7-2022

Using constraint programming and graph representation learning for generating interpretable cloud security policies

Mikhail KAZDAGLI

Mohit TIWARI

Akshat KUMAR
Singapore Management University, akshatkumar@smu.edu.sg

Using Constraint Programming and Graph Representation Learning for Generating Interpretable Cloud Security Policies

Mikhail Kazdagli[1,2] , **Mohit Tiwari**[1,2] , **Akshat Kumar**[1,3]
[1]Symmetry Systems
[2]The University of Texas at Austin
[3]Singapore Management University
{mikhail, mohit, akshat.kumar}@symmetry-systems.com

Abstract

Modern software systems rely on mining insights from business sensitive data stored in public clouds. A data breach usually incurs significant (monetary) loss for a commercial organization. Conceptually, cloud security heavily relies on Identity Access Management (IAM) policies that IT admins need to properly configure and periodically update. Security negligence and human errors often lead to misconfiguring IAM policies which may open a backdoor for attackers. To address these challenges, *first*, we develop a novel framework that encodes generating *optimal* IAM policies using constraint programming (CP). We identify reducing *dormant permissions* of cloud users as an optimality criterion, which intuitively implies minimizing unnecessary datastore access permissions. *Second*, to make IAM policies interpretable, we use graph representation learning applied to historical access patterns of users to augment our CP model with *similarity* constraints: similar users should be grouped together and share common IAM policies. *Third*, we describe multiple attack models and show that our optimized IAM policies significantly reduce the impact of security attacks using real data from 8 commercial organizations, and synthetic instances.

1 Introduction

Cloud computing has recently become the dominant computing paradigm which provides the flexibility of on-demand compute, and reduced cost due to economies of scale. However, such benefits come at a cost—private and business sensitive data is stored on public clouds. Managing identity access policies (IAM), which intuitively means deciding which user should have access to which datastore, for public clouds such as Amazon AWS [IAM, 2021], is complex even for small and medium-sized companies with few hundred users and datastores [Kuenzli, 2020]. IAM policies are configured by IT admins of the organization who often do not have access to automated decision support tools for IAM policy optimization. As a result, due to the inherent complexity of IAM policy optimization, security negligence, and human errors may often lead to misconfigured IAM policies and result in data breaches [Security, 2021; Scroxton, 2020; Marks, 2021b; Marks, 2021a].

Related work. There has been prior work in AI and ML on securing the sensitive data in a cloud. Traditionally security researchers focus on developing dynamic behavioral detectors that analyze system behavior using (un)supervised ML methods at different levels - system and API calls [Canali *et al.*, 2012; Wu *et al.*, 2012], hardware signals [Demme *et al.*, 2013; Kazdagli *et al.*, 2016], network traffic [Mirsky *et al.*, 2018; Sommer and Paxson, 2010; Handley *et al.*, 2001], and domain reputations [Huang *et al.*, 2017; Oprea *et al.*, 2018]. Such methods often suffer from raising unacceptably high volume of false positives [Hassan *et al.*, 2019] when analyzing large amount of system events even if the false positive rate is very low [Axelsson, 1999]. Moreover, ML-based detectors are susceptible to adversarial attacks [Lei *et al.*, 2019; Wang *et al.*, 2019; Liu *et al.*, 2021]. Our proposed method, IAMAX, avoids such shortcomings by securing an organization's cloud infrastructure by hardening IAM policies, which are the first line of defense, rather than detecting attackers inside the cloud. Even after a user's account is compromised, our optimized IAM policies minimize the leak of sensitive data.

There has been a history of applying tools from AI planning for security in the context of penetration testing (or pentesting) [Sarraute *et al.*, 2012; Hoffmann, 2015; Shmaryahu *et al.*, 2018]. In pentesting, automated tools based on planning are used to identify vulnerabilities in the network by launching controlled attacks under a variety of settings, such as fully or partially observable network settings. Once vulnerabilities are identified, they can be patched by network administrators. Our proposed work is different from such pentesting methods as our target is to *design* the IAM policies (analogous to network design) from grounds up so that opportunities for catastrophic attacks (where compromising very few users gives hackers access to majority of datastores) are minimized. Furthermore, our approach is customized for public clouds and their security configurations, which is a different setting than the standard pentesting [Shmaryahu *et al.*, 2018].

Contributions. Our main contributions are as follows. *First*, we formalize the problem of IAM policy optimization for public clouds using the constraint programming (CP) framework [Rossi *et al.*, 2006], identfying core objectives and

constraints. We highlight the key role that *dormant permissions* play in defining secure IAM policies. *Second*, given that organizations that we have worked with do not provide the identities and job roles of its cloud users due to privacy concerns, we use graph neural network [Kipf and Welling, 2017; Hamilton *et al.*, 2017] to learn embeddings for users and datastores based on the information flow between them. These embeddings are used for defining constraints that make IAM policies interpretable. *Third*, we describe multiple attack models, based on randomly compromising k users, and adversially compromising those users that lead to worst case outcome. We test our optimized IAM policies on real cloud infrastructures of 8 medium size commercial companies and several realistic synthetic instances generated using the properties of real world data sets, and show the significant potential of our approach to secure cloud infrastructures by reducing dormant permissions significantly. We released the code and the data sets used in our experiments[1].

2 IAM Policies and Cloud Computing

An IAM policy is the fundamental security primitive in any cloud. Though our real world data sets were collected across multiple cloud platforms, we use Amazon AWS IAM policies as an example. An IAM policy is expressed using a declarative policy language that defines what operations (**"Action"**) are (dis)allowed (controlled by the **"Effect"** field) on a resource (**"Resource"** field). Figure 1 shows an oversimplified IAM policy to be attached to an AWS identity (e.g. a user, a role) or a *group* of identities. It stipulates that the identity that the policy is attached to is allowed to perform the operation "s3:ListObjects" (read the bucket content) on the S3 bucket "arn:aws:s3:::bucket-name".

Overpriviliged security policies. To make the number of policies manageable, IT admins group identities together into permission groups and/or roles (on AWS) and attach IAM policies to them. We call such user aggregations ***data access groups***. Typically, each data access group carries a certain semantic meaning (e.g. web developers). A policy assigned to a data access group grants access to all cloud resources that individual identities in the group access to. Such an approach creates *over-privileged* identities by granting each identity access to cloud resources that are used by some identities in a group [Security, 2021; Verizon, 2021]. We call permissions that identities are granted, but never use, ***dormant permissions***. Dormant permissions violate the fundamental security principle - the *principle of least privilege* [Saltzer and Schroeder, 1975] that stipulates that every user and an automated service should operate using the least set of privileges necessary to complete the job. Dormant permissions allow attackers after compromising an identity's credentials to not only get access to cloud resources that identity actively uses, but also get access to additional resources that the compromised identity has access to because of dormant permissions.

IAM policies are not necessarily poorly designed from the beginning - they are likely to deteriorate over time as organization evolves (e.g. new automated services get developed, employees move between teams, etc). Policies rarely get updated to reduce the number of dormant permissions, thus causing the set of dormant permissions to expand over time which gives an advantage to attackers.

[1]https://github.com/mikhail247/IAMAX

"Statement": [{
"Effect": "Allow",
"Action": "s3:ListObjects",
"Resource": "arn:aws:s3:::bucket-name" }]

Figure 1: An example of an AWS IAM policy

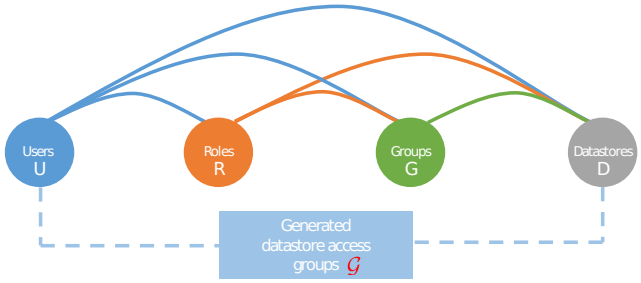


Figure 2: Schematic representation of the way users access datastores in AWS cloud. For simplicity we refer to both Roles and Groups as existing data access groups.

Our approach. We minimize the amount of dormant permissions to reduce damage caused by potential attacks. We intentionally refrain from considering a trivial solution - block a user from accessing unused datastores. Though such an approach may work, but in practice it is not being used because it would be hard for IT admins to maintain up-to-date per-user policies (specific accesses that should be allowed/blocked per user) especially when users transition between teams within an organization.

Figure 2 shows the schema denoting how users are partitioned into *roles* and *groups* and how access to different datastores is determined based on permissions granted to roles, groups and users themselves. Our approach to minimizing dormant permissions is to create a set of additional *datastore access groups* $\mathcal{G}$ (shown in figure 2). Solid edges represent existing IAM policies, while the dashed edges represent generated IAM policies. Conceptually, dashed edges are supposed to replace solid edges and thus remove dormant permissions. Both types of edges depict mapping of users to datastore access groups, which are named as roles and groups by cloud platforms, and mapping of those groups to datastores. Using CP, we optimize: (1) partitioning of users into different generated groups $g \in \mathcal{G}$, and (2) what datastores are accessible by each group $g \in \mathcal{G}$. From an implementation perspective, the policy generated by the CP solver is concatenated with original over-permissive policies via *logical AND* operation using cloud IAM policy management tools, thereby reducing dormant permissions. We did not edit existing IAM policies to avoid introducing unintentional errors due to high complexity of the policy language.

3 Optimizing IAM Policies

We start by introducing different objects and relationship among them (as depicted in Figure 2). These objects and relationships will define the variables, constraints and objective of the IAM policy optimization problem. Set U denotes the set of users; D denotes the set of datastores; G denotes the set of existing data access groups; set $\mathcal{G}$ denotes the generated access groups as noted in Section 2. Set T denotes data types in a datastore according to the organization's data classification scheme (e.g, social security number, email, credit card number among others). Real world data sets used in our experiments contain at most 10 data types. Such objects within an organization are extracted from mining its cloud infrastructure. The number of datastore access groups ($|\mathcal{G}|$) is a configurable parameter that intuitively defines the flexibility we have in reconfiguring the IAM policies (will be discussed later). We deliberately avoid making the number of datastore access groups ($|\mathcal{G}|$) to be a variable to reduce model complexity and thus increase scalability.

Users and existing groups. As noted in Section 2, users are end users (either humans or services) that need to access different datastores. Users may directly access a datastore, however, usually they either assume a role or inherit permissions from a permission group. Original IAM policies include 13–150 users, 90–1792 roles, and 5–833 permission groups. Without loss of generality, we denote both roles and permission groups as 'existing data access groups' G. Thus, users are mapped to different groups G, which are mapped to datastores. We mine all such links, and create following relationships.

Users and datastores. Based on existing permissions (users-roles-groups-datastores), we can also compute *all* the datastores $d \in D$ that a user $u \in U$ can potentially access. This is stored in variable $\widehat{\mathrm{UD}}(u,d) \in \{0,1\}$. These variables are constants as they are based on pre-defined permissions in an organization's cloud.

Users and accessed datastores. Using an organization's historical data, we can determine which datastores have been accessed by different users. Let $\mathrm{UD}(u,d) \in \{0,1\}$ denote if user $u \in U$ has accessed datastore $d \in D$ in the past. We shall require any refactored IAM policy to preserve access of different users to the datastores they have accessed in the past.

Groups and datastores. Let $\mathrm{GD}(g,d) \in \{0,1\}$ denote if group $g \in G$ provides access to the datastore $d \in D$. This is a constant mapping.

Users and datastore access groups. We create $|\mathcal{G}|$ datastore access profiles that allow us to further control user access to datastores. Let variable $\mathrm{U}\mathcal{G}(u,\tilde{\mathrm{g}}) \in \{0,1\}$ denote whether user $u \in U$ belongs to the access group $\tilde{\mathrm{g}} \in \mathcal{G}$.

Datastore access groups and datastores. Let variable $\mathcal{G}\mathrm{D}(\tilde{\mathrm{g}},d) \in \{0,1\}$ denote whether datastore access group $\tilde{\mathrm{g}} \in \mathcal{G}$ provides access to the datastore $d \in D$.

Datastores and data types. Let $\mathrm{DT}(d,t) \in \{0,1\}$ denote whether datastore d contains data of type $t \in T$. This is a constant mapping.

Constraints. We now describe the relationship between different variables that define the IAM policy optimization problem. We next show for a given relationship among users, groups and datastore access groups, how to compute how many total datastores a user u can access, and based on that compute *redundant* datastores a user has access to. These redundant permissions constitute the so-called *dormant permissions* that we want to minimize.

We create a variable $\widetilde{\mathrm{UD}}(u,d) \in \{0,1\}$ to denote if user u can access datastore d *after* we incorporate additional permissions and relations from the datastore access groups $\mathcal{G}$. Constraints on it are the following:

Frequently accessed datastores must still be accessible.

$$\widetilde{\mathrm{UD}}(u,d) \geq \mathrm{UD}(u,d)\ \forall u \in U,\ d \in D$$

Computing $\widetilde{\mathrm{UD}}(u,d)$. A user u can only access datastore d iff the following conditions hold:

- User u is part of a datastore access group $\tilde{\mathrm{g}}$ and $\tilde{\mathrm{g}}$ has permission to access the datastore d ***AND***
- The current permissions based on user-role-groups-datastore also allow user u to access d

$$\widetilde{\mathrm{UD}}(u,d) = \mathrm{OR}\Big(\mathrm{U}\mathcal{G}(u,\tilde{\mathrm{g}}) \bigwedge \mathcal{G}\mathrm{D}(\tilde{\mathrm{g}},d) \forall\, \tilde{\mathrm{g}} \in \mathcal{G}\Big) \bigwedge \widehat{\mathrm{UD}}(u,d)$$

where OR stands for the logical *or* operator.

Data type constraints. A data type constraint stipulates that a user should not get an access to a completely new data type that they never worked before with. We make an assumption that a user accesses all data types that are in the datastore d (if the user is allowed to access d). More fine-grained formulation of this constraint would require operating at an object level (e.g. directory/table level in a datastore), but this would significantly increase the number of variables in the problem and make it computationally intractable.

The data type constraint can be formulated as:

$$\mathrm{OR}\Big(\mathrm{UD}(u,d) \bigwedge \mathrm{DT}(d,t)\ \forall d \in D\Big) \geq$$
$$\mathrm{OR}\Big(\widetilde{\mathrm{UD}}(u,d) \bigwedge \mathrm{DT}(d,t)\ \forall d \in D\Big) \forall u \in U, t \in T \quad (1)$$

3.1 Constraint Program for Reducing Dormant Permissions

Based on the variables and relationships among them, we now describe the constraint program (CP) that optimizes IAM policies by minimizing the dormant permissions. Formally, the reduction in the number of user-datastore permissions in the refactored IAM policy is:

$$\sum_{u,d} \widehat{\mathrm{UD}}(u,d) - \sum_{u,d} \widetilde{\mathrm{UD}}(u,d)) \quad (2)$$

Maximizing the above objective would minimize the second term in the objective (first term is constant based on the existing IAM policy). Therefore, users in the refactored policy would have access to as few datastores as necessary given the problem constraints, and number of access groups $|\mathcal{G}|$. Thus, it will also reduce dormant permissions, and result in a *least-privilege* IAM policy.

$$\max \Big(\sum_{u,d} \widehat{\mathrm{UD}}(u,d) - \sum_{u,d} \widetilde{\mathrm{UD}}(u,d) \Big) - \sum_{u} \psi_u \quad (5)$$

$$\widetilde{\mathrm{UD}}(u,d) \geq \mathrm{UD}(u,d)\ \forall u \in U,\ d \in D \quad (6)$$

$$\widehat{\mathrm{UD}}(u,d) = \mathrm{OR}\Big(\mathrm{U}\mathcal{G}(u,\tilde{\mathrm{g}}) \bigwedge \mathcal{G}\mathrm{D}(\tilde{\mathrm{g}},d)\ \forall \tilde{\mathrm{g}} \in \mathcal{G} \Big) \bigwedge \widehat{\mathrm{UD}}(u,d)\ \forall u \in U,\ d \in D \quad (7)$$

$$\mathrm{OR}\Big(\mathrm{UD}(u,d) \bigwedge \mathrm{DT}(d,t)\ \forall d \in D \Big) \geq \mathrm{OR}\Big(\widetilde{\mathrm{UD}}(u,d) \bigwedge \mathrm{DT}(d,t)\ \forall d \in D \Big) \forall u \in U, t \in T \quad (8)$$

$$v_u = \sum_{d} \widetilde{\mathrm{UD}}(u,d) - (1.0+\epsilon)\,\mathrm{UD}(u,d) \forall u \in U \quad (9)$$

$$\psi_u = \max(v_u, \gamma v_u) \forall u \in U \quad (10)$$

$$\mathcal{G}\mathrm{D}(\tilde{\mathrm{g}},d), \mathrm{U}\mathcal{G}(u,\tilde{\mathrm{g}}) \in \{0,1\}\ \forall u, \tilde{\mathrm{g}}, d \quad (11)$$

Table 1: Constraint program for minimizing dormant permissions

Penalty on dormant permissions per user. The objective (2) minimizes the total number of dormant permissions in the system. However, we also want to limit the maximum number of dormant permissions per user for increased robustness to attacks. We formulate such constrains as a soft constraints, violation of which incurs a penalty of $\psi_i(v)$ computed as below:

$$v_u = \sum_{d} \widetilde{\mathrm{UD}}(u,d) - (1.0+\epsilon)\,\mathrm{UD}(u,d) \forall u \in U \quad (3)$$

$$\psi_u = \max(v_u, \gamma v_u) \forall u \in U \quad (4)$$

where ϵ is a hyper-parameter in our model, which should be tuned by a domain expert according to the amount of risk they are willing to take. If the parameter is set to 0, then our CP formulation will softly penalize any user that has non-zero amount of dormant permissions. Hence, if $\epsilon > 0$, then the model does not penalize users with less than ϵ fraction of dormant permissions. Based on the empirical analysis of IAM policies, we believe that allowing a small fraction of dormant permissions (e.g. 15%) increases policy interpretability and is unlikely to significantly affect security aspect of the problem. The parameter γ penalizes such soft constraint violations more harshly if it is more than 1. We can also put per user dormant permissions as constraints, however in practice it made the program infeasible. The overall CP formulation is given in Table 1.

3.2 Group Homogeneity Constraints

We further enhance the basic CP formulation (Table 1) by including group homogeneity constraints. Intuitively, all users in a group $\tilde{\mathrm{g}} \in \mathcal{G}$ should be *similar* w.r.t. a job role in an organization to make the solution more explainable and interpretable to IT admins. Moreover, heterogeneous groups may increase an impact of a potential attack: similar users are usually susceptible to the same attack vector (e.g. phishing, waterhole, etc). If they are spread across multiple groups, then attackers may get additional advantage if the number of residual dormant permissions across all those groups is larger than in the case when similar users are grouped together.

We assume a precomputed pairwise function α_{user} defined on UxU denoting dissimilarity between users. Section 4 shows how this function can be computed using graph neural networks (GNNs) and the data that we mine from an organization's cloud infrastructure. We add the homogeneity constraints as below. If a user u is mapped to datastore access group $\tilde{\mathrm{g}}$ (i.e., $\mathrm{U}\mathcal{G}(u,\tilde{\mathrm{g}}) = 1$), then we use the shorthand $u \in \tilde{\mathrm{g}}$ for exposition clarity.

$$\max \big\{ \alpha_{user}(u_1,u_2) \forall u_1, u_2 \in \tilde{\mathrm{g}} \big\} \leq \alpha, \forall\, \tilde{\mathrm{g}} \in \mathcal{G} \quad (12)$$

where α is a diversity threshold that we show in Section 4 how to compute from the data.

Constraint generation. Constraints (12) create memory issues when incorporated in a single program as quadratic number of terms are there in the number of users in a group $\tilde{\mathrm{g}}$. Also, they often make the CP problem infeasible. Therefore, we follow a constraint generation approach [Ben-Ameur and Neto, 2006] where we first solve the program in Table 1 with no homogeneity constraints. We then follow an iterative constraint generation procedure to add back most violated homogeneity constraints and solve the program again.

Separation oracle. At every iteration for each datastore access group $\tilde{\mathrm{g}} \in \mathcal{G}$, we identify user pairs (u_1,u_2) s.t. $\mathrm{U}\mathcal{G}(u_1,\tilde{\mathrm{g}}) = \mathrm{U}\mathcal{G}(u_2,\tilde{\mathrm{g}}) = 1$, and $\alpha_{user}(u_1,u_2) > \alpha$. We then add the constraint $\mathrm{U}\mathcal{G}(u_1,\tilde{\mathrm{g}}) + \mathrm{U}\mathcal{G}(u_2,\tilde{\mathrm{g}}) \leq 1 \quad \forall\, \tilde{\mathrm{g}} \in \mathcal{G}$ to the CP and solve again. This process continues until all the homogeneity constraints are satisfied or we get an infeasible program, in which case we output the solution generated in the last iteration. It may happen that no solution may exist that satisfies all the homogeneity constraints (based on the threshold α). In such a case, the constraint generation approach provides the solution with most violated constraints being satisfied.

4 Graph Representation Learning

Incorporating GNN into CP. Interpretable policies require group homogeneity with respect to users' job roles. However, such information is not shared with us due to high business sensitivity. Group homogeneity constraints are defined in terms of the user dissimilarity function α_{user} and the diversity threshold α (Section 3.2). We use a graph neural network (GNN) to approximate both parameters and share them with the CP solver.

User behavioral graph (UBG). Intuitively, job roles can be inferred from users' interaction with a cloud environment (e.g. executed operations, accessed cloud resources). We represent records of all executed cloud operations over the 6-12 months period as a weighted heterogeneous graph.

- **Nodes, set $\mathcal{V}$**: users, roles, groups, datastores. Each node has its own handcrafted features. For example, a datastore includes distribution of data types, each user node has an associated risk score.
- **Edges**: a pair of nodes is connected with an undirected edge, if they have participated in the execution of a cloud

operation (e.g. reading data, modifying resource configuration, etc) and its weight, $e_{v,u,\tau}$, is proportional to the frequency of such an operation.

- **Edge types, set** $\mathcal{R}$ carry semantic meaning of executed operations: data flow edges (e.g. transferring data), configuration update edges (e.g. updating configuration of cloud resources) and others.

Graph neural network. We adapted Relational GCN [Schlichtkrull *et al.*, 2018] approach to heterogeneous graphs, however, we replaced GCN [Kipf and Welling, 2017] modules with GraphSage [Hamilton *et al.*, 2017] modules (one per each graph relation type) to achieve *inductive* graph representation learning. Our GNN-based embeddings are quite general, we successfully use them for multiple downstream tasks such as anomaly detection and data visualization.

Conceptually, our GNN is a superposition of multiple *weighted* GraphSage neural networks where each network operates on a specific relation type $\tau \in \mathcal{R}$ (Eq. (13)–(15)). At each search depth (parameter k) nodes aggregate information from their local neighbors, $\mathcal{N}_\tau(v)$, using the *weighted mean* aggregator, into the vector $\mathbf{h}^k_{\mathcal{N}_\tau(v)}$, where weights are normalized edge frequencies $e_{v,u,\tau}$ (Eq. (13)). After that, each node embedding $\mathbf{h}^k_{v,\tau}$ gets updated according to the rule (14), where $\mathbf{W}^k_\tau$ are trainable weight matrices and $\oplus$ stands for concatenation of two vectors. The final node embedding $\mathbf{h}^k_v$ at the search depth k (Eq. (16)) is a mean value of normalized relation-specific embedding vectors $\mathbf{h}^k_{v,\tau}$ (Eq. (15)).

The search depth is a design parameter and if it is too high, then a GNN may suffer from over-smoothing. Thus, we set it to 2 because it is sufficient for our experiments. We use 50-dimensional $\mathbf{h}^2_v$ (Eq. (16)) vectors as node embeddings in our experiments. Note that $\mathbf{h}^0_v$ vectors in the Eq. (13), (14) are initialized by corresponding node feature vectors. We train our GNN on link prediction task with a cosine distance between node embeddings for 500 epochs.

$$\mathbf{h}^k_{\mathcal{N}_\tau(v)} \leftarrow mean_{e_{v,u,\tau}}(\mathbf{h}^{k-1}_{u,\tau} \quad \forall u \in \mathcal{N}_\tau(v)) \tag{13}$$

$$\mathbf{h}^k_{v,\tau} \leftarrow \sigma(\mathbf{W}^k_\tau \cdot (\mathbf{h}^{k-1}_{v,\tau} \oplus \mathbf{h}^k_{\mathcal{N}_\tau(v)})) \quad \forall v \in \mathcal{V}\ \forall \tau \in \mathcal{R} \tag{14}$$

$$\mathbf{h}^k_{v,\tau} \leftarrow \mathbf{h}^k_{v,\tau} / ||\mathbf{h}^k_{v,\tau}||_2 \quad \forall v \in \mathcal{V} \tag{15}$$

$$\mathbf{h}^k_v \leftarrow mean(\mathbf{h}^k_{v,\tau} \quad \forall \tau \in \mathcal{R}) \tag{16}$$

User embeddings to CP parameters. After training is complete we extract user embeddings and cluster them with k-means algorithm. A grid search with k in range of [5, 25] gives us an optimal value of k, k_{opt}, that corresponds to a point of maximum curvature ('knee' point) of the function that maps k to k-means' objective value (sum of squared distances of samples to their closest cluster center) [Satopaa *et al.*, 2011]. The range of k encodes our domain-specific knowledge - we expect to identify between 5 and 25 sufficiently different job roles at an organization. Figure 3a visualizes 15 clusters detected in one of the data sets. We shared clustering results with organizations and received back a confirmation of good approximation quality of employees' job roles.

Algorithm 1 IAMAX: high-level description

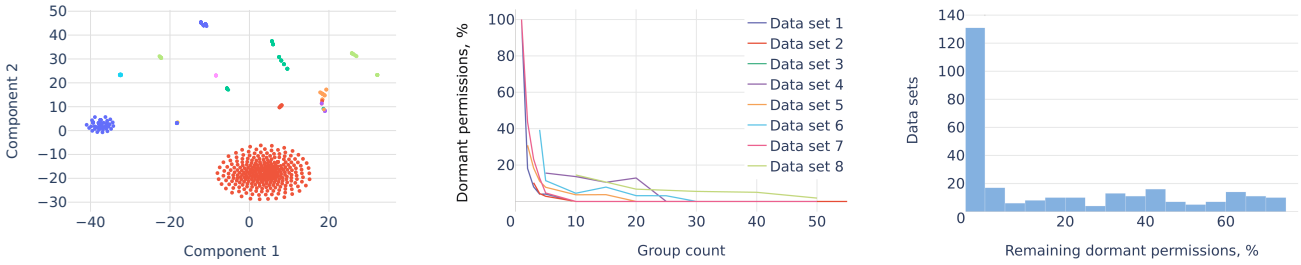
1: **Input:** $org's\ cloud\ configuration$, $|\mathcal{G}|$ ▷ Section 3
2: **Output:** $Hardened\ IAM\ policies$
3:
4: $Train\ Graph\ Neural\ Network\ (GNN)$ ▷ Section 4
5: $\alpha_{user}(u_1, u_2), user\ cluster\ assignment \leftarrow GNN$
6: $CP \leftarrow CP_{basic}$ ▷ Table 1
7: $solution \leftarrow solve(CP)$
8:
9: **if** $solution = \emptyset$ **then**
10: **return** *Infeasible CP*
11: **end if**
12:
13: **repeat** ▷ Section 3, "Constraint generation"
14: **for all** $\tilde{\mathrm{g}} \in \mathcal{G}$ **do**
15: *# Identify dissimilar user pair* (u_1, u_2)
16: $(u_1, u_2)\ s.t.\ u_1, u_2 \in \tilde{\mathrm{g}}\ and\ \alpha_{user}(u_1, u_2) > \alpha$
17: *# Add additional constraints*
18: $CP \cup \{\mathrm{U}\mathcal{G}(u_1, g\prime) + \mathrm{U}\mathcal{G}(u_2, g\prime) \leq 1\ \forall g\prime \in \mathcal{G}\}$
19: **end for**
20: $solution \leftarrow solve(CP)$
21: **until** $solution \neq \emptyset$
22:
23: *Hardened IAM poclicies ← extract(last feasible solution)*

GNN embeddings let us define user dissimilarity function $\alpha_{user}(u_1, u_2)$ consistently with GNN design: it is the cosine distance between embeddings of users u_1 and u_2. The threshold α is the maximal cluster diameter computed as the cosine distance between the most dissimilar users within any cluster when setting the parameter k to k_{opt}. Significant entropy reduction (Section 7.2, Figure 4b) of data access groups after iteratively adding homogeneity constraints empirically validates the choice of the parameter α. If α was too high, then we would not observe entropy reduction.

5 Algorithm

In this section we turn concepts outlined in Sections 3, 4 into an imperative Algorithm 1 to clarify IAMAX's workflow. After IAMAX gets deployed in an organization's cloud environment, it starts mining cloud configuration to convert it into a graph representation. Hence, those details lie outside the scope of our paper. Having computed the graph representation, IAMAX trains a GNN (Section 4), which is used to generate graph node embeddings and to cluster users (Algorithm 1, lines 4,5). At this point, IAMAX can proceed with solving the basic CP problem outlined in the Table 1 (Algorithm 1, lines 6,7). The basic CP problem may turn out to be infeasible (Algorithm 1, lines 9–11) if the specified number of datastore access groups is insufficient (input parameter $|\mathcal{G}|$).

If IAMAX can find a feasible solution to the basic CP problem (Algorithm 1, line 7), then it proceeds with iterative constraint generation (Algorithm 1, lines 13–21). This process continues until all user dissimilarity constraints get satisfied or the CP problem turns infeasible at some iteration. In practice, constraint generation usually stops due to encountering



(a) (b) (c)

Figure 3: (*a*) T-SNE projection of 50-dimensional Graph-NN identity embeddings (data set 7); 15 distinct clusters corresponding to different job roles are highlighted. (*b*) Relative percentage of remaining dormant permissions after IAM policy optimization as a function of the number of generated data access groups. IAMAX significantly reduces the amount of dormant permissions and often reaches 0% level with a small number of datastore access groups. For group count more than 50, there are no dormant permissions, therefore not shown in the plot. (*c*) To verify the generalizability of our approach, we evaluate IAMAX on 280 synthetic data sets. In most cases, IAMAX achieves significant reduction of dormant permissions.

an infeasible CP problem. Also, at each iteration we add mini-batches of constraints rather than individual constraints to amortize time needed for running the CP solver. When the execution reaches the end (Algorithm 1, line 23), it is guaranteed that the feasible solution has been found at either the final iteration or the one before that. Finally, the CP solution gets decoded into new IAM policies that are concatenated with existing ones using *logical* AND operation to reduce the amount of *dormant* permissions.

6 Attack Models

Besides reducing the dormant permissions, we now describe different attack models that further test our optimized IAM policies. In *random* attack model, we assume that an attacker does not have information about the internal IAM configuration of an organization. Therefore, the attacker randomly tries to compromise k users. Once compromising k users, the attacker gets access to all the datastores that can be accessible by any of the k users.

In the *worst-case* attack model, we assume that an attacker has full observability of an organization's IAM policies. Therefore, the attacker carefully plans to attack k users such that it can maximize the blast radius (or the number of datastores it can access). We show that this problem is NP-Hard; also, it is *monotone* and *submodular*, thus a simple greedy approach provides a constant factor approximation.

6.1 Formalization of the Worst-case Attack Problem

The proof of NP-hardness relies on reduction from the set cover problem. Suppose we have an instance of the Set Cover problem, where the sets are X_1 to X_m. Each set X_i is composed of elements u_{i1} to u_{in_i} where n_i denotes the number of elements in set X_i. Let the total number of elements in the universe be n. We create an instance of IAM problem as next.

- We create m users corresponding to each set X_1 to X_m.
- We create n datastores, one for each element of the universe.
- If set X_i contains element j, we allow user X_i access to the datastore j.

We now solve the worst-case attack problem on this datastore access graph by choosing k users to compromise. If the number of datastores accessible is the same as the universe, then the set cover problem has a solution with k-cover, otherwise not.

To properly simulate the worst-case attack, which is NP-hard, we have to uncover its submodular nature and use an appropriate approximation.

Monotonicity. Let S be the set of users being compromised. Let $f(S)$ denotes the number of datastores that become accessible as a result. The function f is monotone. Let $u \notin S$ be another user. $f(S \cup \{u\}) \geq f(S)$ as the number of accessible datastores cannot decrease if an additional user is compromised.

Submodularity. The function f is also submodular. Let $S' \subseteq S$, and let u be a user not in S. Then, we must show:

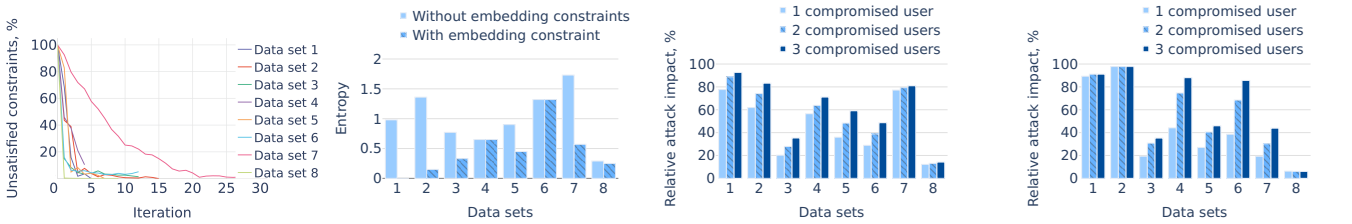
$$f(S \cup \{u\}) - f(S) \leq f(S' \cup \{u\}) - f(S') \qquad (17)$$

Let us consider the expression $f(S \cup \{u\}) - f(S)$. It will count those datastores that can only be accessible by user u and none of the users in the set S. Let us denote this number as $n_{u|S}$; $n_{u|S'}$ is defined analogously. We must have $n_{u|S} \leq n_{u|S'}$ because $S' \subseteq S$. This is becuase the additional access to datastores granted by the user u over datastores accessible by S must be smaller than the additional access granted by the user u over datastores accessible by S'.

Constant factor approximation. The standard greedy algorithm that iteratively selects the user u that provides the maximum marginal gain in terms of the function f is guaranteed to provide $(1 - 1/e)$-approximation [Nemhauser *et al.*, 1978]. Using this approach, we can approximately select k users to compromise for simulating the worst-case attack.

7 Experimental Setup

We evaluate IAMAX on both synthetic and real world instances. Each real world instance represents cloud infrastruc-



(a) (b) (c) (d)

Figure 4: (*a*) Cumulative fraction of satisfied user embedding constraints at each iteration. (*b*) Entropy of generated solutions before and after adding user embedding constraints. (*c, d*) Attack impact (lower better) on an organization under an assumption of 1-3 compromised users. (*c*) The average-case impact considering an attacker possesses incomplete information about IAM policies. (*d*) The worst-case impact considering an attacker possesses perfect information about IAM policies.

Graph	Users	Data stores	Dynamic edges	Permission edges
1	13	56	513	1969
2	32	89	1192	16791
3	39	341	1602	28880
4	57	572	7153	37854
5	60	88	757	4115
6	64	258	2418	72272
7	112	163	2789	4025
8	150	600	3095	11517

Table 2: Properties of real world graph instances

ture within one or more departments at a commercial organization. In addition to 8 real world data sets, we generated 280 synthetic data sets using real data sets as baselines. Throughout the paper we mostly focus on real world data sets and use synthetic data only for evaluation of statistical effectiveness of the proposed approach. Independent security experts verified correctness and interpretability of generated IAM policies. Moreover, generated policies are correct by design (Section 3). The section is organized as follows: first we show effectiveness of our basic CP formulation, then we demonstrate the effect of adding constraints based on GNN embeddings, and finally we conclude with attack simulations.

CP Versus MILP. Even though our CP formulation can be solved with an MILP solver after being linearized, we used IBM CP solver [IBM, 2021] because linearization of non-linear functions (e.g. max, logical operators, etc) introduces a large number of additional variables, thus making the problem unsolvable in a practical amount of time. In the case of the largest instance (instance #8) linearization increases the number of variables by $\sim$32 times and the number of constraints by $\sim$9 times. Specifically, MILP formulation contains 3,900,750 variables and 4,280,130 constraints vs 120,000 variables and 472,801 constraints in the CP case (not counting homogeneity constraints). As a consequence, IBM MILP solver failed to find any feasible solution within 2.5 hours. Across all 8 real data sets, the size of CP problem lies within the range of 2,108–196,000 variables and 6,528–300,000 constraints (after incorporating homogeneity constraints). IAMAX is designed to be used as a decision support tool by system admins, thus we prioritize fast solving time by setting the time limit to 15 minutes. All experiments were conducted on AWS c5.24xlarge virtual machine equipped with 96 vCPU and 192 GB of RAM.

Real world graphs. Real world graphs were shared by IT departments of 8 commercial organizations (Table 2). Graphs are very sparse as we would expect in a security setting - only certain accesses (permission edges) are allowed. However, most of them remain unused: the number of *dynamic edges* that represent actual data accesses is even smaller. Densities of graphs built on permission and dynamic edges differ by 1.4 - 29.9 times. High coefficients correspond to poorly designed (over-priviledged) security models. The number of users varies between 13 and 150, while the number of datastore nodes lies within the range of 56 - 600.

Synthetic graphs. We use real graphs as baselines to generate 280 synthetic graphs. We vary the number of nodes, but keep graph density as is, i.e. in the range of 0.259 ± 0.198 (avg $\pm$ std). To generate a synthetic graph, we first sample the number of users and datastores from uniform distributions over the following intervals [10, 150] and [50, 300] respectively that cover variations of those parameters across real graphs. We deliberately set the maximum number of data stores fewer than 600 (instance 8) to speed up computations. After fixing node counts we sample with replacement the actual nodes from a real world graph, which is chosen at random. Then we add Gaussian $N(0, 0.01)$ noise to node embeddings and renormalize them. To match the graph density with the density of the underlying baseline we sample edges from a multinomial distribution, where each component is proportional to the cosine distance between a user and a datastore embeddings. Also we enforce the invariant that dynamic edges are always a subset of all permission edges. A synthetic graph generated in such a way is an "upsampled" version of an underlying real world graph.

7.1 Reduction of Dormant Permissions

Real world instances. IAMAX significantly reduces *dormant* permissions (Figure 3b) over the current IAM policy of companies. For this purpose we vary the number of datastore access groups $|\mathcal{G}|$ from 1 up to 100 with an increment of 5. The solver's time limit is fixed at 15 minutes. When the number of datastore access group is too small, the problem often becomes infeasible. For most data sets, the fraction

of remaining *dormant* permissions quickly goes down to 0% as we increase the number of datastore access groups. The only exception is the instance 8 (the largest data set), which requires 50 data access groups. According to these results we set the number of data access groups to 20 for instances 1-7, and 40 for the instance 8 in other experiments. We also highlight that it is important to have as few as possible datastore access groups to keep IAM policy interpretable. Our results show that even with additional 5 groups, dormant permissions reduce significantly over the existing IAM policies.

Synthetic experiments. To evaluate IAMAX across larger number of instances, we use 280 synthetic graphs. Figure 3c shows unnormalized distribution of the remaining dormant permissions in a graph while running the solver for at most 15 minutes, and fixing the datastore access groups at 20. Each bin spans 5% interval, graphs with no *dormant* permissions (0%) fall into the very first bin. The number above a bar denotes how many instances fall in the corresponding interval. The histogram is skewed towards the left-hand side: in 46.7% of cases IAMAX reduces dormant permissions down to 0%. However, we observe some outliers where the solver is unable to reach dormant permissions 0% level. This mostly happens because of either exceeding the time limit or the need for a larger number of data access groups due to instance complexity. These results can be improved by setting a higher time limit for the CP solver or increasing the number of data access groups.

7.2 User Embedding Constraints

To make generated IAM policies interpretable we follow constraint generation approach outlined in Section 3.2, which leads to generating mostly homogeneous data access groups.

Figure 4a shows the fraction of unsatisfied embedding constraints at each iteration when setting the number of data access groups to 20. Depending on the data set, it takes 10-20 iterations to satisfy more than 95% of embedding constraints. To verify that such an approach produces mostly homogeneous groups, we compare the entropy of generated data access groups with respect to users' cluster assignment (Figure 4b) before and after adding embedding constraints. In all cases except the data sets 4 and 6, embedding constraints drastically reduce group entropy. In the case of the data sets 4 and 6, the problem becomes infeasible at the very first constraint relaxation iteration.

7.3 Simulated Security Attacks

We consider two security attacks that fundamentally differ in terms of knowledge that an attacker posses. In both cases an attacker tries to compromise k users, where $k \in \{1, 2, 3\}$. An attack's impact is the number of datastores that an attacker can get access to. Hence, we report the relative attack impact, which is the the ratio between the number of compromised datastores after applying IAMAX and the number of compromised datastores in the existing cloud infrastructure. The lower ratio is, the more noticeable effect IAMAX has on the IAM policy optimization. We notice that IAMAX's effect can be masked by high-degree user nodes, especially, in the worst-case attack because the greedy algorithm keeps selecting such nodes. High-degree user nodes impose a severe security risk on an organization and this issue should be mitigated using traditional software engineering methods - splitting nodes into multiple nodes of lower degrees. To evaluate IAMAX in the worst-case attack scenario, we removed top-30% of high-degree nodes.

If an attacker has no information about the implemented IAM policies, then we can estimate the average impact of compromising k user identities (Figure 4c). For example, in the case of the instance 8 IAMAX achieves the highest impact reduction (86% - 88%) even though it is the largest real world data set. However, impact reduction is minimal for the instance 1 (8% - 22%) due to the presence of high-degree user nodes.

If an attacker has a perfect knowledge of the IAM policies, then the attacker can cast the problem to max k-cover and solve it using a greedy approximation algorithm. Figure 4d illustrates such a case study. IAMAX minimizes the worst case attack impact for organizations 3, 5, and especially 8 by 41% - 89%. However, organizations 1 and 2 remain almost unaffected due to the large number of high-degree nodes remaining in the data set after removing the top-30% of user nodes sorted by their degrees.

Our current simulated attacks are primarily based on the interaction patterns between users and datastores, and the observations available to an attacker. We note that using the rich language of constraint programming, we can also generate *automated* attacks that achieve malicious goals while complying with IAM policies. We leave this as part of the future work.

8 Conclusion and Future Work

Given the increasing popularity of cloud computing, associated security issues are also increasing in severity. We developed a principled approach for the key problem of optimizing IAM policies to reduce the attack surface of an organization' cloud setup (or the so-called *dormant* permissions that allow users access datastores which are not needed for users' business functions). We presented a formulation of IAM policy optimization using constraint programming and graph representation learning, identified key constraints which IAM policies should satisfy, and then tested the resulting IAM policies on 8 real world and multiple synthetic data sets. Our results show that IAMAX is highly effective in reducing *dormant* permissions and generating interpretable IAM policies. Our framework also opens the door to the application of a host of AI methods to address additional security problems in cloud infrastructure.

Several promising opportunities exist for enriching IAM policy generation with more complex constraint types. For example, temporal constraints that define the order of datastore accesses and set constraints that define what subset of automated services can simultaneously access a given resource can significantly limit the range of attackers' actions after they compromise a cloud environment. To formulate such constraints, we can use program invariants inferred during the program analysis of automated services. Moreover, violations of such constraints at runtime can also be used for anomaly detection.

References

[Axelsson, 1999] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *CCS*, 1999.

[Ben-Ameur and Neto, 2006] Walid Ben-Ameur and José Neto. A constraint generation algorithm for large scale linear programs using multiple-points separation. *Mathematical Programming*, 2006.

[Canali *et al.*, 2012] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA 2012*, 2012.

[Demme *et al.*, 2013] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ISCA*, 2013.

[Hamilton *et al.*, 2017] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, volume 30, pages 1025–1035, 2017.

[Handley *et al.*, 2001] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *USENIX*, 2001.

[Hassan *et al.*, 2019] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *NDSS*, 2019.

[Hoffmann, 2015] Jörg Hoffmann. Simulated penetration testing: From "dijkstra" to "turing test++". In *ICAPS*, pages 364–372, 2015.

[Huang *et al.*, 2017] Cheng Huang, Shuang Hao, Luca Invernizzi, Jiayong Liu, Yong Fang, Christopher Kruegel, and Giovanni Vigna. Gossip: Automatically identifying malicious domains from mailing list discussions. In *AsiaCCS*, 2017.

[IAM, 2021] AWS IAM. Policies and permissions in iam. https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies.html, 2021. Accessed: 2022-05-29.

[IBM, 2021] IBM. Cp optimizer user's manual. https://www.ibm.com/docs/en/SSSA5P_12.8.0/ilog.odms.studio.help/pdf/usrcpoptimizer.pdf, 2021. Accessed: 2022-05-29.

[Kazdagli *et al.*, 2016] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *MICRO*, 2016.

[Kipf and Welling, 2017] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[Kuenzli, 2020] Stephen Kuenzli. Why are good aws security policies so difficult? https://www.k9security.io/posts/2020/06/why-are-good-aws-security-policies-so-difficult, 2020. Accessed: 2022-05-29.

[Lei *et al.*, 2019] Qi Lei, Lingfei Wu, Pin-Yu Chen, Alex Dimakis, Inderjit S. Dhillon, and Michael J Witbrock. Discrete adversarial attacks and submodular optimization with applications to text classification. In *MLSys*, pages 146–165, 2019.

[Liu *et al.*, 2021] Ninghao Liu, Mengnan Du, Ruocheng Guo, Huan Liu, and Xia Hu. Adversarial attacks and defenses: An interpretation perspective. *SIGKDD Explor. Newsl.*, 2021.

[Marks, 2021a] Gene Marks. 533 million facebook users' phone numbers and personal data have been leaked online. https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4, 2021. Accessed: 2022-05-29.

[Marks, 2021b] Gene Marks. A linkedin 'breach' exposes 92% of users. https://www.forbes.com/sites/quickerbettertech/2021/07/05/a-linkedin-breach-exposes-92-of-usersand-other-small-business-tech-news, 2021. Accessed: 2022-05-29.

[Mirsky *et al.*, 2018] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *NDSS*, 2018.

[Nemhauser *et al.*, 1978] G L Nemhauser, L A Wolsey, and M L Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 1978.

[Oprea *et al.*, 2018] Alina Oprea, Zhou Li, Robin Norris, and Kevin Bowers. Made: Security analytics for enterprise threat detection. In *ACSAC*, 2018.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., 2006.

[Saltzer and Schroeder, 1975] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[Sarraute *et al.*, 2012] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *AAAI*, 2012.

[Satopaa *et al.*, 2011] Ville Satopaa, Jeannie R. Albrecht, David E. Irwin, and Barath Raghavan. Finding a "kneedle" in a haystack: Detecting knee points in system behavior. In *IEEE ICDCS*, 2011.

[Schlichtkrull *et al.*, 2018] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, 2018.

[Scroxton, 2020] Alex Scroxton. Leaky aws s3 bucket once again at centre of data breach. https://www.computerweekly.com/news/252491842/Leaky-AWS-S3-bucket-once-again-at-centre-of-data-breach, 2020. Accessed: 2022-05-29.

[Security, 2021] Aqua Security. Cloud configuration risks exposed. https://www.aquasec.com/news/cloud-misconfigurations-on-the-rise-2021-cloud-security-report, 2021. Accessed: 2022-05-29.

[Shmaryahu *et al.*, 2018] Dorin Shmaryahu, Guy Shani, Jörg Hoffmann, and Marcel Steinmetz. Simulated penetration testing as contingent planning. In *ICAPS*, pages 241–249, 2018.

[Sommer and Paxson, 2010] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE S&P*, 2010.

[Verizon, 2021] Verizon. Data breach investigations report. https://www.verizon.com/business/resources/reports/2021/2021-data-breach-investigations-report.pdf?_ga=2.175409509.678363106.1639526451-2015926119.1639526451, 2021. Accessed: 2022-05-29.

[Wang *et al.*, 2019] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. Neural cleanse: Identifying and mitigating backdoor attacks in neural networks. In *IEEE S&P*, 2019.

[Wu *et al.*, 2012] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *AsiaJCIS*, 2012.