin. 2018. Authorization policies specification and consistency management within multi-cloud environments. In *Nordic Conference on Secure IT Systems*. Springer, 272–288.

[38] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1296–1310.