Fixing Privilege Escalations in Cloud Access Control with MaxSAT and Graph Neural Networks

Yang Hu*, Wenxi Wang*, Sarfraz Khurshid, Kenneth L. McMillan, Mohit Tiwari
huyang@utexas.edu, wenxiw@utexas.edu, khurshid@ece.utexas.edu, kenmcm@cs.utexas.edu, tiwari@austin.utexas.edu
The University of Texas at Austin
Austin, Texas, U.S.A.

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00167

***Abstract*—Identity and Access Management (IAM) is an access control service employed within cloud platforms. Customers must configure IAM to establish secure access control rules for their cloud organizations. However, IAM misconfigurations can be exploited to conduct Privilege Escalation (PE) attacks, resulting in significant financial losses. Consequently, addressing these PEs is crucial for improving security assurance for cloud customers. Nevertheless, the area of repairing IAM PEs due to IAM misconfigurations is relatively underexplored. To our knowledge, the only existing IAM repair tool called `IAM-Deescalate` focuses on a limited number of IAM PE patterns, indicating the potential for further enhancements.**

We propose a novel IAM Privilege Escalation Repair Engine called `IAMPERE` that efficiently generates an approximately minimal patch for repairing a broader range of IAM PEs. To achieve this, we first formulate the IAM repair problem into a MaxSAT problem. Despite the remarkable success of modern MaxSAT solvers, their scalability for solving complex repair problems remains a challenge due to the state explosion. To improve scalability, we employ deep learning to prune the search space. Specifically, we apply a carefully designed GNN model to generate an *intermediate patch* that is relatively small, but not necessarily minimal. We then apply a MaxSAT solver to search for a minimum repair within the space defined by the intermediate patch, as the final approximately minimum patch. Experimental results on both synthesized and real-world IAM misconfigurations show that, compared to `IAM-Deescalate`, `IAMPERE` repairs a significantly larger number of IAM misconfigurations with markedly smaller patch sizes.

***Index Terms*—Cloud Access Control, MaxSAT, Graph Neural Networks**

I. Introduction

IAM [1], or Identity and Access Management, is an access control service utilized within cloud platforms. Its purpose is to securely manage access to resources based on a customer-specified IAM configuration, which contains access control rules for their cloud organization. An IAM configuration comprises two components: entities (such as users, services like Amazon EC2 instances, and roles representing job functions, responsibilities, and privileges within an organization) and permissions which refer to privileges of performing operations. When a service request is made and an IAM configuration is provided, IAM can determine whether the request complies with or violates the configuration, subsequently deciding whether to grant or deny access.

Maintaining correct IAM configurations is essential for robust cloud access control, ensuring that only authorized users and applications can access sensitive data and resources. Incorrect IAM configurations, namely IAM *misconfigurations*, can result in severe security consequences, including data breaches, denial of service attacks, and resource hijacking [2]–[6], leading to substantial financial losses globally [7]. IAM PE [8]–[10] is a type of attack that targets cloud access control systems by exploiting vulnerabilities within IAM, allowing attackers to gain extra permissions for carrying out sensitive operations or accessing confidential data. A prevalent method for achieving IAM PE involves exploiting IAM misconfigurations, which enable attackers to modify the configuration and subsequently obtain additional sensitive permissions through the modified configuration, posing a significant risk to the organization's security posture.

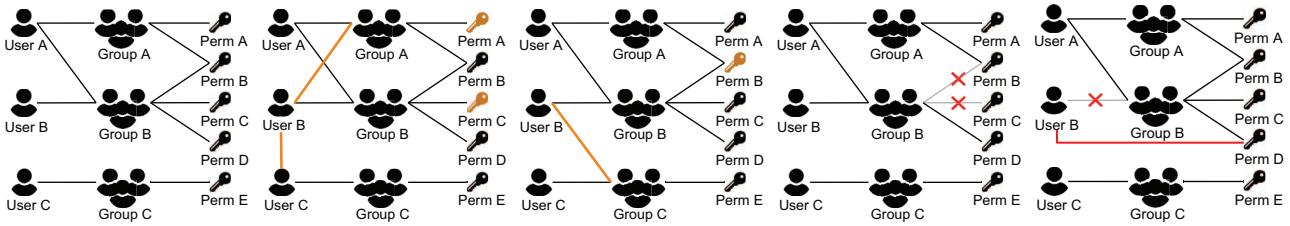
Over the past few years, there has been a notable surge in research on detecting and verifying IAM PEs [8], [11]–[16]. `TAC` [15], a recently proposed *permission flow graph* based IAM detector, focuses on detecting transitive IAM PEs where attackers can indirectly obtain sensitive permissions through any intermediate entities. Moreover, `TAC` proposes `IAMVulGen`, a generator for randomly generating IAM misconfigurations using a large entity and permission space with diverse types, manually identified from AWS official documentations [1], [17], [18] and studies [8]–[10], [19] on IAM PEs.

While there has been a considerable rise in research focused on detecting and verifying IAM misconfigurations, the area of repairing IAM misconfigurations remains relatively underexplored. To the best of our knowledge, `IAM-Deescalate` [20] is the only existing tool capable of repairing PEs in IAM misconfigurations. `IAM-Deescalate` utilizes the graph modeling approach of an IAM PE detector called `PMapper` [13], where each node in the graph represents a user/role, and each edge represents if a user/role can be authenticated as another user/role. A PE is represented as a path from a non-admin user to an admin user. Consequently, repairing PEs in an IAM misconfiguration is equivalent to breaking PE paths in the modeled graph.

`IAM-Deescalate` has four key limitations. First, its graph modeling primarily focuses on representing authentications between users and roles, rendering it incapable of repairing potential PEs through non-authentication strategies, such as changing the default version of IAM policies. Second, it overlooks the possibility of the transitive PEs where attackers can obtain sensitive permissions through entity types

* these authors contributed equally to this work.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00167



(a) IAM misconfiguration. (b) PE 1: transitive PE. (c) PE 2. (d) Valid repair. (e) Optimal repair.

Fig. 1: (a) shows an IAM misconfiguration example having three user entities, three user group entities, and five permissions including `Perm A` allowing to change the password of `User C`, `Perm B` allowing to add `User B` to `Group C`, and `Perm C` allowing to add `User B` to `Group A`. Each connection between a user entity and a user group entity represents that the user is in the user group. Note that users in a user group can obtain all permissions of the user group. Each connection between an entity and a permission represents that the permission is directly assigned to the entity. We assume that `User B` is controlled by an attacker and `Perm E` is the sensitive permission the attacker targets to obtain. (b) and (c) show two possible PEs for an attacker to obtain `Perm E`. (d) shows a valid but non-optimal repair example of the IAM misconfiguration changing the permission assignment of `User A`, which is unrelated to the two possible PEs. (e) shows an optimal repair example.

beyond admin users, such as services and non-admin users. Third, it does not support repair operations beyond revoking permissions from user/role, such as removing a user from a user group. Lastly, its repair objective is solely to identify a patch, rather than reducing the identified patch or even finding a minimum patch.

To address all of these limitations, we introduce a more comprehensive and efficient approach called `IAMPERE` for repairing a broader range of IAM PEs, using an approximately minimum patch containing two kinds of repair operations. Drawing inspiration from `TAC`, we initially model IAM misconfigurations as a reduced permission flow graph (PFG), which takes into account the entities and direct/indirect permission assignments in IAM configurations from a more general and concise perspective. Leveraging the simplified PFG based IAM modeling, we encode the repair problem into a MaxSAT problem via model checking technique.

Despite the remarkable success of modern MaxSAT solvers, their scalability for solving complex repair problems remains a challenge due to the state explosion. To enhance the scalability of MaxSAT solving, we utilize GNN to prune the search space for the solver. Specifically, a carefully designed GNN model is trained to learn a good but imperfect patch, referred to as the *intermediate patch*. The MaxSAT solver then refines the intermediate patch into an approximately minimum patch. As a result, the search space for the MaxSAT solver is reduced to a smaller space defined by the intermediate patch.

We evaluate `IAMPERE` on two IAM misconfiguration datasets: `Test-A` and `Test-B`. `Test-A` contains 1,000 randomly synthesized large-scale IAM misconfigurations with PE issues by `IAMVulGen`, including hundreds of entities and tens of thousands of permissions. `Test-B` contains two real-world IAM misconfigurations with PEs collected from a US-based cloud security startup.

The experimental results on `Test-A` show that `IAMPERE` is able to repair 57.6% of the IAM misconfigurations within the time limit of 600 seconds, while `IAM-Deescalate` can only repair 21.6%. In addition, `IAMPERE` generates 145 more small patches (i.e., relative patch size is less than 0.2) than `IAM-Deescalate`. Moreover, our ablation study shows that our GNN assisted MaxSAT repair approach significantly outperforms the pure MaxSAT repair approach, repairing 24.1% more IAM misconfigurations. The results on `Text-B` show that `IAMPERE` successfully repairs both real-world IAM misconfigurations within the time limit of 2 hours, while `IAM-Deescalate` fails on both. The contributions of our paper are:

- **Problem Formulation.** The problem of repairing for IAM PEs due to misconfigurations is formulated into a MaxSAT problem.
- **Problem Solving.** GNN is applied to improve the scalability of solving the formulated MaxSAT problem.
- **Implementation.** `IAMPERE` is implemented as a prototype for AWS IAM, which is available at https://github.com/githubhuyang/iampere.

II. Motivation

PE in IAM [8]–[10] involves exploiting design flaws to gain unauthorized access to perform sensitive operations or access confidential data and resources. A misconfigured IAM could allow an attacker to modify its configuration, granting them additional, sensitive permissions. This paper focuses on PEs resulting from IAM misconfigurations. Repairing IAM misconfigurations presents a significant challenge due to three primary reasons.

First, IAM misconfigurations can potentially be exploited to perform *transitive* PEs. IAM configurations can be intricate, including hundreds or even thousands of entities and permissions, and numerous types of relationships among them. Given the complex semantics of the relations among entities and permissions, numerous ways exist for entities to acquire additional permissions. These can be combined in ways that grant an entity (controlled by the attacker) with sensitive permissions, resulting in a transitive PE. For example, Figure 1b shows a transitive PE due to the IAM misconfiguration

example shown in Figure 1a. In the PE, the attacker controls `User B` to apply `Perm C` to become a member of `Group A`, obtain the additional permission `Perm A` through `Group A` (users in a user group can access all permissions of the group), then apply `Perm A` to change the password of `User C` to control `User C`, and finally obtain the sensitive permission `Perm E` through `Group C` to which `User C` belongs.

Second, an IAM misconfiguration may lead to multiple PEs, and a valid repair should eliminate all of these PEs. For example, Figure 1c shows another possible PE due to the misconfiguration example, where the attacker controls `User B` to obtain `Perm B` through `Group B` to which `User B` belongs, applies `Perm C` to become a member of `Group C`, and finally obtains the sensitive permission `Perm E` through `Group C`. Figure 1d shows a valid repair of the misconfiguration example which removes `Perm B` and `Perm C` from `Group B`, eliminating both illustrated PEs shown in Figure 1b and Figure 1c.

Third, an optimal repair should make minimal changes to the semantics of the original IAM configuration, avoiding any alterations to the semantics of IAM configurations that are irrelevant to PEs. For instance, the optimal repair of the misconfiguration example shown in Figure 1a should semantically revoke `Perm B` and `Perm C` from `User B`, without affecting the permission assignments of other entities. The repair in Figure 1d is not an optimal repair, as it inadvertently revokes `Perm C` from `User A`, who is unrelated to any PE. Figure 1e shows an optimal repair which removes `User B` from `Group B` and assigns `Perm D` to `User B`.

To address these three challenges, we introduce `IAMPERE`, an approach that combines MaxSAT solving with GNN to generate near-optimal repairs for IAM misconfigurations.

III. Background

In this section, we briefly introduce GNNs and IAM PEs.

A. Basics of GNNs

GNNs [21], [22] are a class of neural network architectures designed to work with graph data. They typically employ a recursive neighborhood aggregation scheme known as message passing [23]. GNNs accept graphs as input, with nodes containing node features and edges containing edge features. The output consists of node or edge embeddings for each node or edge, respectively. To create node embeddings, GNNs iteratively update the feature vector of each node based on its neighbors. In each iteration, a message-passing layer processes the input graph and produces an updated graph with updated node feature vectors. Traditional GNN models [23]–[25] often use multiple message-passing layers to facilitate iterative updates. Edge embeddings are generally generated using node embeddings of source and sink nodes within the edge, allowing GNNs to effectively capture the graph structure.

B. Basics of IAM PEs

Hu et al. [15] provide detailed definitions and formalizations for IAM configurations and PEs. Here, we offer a simplified and generalized summary of these concepts. We use a simplified version of a real-world IAM misconfiguration as our illustrative example. This misconfiguration led to a significant real-world IAM PE incident in 2019 [2], which resulted in a breach of approximately 100 million credit card records from a US-based financial institution.

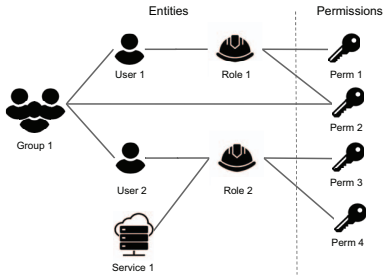


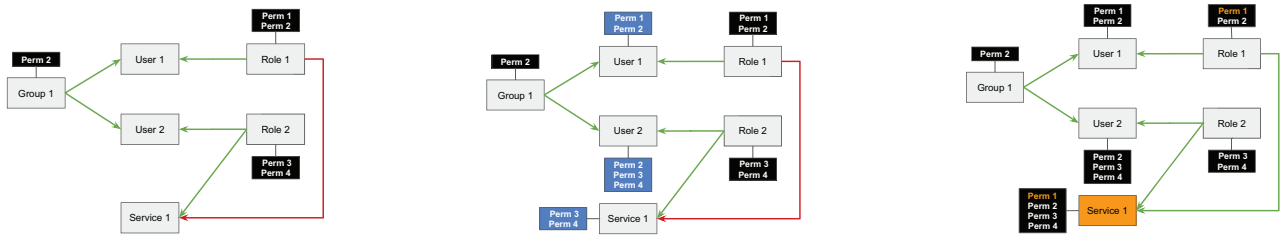
Fig. 2: The relational representation of our illustrative IAM misconfiguration derived from a real-world IAM PE in 2019.

1) The Relational Representation of IAM configurations: An IAM configuration consists of two components: *entities* and *permissions*. An entity represents either a subject or a role in an IAM configuration. Subjects including users, user groups and services can actively perform actions. Roles are created to represent job functions, responsibilities, and privileges within an organization. The entities controlled by the attackers are called *compromised entities*.

Permissions refer to privileges of performing operations. There are three kinds of permissions that are related to IAM PEs, including the *target permissions* which are the sensitive permissions the attacker targets to obtain; and two identified permission types called `Type-I` and `Type-II` permissions that are utilized by the compromised entity to possibly obtain the target permission. A `Type-I` permission allows all permissions of one entity e_1 to be automatically assigned to another entity e_2. Therefore, if a compromised entity u has a `Type-I` permission, u can apply the permission to make e_2 inherit all permissions of e_1. A `Type-II` permission allows to directly assign a `Type-I` or target permission (denoted as p) to an entity e. Therefore, if a compromised entity u has a `Type-II` permission, it can apply the permission, enabling e to acquire p. For example, the permission, that adds a user to a user group in order to make all permissions of the user group assigned to the user, is a `Type-I` permission. The permission attaching an IAM policy to a user is a `Type-II` permission, as it directly assigns each permission in the policy to the user.

There are two kinds of relations in IAM configurations: the entity-permission relations and the entity-entity relations. An entity-permission relation represents which permissions are directly assigned to which entities. In IAM configurations, permissions can be directly assigned to users, user groups and roles. The entity-entity relations can be manually summarized from AWS documentations and relevant studies [8]–[10], [17], [19]. For example, the `Service-Role` and `User-Role` relations represent services and users assuming roles (i.e., becoming members of roles), respectively. The `User-Group` relation represents users are in user groups.

The relational representation of an IAM configuration is a multi-relation graph which includes entities and permissions



(a) The initial PFG. (b) The semantic representation. (c) The PE.

Fig. 3: The PFG initially converted from the relational representation of our illustrative IAM misconfiguration (a), the semantic representation of our illustrative example which is the PFG reaching to the fix point iteration w.r.t. permission flow function (b), and the PFG representing our illustrative IAM PE (c). Enabled permission flows are annotated in green; disabled permission flows are in red; the target permission (i.e., `Perm 1`) and compromised entity (i.e., `Service 1`) are in orange.

as its nodes and relations as its undirected edges. Figure 2 shows the relational representation of our illustrative IAM misconfiguration, including six entities (`Group 1`, `User 1`, `User 2`, `Service 1`, `Role 1` and `Role 2`) and four permissions (`Perm 1`, `Perm 2`, `Perm 3` and `Perm 4`). There are five undirected edges representing three entity-entity relations: `User-Group`, `User-Role`, and `Service-Role` relations, meaning that `User 1` / `User 2` is in `Group 1`, `User 1` / `User 2` is the member of `Role 1` / `Role 2` respectively, and `Service 1` assumes `Role 2`. There are five undirected edges representing two entity-permission relations: `Role-Perm` and `Group-Perm` relations, meaning that `Perm 1` and `Perm 2` are directly assigned to `Role 1`, `Perm 3` and `Perm 4` are directly assigned to `Role 2`, and `Perm 2` is directly assigned to `Group 1`.

2) The Permission Flow Graph: Note that an entity in an IAM configuration can also obtain permissions in indirect ways. For example, all permissions assigned to a role can be indirectly assigned to a user who assumes the role; all permissions assigned to a user group can be indirectly assigned to a user in the user group. For the illustrative example, since `Perm 2` is directly assigned to `Group 1`, users (i.e., `User 1` and `User 2`) in `Group 1` can indirectly obtain `Perm 2`.

A *permission flow* is proposed to represent an indirect permission assignment from one entity to another. Each permission flow has a *flow state* representing whether the permission flow is currently enabled or disabled. If a permission flow from entity e_1 to entity e_2 is enabled, all permissions assigned to e_1 can be automatically assigned to e_2; otherwise (i.e., the permission flow state is disabled), the permissions cannot be automatically assigned.

A *permission flow graph* (PFG) is proposed to represent how permissions are directly or indirectly assigned to entities, which includes entities as its nodes (annotated with permissions assigned to the entities) and permission flows as its edges. Formally, a PFG is defined as a tuple $G = (E, F, \mathcal{A}, \mathcal{W})$, where E is the entity space; $F \subseteq E \times E$ is a set of permission flows; $\mathcal{A} : E \mapsto 2^P$ is a permission assignment function which maps an entity to the permission space P containing `Type-I`, `Type-II` and target permissions; $\mathcal{W} : F \mapsto \{\texttt{true}, \texttt{false}\}$ is a concrete flow state function which outputs whether a permission flow is currently enabled (denoted as `true`) or disabled (denoted as `false`).

3) The Semantic Representation of IAM configurations: The semantic representation is proposed to discover transitive IAM PEs. There are two steps to convert a relational representation of an IAM configuration into a semantic representation.

Step 1: Converting the relational representation to a PFG. To create the semantic representation of an IAM configuration, the first step is to convert its relational representation into a PFG. The nodes in the converted PFG are simply the entities in the relational representation, initially annotated with the directly assigned permissions. To obtain the PFG, the main task is to add edges among the entities for both enabled and disabled permission flows.

The enabled permission flow edges are added based on the semantics of relevant entity-entity relations in the configuration. Figure 3a shows five added enabled permission flows in the PFG converted from the relational representation of the illustrative example in Figure 2. For example, as introduced above, users in the user group can obtain all the permissions assigned to the group; since there is an edge between `Group 1` and `User 1` / `User 2` in the relational representation denoting that `User 1` / `User 2` is in `Group 1`, an enabled permission flow is added from `Group 1` to `User 1` / `User 2`. Note that while all entity-entity relations are converted into enabled permission flow edges in the illustrative example, it is possible that some entity-entity relations do not convey any information about permission flows, and therefore cannot be transformed into enabled permission flow edges.

The disabled permission flow edges are added based on the semantics of each `Type-I` permission in the configuration which allows to enable a permission flow from one entity to another. The `Type-I` permissions are modeled as disabled permission flows which can be enabled in the future, when a compromised entity obtains and applies them. For example, `Perm 3` is the only `Type-I` permission in the illustrative example. Since `Perm 3` allows `Service 1` to assume `Role 1`, a disabled permission flow edge is added from `Role 1` to `Service 1`, as shown in Figure 3a (highlighted in red). The disabled edge can be enabled in the future if the compromised entity applies `Perm 3` to assume `Role 1`.

Step 2: The semantic representation. With the initially converted PFG, the directly assigned permissions are indirectly

assigned to the corresponding entities according to the enabled permission flows. For each entity e, all permissions of the entities which have enabled permission flows to e are assigned to e. This process iteratively continues until reaching to the fixed point, meaning that no new permission is indirectly assigned to any entity. The resulting PFG is the semantic representation of the IAM configuration. Figure 3b shows the semantic representation of our illustrative example, which is the fixed point of the PFG in Figure 3a. The newly assigned permissions through permission flows are highlighted in blue.

Formally, given a PFG G_r converted from a relational representation, its semantic representation is the resulting PFG G_s generated by performing the fixed point iteration on G_r w.r.t. the permission flow function $\mathcal{M}$. The permission flow function $\mathcal{M}$ takes a PFG $G = (E, F, \mathcal{A}, \mathcal{W})$ as an input and outputs a new PFG $G' = (E, F, \mathcal{A}', \mathcal{W})$ where

$$\mathcal{A}'(e_2) = \bigcup_{\{(e_1,e_2)\in F | \mathcal{W}(e_1,e_2)=\texttt{true}\}} \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$$

meaning that, for each entity e_2, all permissions of the entities which have enabled permission flows to e_2 are assigned to e_2. Since $\mathcal{M}$ is monotonic, we have $G_s = \mathcal{M}^*(G_r)$, where $\mathcal{M}^*$ refers to the fixed point iteration of $\mathcal{M}$ (i.e., $\mathcal{M}^*(G_r) = \mathcal{M}^n(G_r)$ s.t. $\mathcal{M}^n(G_r) = \mathcal{M}^{n-1}(G_r)$).

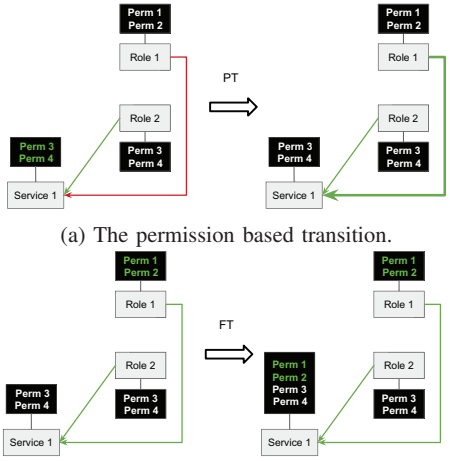
4) PEs due to IAM Misconfigurations: Given a set of compromised entities and a set of target permissions, there is a PE in an IAM configuration iff there exists at least one compromised entity which obtains at least one target permission based on the permissions initially assigned to all compromised entities. We say that an IAM configuration is *safe* if there is no PE in the IAM configuration.

Figure 3c shows a PFG representing a notable PE happened in 2019 due to the illustrative IAM misconfiguration example shown in Figure 2. In the PE example, the compromised entity is an Amazon EC2 instance; the target permission allows access to a sensitive S3 bucket, which contains millions of customers' credit card information. In our illustrative example, `Service 1` corresponds to the compromised entity and `Perm 1` corresponds to the target permission. The semantic of `Perm 3` is that the entity who has `Perm 3` can let `Service 1` assume `Role 1`. To obtain the permission `Perm 1`, the compromised `Service 1` applies `Perm 3` to assume `Role 1`, enabling the permission flow from `Role 1` to `Service 1`. Since `Role 1` is assigned `Perm 1`, the compromised `Service 1` is indirectly assigned the target permission `Perm 1`.

IV. Repair on IAM Semantic Representations

A. The Semantic Representation Reduction

Since our focus is only on the verification and repair for IAM PEs, the semantic representation of IAM configurations can be reduced to include only the entities and permission flows that are relevant to the IAM PEs. Specifically, the entities which never reaches to the compromised entities through enabled/disabled permission flows are removed, because the permissions of those entities can never be assigned to the compromised entities. Correspondingly, the permission flows of those removed entities are also removed. For example, consider the semantic representation in Figure 3b, where `Service 1` is the compromised entity. We notice that `Group 1`, `User 1` and `User 2` can never reach `Service 1` through either enabled or disabled permission flows. Therefore, these three entities and related permission flows are removed. The resulting simplified semantic representation is shown in Figure 4a (on the left). We take the simplified representation as the semantic representation of an IAM configuration for the rest of this paper.



(a) The permission based transition.

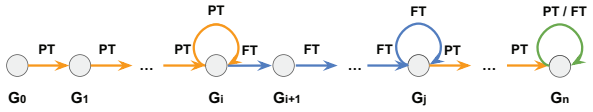
(b) The flow based transition.

Fig. 4: Two types of state transitions.

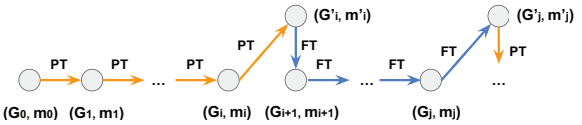
B. Safety Verification of IAM Configurations

To repair IAM misconfigurations, we need to verify if a given IAM configuration is safe. We initially model the problem as a Model Checking [26] problem and apply Fixed Point Iteration (FPI) based approach to solve the problem. To achieve this, we first model the behavior that the compromised entities apply their permissions to modify an IAM configuration as a finite-state machine. Each state in the machine is represented as a PFG; and the initial state is the semantic representation of the IAM configuration.

Each state transition in the machine is triggered by either 1) applying permissions (i.e., `Type-I` or `Type-II` permission), which is called Permission based Transition (PT); or 2) indirectly assigning permissions through enabled permission flows, which is called Flow based Transition (FT). PT models the direct effect of applying a `Type-I` or `Type-II` permission: the application of a Type-I permission is represented as enabling a permission flow, while applying a Type-II permission is depicted as directly assigning a permission to an entity. FT models the side effect of applying a Type-I or Type-II permission, where permissions can "flow" through enabled permission flows. Figure 4a shows a PT of the illustrative IAM configuration, where the state of the permission flow from `Role 1` to `Service 1` changes from disabled to enabled, after the compromised entity `Service 1` applies all its permissions `Perm 3` (which allows to enable a permission



(a) The finite state machine in FPI based verification.



(b) The finite state machine in BMC based repair.

Fig. 5: The finite state machine of IAM safety verification.

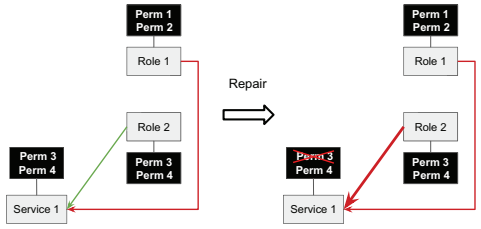


Fig. 6: The minimum patch for the semantic representation of the illustrative IAM misconfiguration.

flow from `Role 1` to `Service 1`) and `Perm 4`. Figure 4b shows a FT of the illustrative example, where the `Service 1` obtains all permissions of `Role 1` through the enabled permission flow between them.

Given the finite-state machine, we check the safety property, asserting that no *error* states (where a compromised entity has at least one target permission) can be reached from the initial state. To perform model checking in polynomial time, we take advantage of the fact that both PT and FT are monotonic transitions. According to fixed point theory [26], a global fixed point state must exist w.r.t. both PT and FT, such that every state in the machine can reach the global fixed point state (through PTs or FTs). Specifically, due to the monotonicity of state transitions, if the global fixed point state is not an error state, then other states must not be error states, thus proving the IAM configuration to be safe.

Figure 5a illustrates the computation of the global fixed point state. Starting from the initial state G_0, a local fixed point state G_i w.r.t. PT is computed. Then, beginning with G_i, a local fixed point state G_j w.r.t. FT is computed. The local fixed points w.r.t. PT or FT are computed alternately until a safe global fixed point state G_n is reached, in which case the IAM configuration is proven to be safe; or an error state is reached after k transitions (also names as *bound*), proving the configuration to be unsafe within the bound k.

C. Repair on IAM Semantic Representations

To conduct a repair towards the semantic representation of an IAM misconfiguration, we apply two kinds of repair operations: 1) revoking an enabled permission flow, (i.e., changing the state of the permission flow from enabled to disabled), and 2) revoking a permission assignment (i.e., removing an assigned permission from an entity). The repair goal for the semantic representation of an IAM misconfiguration is to find a minimum patch containing a sequence of repair operations which makes the IAM configuration safe. Formally, given an IAM semantic representation s, we aim to find a minimum patch t_{min} satisfying that

$$t_{min} = argmin_t |t| \ s.t. \ \mathcal{R}(s,t) \text{ is safe}$$

where $\mathcal{R}(s,t)$ outputs a new semantic representation by applying the patch t to the semantic representation s, and $|t|$ refers to the patch size (i.e., the number of repair operations in a patch). Figure 6 shows the minimum patch for the illustrative IAM misconfiguration example including a sequence of two repair operations: 1) revoking the enabled permission flow from `Role 1` to `Service 1`, and 2) revoking the permission `Perm 3` assigned to `Service 1`.

Algorithm 1 Repair Core

Input: an IAM misconfiguration s (semantic representation), a compromised entity set U, a target permission set L.
Output: a minimum patch t_{min}.

```
function REPAIR_CORE(s, U, L)
    safe, bound ← fpi_verify(s, U, L)
    do
        t_min ← bmc_maxsat_repair(s, U, L, bound)
        r ← apply_rop(s, t_min)
        safe, bound ← fpi_verify(r, U, L)
    while ¬safe
    return t_min
end function
```

To solve such repair problem, the common approach is to translate it into a MaxSAT problem, which encodes repair operations as soft constraints, and the safety verification of the repair as hard constraints. The problem can then be solved using a MaxSAT solver.

To verify the repair, directly encoding the finite state machine of the FPI based safety verification (introduced in the last section) to cover all possible states can lead to a huge number of propositional constraints. To solve this issue, bounded model checking (BMC) is often adopted to encode the verification problem within a certain bound. However, BMC can only check the bounded safety property, instead of the real safety property. To tackle this, the general idea is to use the FPI based verification to decide the initial bound and then iteratively increase the bound until the misconfiguration applied with the proposed minimum patch becomes truly safe. The general algorithm of the BMC based repair using MaxSAT is described in Algorithm 1.

In the following subsections, we first introduce the BMC based safety verification including its finite state machine and bounded safety property, and then illustrate how the MaxSAT problem is encoded with hard and soft constraints.

1) The BMC based Safety Verification: The property that BMC verifies is the bounded safty property, stating that the given IAM configuration is verified to be safe within a certain bound. Figure 5b shows the finite state machine of the BMC based verification. There are still two types of state transitions (i.e., PT and FT). Comparing to the FPI based verification,

$$\frac{G' = G}{m' = \neg m}\ \texttt{T1} \qquad \frac{p \in \mathcal{A}(e)}{p \in \mathcal{A}'(e)}\ \texttt{T2} \qquad \frac{\mathcal{W}(e_1, e_2)}{\mathcal{W}'(e_1, e_2)}\ \texttt{T3}$$

$$\frac{m \quad u \in U \quad p \in \mathcal{A}(u) \quad [\![p]\!] = (e_1, e_2)}{\mathcal{W}'(e_1, e_2)}\ \texttt{PT1}$$

$$\frac{m \quad \neg\mathcal{W}(e_1, e_2) \quad \forall u \in U, p \in \mathcal{A}(u).[\![p]\!] \neq (e_1, e_2)}{\neg\mathcal{W}'(e_1, e_2)}\ \texttt{PT2}$$

$$\frac{m \quad u \in U \quad p_1 \in \mathcal{A}(u) \quad [\![p_1]\!] = (e, p_2)}{p_2 \in \mathcal{A}'(e)}\ \texttt{PT3}$$

$$\frac{m \quad p_2 \notin \mathcal{A}(e) \quad \forall u \in U, p_1 \in \mathcal{A}(u).[\![p_1]\!] \neq (e, p_2)}{p_2 \notin \mathcal{A}'(e)}\ \texttt{PT4}$$

$$\frac{\neg m \quad p \in \mathcal{A}(e_1) \quad \mathcal{W}(e_1, e_2)}{p \in \mathcal{A}'(e_2)}\ \texttt{FT1}$$

$$\frac{\neg m \quad p \notin \mathcal{A}(e_2) \quad \forall e_1 \in \mathcal{N}(e_2).\mathcal{W}(e_1, e_2) \rightarrow p \notin \mathcal{A}(e_1)}{p \notin \mathcal{A}'(e_2)}\ \texttt{FT2}$$

$$\frac{\neg m \quad \neg\mathcal{W}(e_1, e_2)}{\neg\mathcal{W}'(e_1, e_2)}\ \texttt{FT3}$$

Fig. 7: Key operational semantic rules of state transitions. The current state is denoted as (G, m) where $G = (E, F, \mathcal{A}, \mathcal{W})$, and the next state is denoted as (G', m') where $G' = (E, F, \mathcal{A}', \mathcal{W}')$; U refers to a set of compromised entities; $\mathcal{N}(e)$ refers to entities connecting to e through flows.

the main change of the finite state machine in BMC based verification is that the self-loop transition representing the fixed point is converted into two transitions in which the transition type alternates. To distinguish the two state transition types, an extra boolean variable m is introduced, with the value of $true$ representing PT and $false$ representing FT. The states in the finite-state machine becomes the pair (G, m) of a PFG G representing the current state of the IAM configuration and m indicating the current transition type.

Figure 7 shows 10 operational semantic rules for defining state transitions in the finite state machine. First, we define three universal rules for both PTs and FTs:

Rule `T1` defines that the state transition type needs to be alternated when a fixed point w.r.t. PT or FT is reached.

Rule `T2` states that a permission assigned to an entity in the current state will be assigned to the entity in the next state.

Rule `T3` defines that the state of enabled permission flows will not change in the next state.

Next, we introduce four rules for PTs. Rules `PT1` and `PT2`, and rules `PT3` and `PT4` describe transitions triggered by applying `Type-I` and `Type-II` permissions, respectively. Detailed definitions are as follows:

Rule `PT1` defines that if a compromised entity in the current state has a `Type-I` permission which can enable a permission flow, then the permission flow will be enabled in the next state.

Rule `PT2` is the complement of Rule `PT1`, defining that, if a permission flow is disabled and there is no compromised entity having the permission to enable the flow in the current state, the flow will still be disabled in the next state.

Rule `PT3` defines that if a compromised entity has a `Type-II` permission to directly assign a new permission to an entity, the entity will obtain the new permission in the next state.

Rule `PT4` is the complement of Rule 3, defining that if in the current state, an entity is not assigned with a permission, and there is no compromised entity which can directly assign the permission to the entity with a `Type-II` permission, the entity will not have the permission in the next state.

Last, we introduce three rules for FTs:

Rule `FT1` describes how permissions are assigned through enabled permission flows (w.r.t. the permission flow function).

Rule `FT2` is the complement of Rule `FT1`, defining that if in the current state, an entity is not assigned with a permission and no other entities having the permission are connected to the entity through enabled permission flows, then the entity will not have the permission in the next state.

Rule `FT3` defines that if a permission flow is disabled in the current state, it will be disabled in the next state.

2) The MaxSAT Encoding: To generate hard constraints encoding the BMC based safety verification, we first describe how we encode the PFG in each state. We encode a PFG with one boolean variable for each permission assignment to represent whether an entity is assigned a permission, and one boolean variable for each permission flow to represent whether the permission flow is enabled. Next, we encode state transitions by converting the operational semantic rules into boolean constraints. Finally, we encode the bounded safety property with the conjunction of negated boolean variables, one representing a permission assignment between each compromised entity and each target permission.

To generate soft constraints, we encode each repair operation with one boolean variable, representing whether a permission assignment in the IAM misconfiguration persists in the repaired configuration, or whether an enabled permission flow in the IAM misconfiguration remains enabled in the repaired configuration. It is important to note that minimizing the number of repair operations is equivalent to satisfying the maximum number of the encoded boolean variables, thus transforming the problem into a MaxSAT problem.

V. Improving MaxSAT Scalability with GNN

The scalability of MaxSAT-based repair depends on the solving capability of MaxSAT solvers, which can be limited when facing large-scale practical problems due to the exponential growth of the search space. To improve scalability, we employ deep learning to prune the search space for the MaxSAT solver. Given that IAM configurations are graph-structured, we choose Graph Neural Networks (GNNs) to enable more effective learning.

The overview of our GNN-assisted MaxSAT repair approach is illustrated in Figure 8, which consists of two phases: the training phase and the testing phase. During the training phase, we first collect training data samples using the MaxSAT solver. Each data sample contains a graph representation of an IAM misconfiguration and the corresponding minimal patch, computed by the MaxSAT solver. A well-designed GNN model is trained using supervised learning on these data samples. The GNN model learns to predict the probability of each repair operation appearing in a minimal patch, given

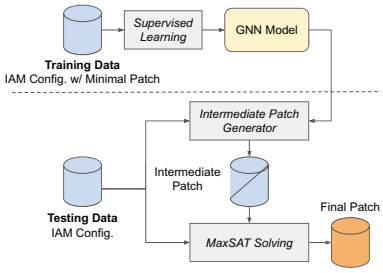


Fig. 8: Training and testing for GNN assisted MaxSAT repair.

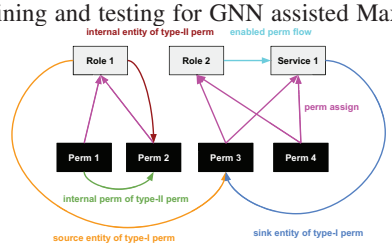


Fig. 9: The graph representation of an IAM configuration which is the input of our GNN model.

the graph representation of an IAM misconfiguration. Details about the graph representation of IAM configurations and the GNN learning are introduced in Subsection V-A and Subsection V-B, respectively.

Based on repair operations sorted by their predicted probability in descending order, a good but imperfect patch called an *intermediate patch* is generated. The MaxSAT solver then takes this intermediate patch and undoes as many unnecessary repair operations as possible, using a greedy algorithm to refine the intermediate patch into an approximately minimal patch. This transforms the repair problem into an easier MaxSAT problem, which is defined as follows:

$$t_{min} = argmin_t|t| \ s.t. \ \mathcal{R}(s,t) \text{ is safe } \wedge \ t \subseteq t_{itm}.$$

where t_{itm} refers the intermediate patch. Therefore, the repair operation search space for the MaxSAT solver is reduced to a smaller search space defined by the intermediate patch. Details about the intermediate patch generation is introduced in Subsection V-C

A. The Graph Representation of an IAM configuration

We transform the semantic representation of an IAM configuration into an input graph for our GNN model. To ensure effective GNN learning, we strive to include all relevant information from the IAM configuration within the graph. The resulting graph representation consists of two node types and six edge types, with each node/edge including its corresponding type as its node/edge features.

Two node types are: nodes representing entities and nodes representing permissions. Six edge types are: edges for enabled permission flows, edges from permissions to assigned entities (representing permission assignments), two edge types for `Type-I` permissions, and two edge types for `Type-II` permissions. A `Type-I` permission, which enables a permission flow, is depicted by one edge from the source entity of the corresponding permission flow to the `Type-I` permission and another edge from the sink entity to the `Type-I` permission. A `Type-II` permission, which allows for the direct assignment of a permission to an entity, is represented by one edge from the corresponding permission to the `Type-II` permission and another edge from the corresponding entity to the `Type-II` permission.

Algorithm 2 The Intermediate Patch Generation

Input: an IAM misconfiguration s (semantic representation), a compromised entity set U, a target permission set L, a list of sorted repair operations α.
Output: an intermediate patch t_{itm}.
function ITM_PATCH_GEN($s,\ U,\ L,\ \alpha$)
  $k \leftarrow \delta$ ▷ δ is the incremental number of repair operations.
  do
    $t_{itm} \leftarrow$ `top_rop`$(\alpha,\ k)$
    $r_{itm} \leftarrow$ `apply_rop`$(s,\ t_{itm})$
    $safe \leftarrow$ `fpi_verify`$(r_{itm},\ U,\ L)$
    $k \leftarrow$ `min`$(k+\delta,$ `size`$(\alpha))$
  while $\neg safe$
  return t_{itm}
end function

Figure 9 shows a converted graph of the semantic representation of the illustrative example shown in Figure 4a (left). The entities, permissions, permission assignments, and enabled permission flows are directly converted. There is only one `Type-I` permission `Perm 3` which allows to enable a permission flow form `Role 1` to `Service 1`. Therefore, one edge from `Role 1` to `Perm 3` and one edge from `Service 1` to `Perm 3` are added. There is only one `Type-II` permission `Perm 2`, which allows to directly assign `Perm 1` to `Role 1`. Therefore, one edge from `Perm 1` to `Perm 2` and one edge from `Role 1` to `Perm 2` are added.

B. The GNN Learning

Our goal is to use GNN to predict the probability of each repair operation appearing in a minimum patch. As discussed in Section IV-C, there are two types of repair operations: revoking an enabled permission flow and revoking a permission assignment. We map these operations to the enabled permission flow edge and the permission assignment edge in the input graph representation, respectively. Thus, the problem becomes predicting the probability of each enabled permission flow edge and each permission assignment edge. To accomplish this, we construct a GNN model comprising six Graph Attention Network layers [27] with graph normalization [28] and skip connections [29] to generate edge embeddings for each permission flow edge and permission assignment edge. Using these edge embeddings, we apply a Multi-Layer Perceptron with one hidden layer to predict the probabilities of the corresponding edges.

C. The Intermediate Patch Generation with GNN predictions

Given an input IAM misconfiguration and a list of repair operations sorted in descending order by probability as predicted by our GNN model, an intermediate patch is generated by sequentially selecting a set number of top repair operations (with high probabilities) until the IAM misconfiguration

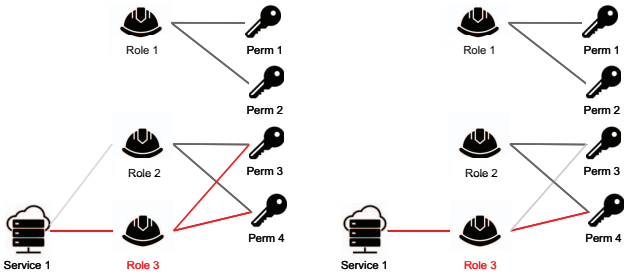


Fig. 10: The repair on the relational representation of the illustrative IAM misconfiguration (for demonstration purposes, we only display the part relevant to the repair). Figure (a) illustrates the relational repair corresponding to the first repair operation, which revokes an enabled permission flow from `Role 2` to `Service 1`. Figure (b) demonstrates the relational repair corresponding to the second repair operation, which revokes `Perm 3` from `Service 1`.

becomes safe. The overall algorithm is outlined in Algorithm 2. During each search round, a patch comprising the top k repair operations is chosen and applied to the input IAM misconfiguration. If the resulting configuration is confirmed to be safe, the applied patch serves as the intermediate patch. If not, another search round begins, featuring a new patch containing the top k repair operations, where k is incremented by a value of δ. This process continues until the repaired IAM misconfiguration is verified to be safe. By default, k is set to 10% of the total number of possible repair operations.

VI. Repair on IAM Relational Representations

Since the relational representation of an IAM configuration has the one-one mapping with the actual IAM configuration, to repair the actual IAM configuration, we need to transfer the patch on the semantic representation to the patch on the relational representation. For the repair operation type that revokes an enabled permission flow from entity e_1 to entity e_2, we remove the edge between e_1 and e_2 in the relational representation, and add edges between e_2 and all permissions assigned to e_2 (in the semantic representation). If entity e_2 cannot be directly assigned permissions, we add an intermediate `Role` entity, create one edge between entity e_2 and the `Role` entity, and add edges between the `Role` entity and all permissions assigned to e_2 (in the semantic representation).

For instance, the first repair operation in the patch for the illustrative IAM misconfiguration involves revoking the permission flow from `Role 2` to `Service 1`. Figure 10a demonstrates the relational repair based on this operation. The repair process starts by removing the edge between `Role 2` and `Service 1`; it then adds the intermediate entity `Role 3`, as `Service 1` cannot be directly assigned permissions; next, it adds an edge between `Service 1` and `Role 3`; finally, it adds edges between `Role 3` and `Perm 3` as well as `Perm 4`, which are assigned to `Service 1` in the semantic representation.

For the other repair operation type, which involves revoking an assigned permission from an entity, if the entity can be directly assigned permissions, we simply remove the edge between the permission and the entity in the relational representation. Otherwise, we delete the edge between the permission and the corresponding intermediate `Role` entity in the relational representation. It is important to note that the order of repair operations in the patch ensures that the revoked permission is directly assigned to the entity, meaning there must be an edge representing such direct permission assignment in the relational representation.

For instance, the second repair operation in the patch for the illustrative misconfiguration involves revoking permission `Perm 3` from `Service 1`. Figure 10b illustrates the relational repair based on this operation, which simply deletes the edge between the intermediate entity `Role 3` and `Perm 3`.

VII. Evaluation

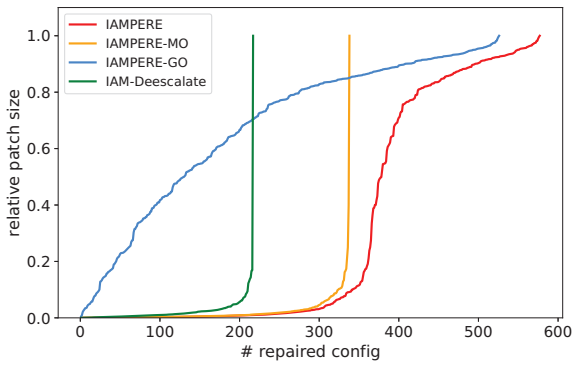
A. Experimental Setup

1) MaxSAT Solver: We apply the complete MaxSAT winner in the Main track of MaxSAT Evaluation 2022 [30] called `CASHWMaxSAT-CorePlus` as our MaxSAT solver.

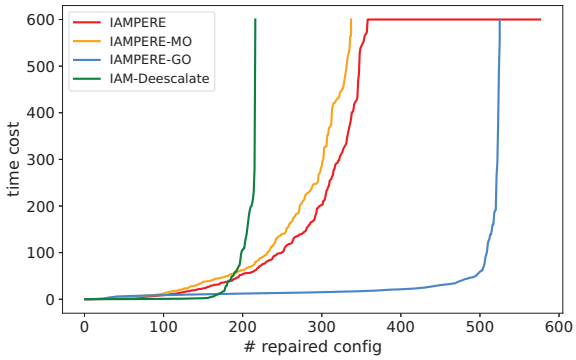
2) Baselines: We adapt `IAM-Deescalate` [20], the only existing IAM PE repair tool (to our knowledge), as our baseline. Furthermore, we construct two variants of `IAMPERE` for conducting an ablation study: `IAMPERE-MO` (MaxSAT Only), which exclusively employs the MaxSAT solver `CASHWMaxSAT-CorePlus`, and `IAMPERE-GO` (GNN Only), which relies solely on GNN to generate the intermediate patch as the final patch.

3) Datasets: To train and evaluate `IAMPERE`, we create four datasets: a training set, a validation set, and three testing sets, named `Test-A`, `Test-B`, and `Test-C`. For the construction of the training and validation sets, we utilize `IAMVulGen` [15], which is an IAM PE task generator. We use it to randomly generate 40,000 IAM misconfigurations with PEs. We then apply `IAMPERE-MO` to obtain minimum patches for these generated IAM misconfigurations. As a result, we acquire 11,933 IAM misconfigurations with minimum patches within the time limit. Each IAM misconfiguration contains between 8 and 336 entities, 24 and 15,525 permissions, and 7 and 15,263 permission flows. We randomly select 90% of these data samples to construct our training set and use the remaining 10% to create our validation set. To construct the testing set `Test-A`, we use `IAMVulGen` to randomly generate 1,000 IAM misconfigurations, each containing between 11 and 315 entities, 42 and 11,737 permissions, and 12 and 11,543 permission flows.

To construct `Test-B`, we gather five real-world IAM configurations owned by cloud customers from a US-based security startup. Our FPI-based model checking approach verifies that two of them are misconfigurations. These two misconfigurations constitute `Test-B`, with one (named as `Real-1`) comprising 251 entities, 2,826 permissions, and 27,939 permission flows, and the other (named as `Real-2`) consisting



(a) The performance w.r.t. the relative patch size of each repair.



(b) The performance w.r.t. the time cost of each repair.

Fig. 11: The effectiveness and efficiency of repair tools on `Test-A`. The cactus plots shows the number of IAM configurations repaired by each tool under a specific relative patch size and time cost, respectively.

of 158 entities, 882 permissions, and 8,704 permission flows. In adherence to the data security protocol established with the startup and its clients, we are restricted from divulging specific details about the two misconfigurations. However, it can be shared that both misconfigurations comprise over 10 PEs, all of which are transitive and have PE path lengths of at least 5. Clearly, these real-world misconfigurations pose significant challenges for repair. Lastly, to construct `Test-C`, we collect 31 IAM misconfigurations with PE from a publicly available IAM PE benchmark set called `IAM-Vulnerable` [31].

4) The Time Limit: Considering that repair problems are typically challenging to solve, we set the time limit for solving each repair problem in the training set, validation set and `Test-A` to 600 seconds, which is double the maximum time limit used in recent MaxSAT evaluations [30], [32], [33]. Due to the complexity and limited number of real-world IAM misconfigurations, we establish a time limit of two hours (7,200 seconds) for solving each problem in `Test-B`. Finally, we set the time limit to 10 seconds for solving each problem in `Test-C`, as its 31 misconfigurations are much simpler than those in `Test-A` and `Test-B`.

5) Evaluation Metrics: The *maximum* patch containing all repair operations (i.e., removing all permission assignments and disabling all permission flows) repairs the IAM misconfiguration in a clearly non-meaningful way. Therefore, in our evaluation, we consider a repair tool that provides a non-maximum patch within the time limit as a valid repair case.

Following the convention in automated software repair, we use *repair rate* (i.e., the percentage of IAM misconfigurations repaired by the tool) and *relative patch size* (i.e., the size of the generated patch divided by the size of the maximum patch) to evaluate the effectiveness of a repair tool. We assess the efficiency of a repair tool based on the time cost of each repair generation.

6) Parameter Settings: For the GNN learning, we train the model in 500 epochs using the AdamW optimizer [34] with a learning rate of 10^{-4}. For the intermediate patch generator and the MaxSAT solver, we apply the default settings.

7) Platform: All experiments are done on a machine with a AMD Ryzen Threadripper 3970X 32-Core Processor, 256GB RAM, and one NVIDIA GeForce RTX 3080 GPU.

B. Research Questions

We aim to answer the following research questions:

- RQ1. How effectively and efficiently does `IAMPERE` perform on `Test-A` with synthesized misconfigurations?
- RQ2. How does `IAMPERE` perform on `Test-B` with real-world misconfigurations, and `Test-C` with misconfigurations from public benchmark set?

C. Experimental Results

1) Performance on `Test-A`: We apply the baseline `IAM-Deescalate`, `IAMPERE` and its two variants `IAMPERE-MO` and `IAMPERE-GO` on `Test-A`. Within the time limit, `IAMPERE` repairs 576 IAM configurations with the highest repair rate (57.6%) among all tools, followed by `IAMPERE-GO` whose repair rate is 52.5%. The repair rate of `IAMPERE-MO` is 33.5%, which is significantly lower than `IAMPERE` and `IAMPERE-GO`. In addition, `IAM-Deescalate` only repairs 216 configurations with the lowest repair rate (21.6%).

The cactus plot in Figure 11a displays the number of IAM configurations on `Test-A` repaired by each tool within a specific relative patch size. We can observe that both `IAMPERE` and `IAMPERE-MO` significantly improve upon `IAM-Deescalate` in terms of patch size. For example, for repaired configurations with a relative patch size less than 0.2, `IAMPERE` and `IAMPERE-MO` repair 145 and 118 more configurations than `IAM-Deescalate`, respectively. Moreover, compared to the pure MaxSAT variant `IAMPERE-MO`, `IAMPERE` not only repairs more configurations but also produces more small patches. Additionally, the performance of `IAMPERE-GO` reveals that the size of the intermediate patches generated by our GNN model is significantly smaller than maximum patches when the number of repaired configurations is small and gradually increases as the number of repaired configurations rises. This demonstrates that our GNN model effectively reduces the search space for the MaxSAT solver.

The cactus plot in Figure 11b displays the number of IAM configurations in `Test-A` repaired by each tool within a spe-

cific time cost. We can observe that IAM-Deescalate has a slight advantage for the first 169 configurations, repairing them within a few seconds, but it is significantly outperformed by IAMPERE and its variants when the repair time increases. Moreover, IAMPERE is consistently more efficient than IAMPERE-MO. We can also observe that IAMPERE is able to repair 220 more IAM misconfigurations exactly at the time limit. This is because IAMPERE can return intermediate patches learned by our GNN model as final patches when the MaxSAT solver struggles. Additionally, the low repair time cost of IAMPERE-GO demonstrates that our GNN model inference for intermediate repairs is highly efficient.

In summary, we conclude that our GNN-assisted MaxSAT repair approach is both effective and efficient in repairing synthesized IAM misconfigurations, achieving a high repair rate and maintaining good patch quality.

2) Performance on Test-B and Test-C: For the IAM misconfiguration Real-1 in Test-B, IAMPERE takes 7,200 seconds to repair the misconfiguration with the relative patch size of 0.889; IAMPERE-GO takes 5,147 seconds to repair with the relative patch size of 0.889; IAMPERE-MO and IAM-Deescalate fail to repair within the time limit. For the IAM misconfiguration Real-2 in Test-B, IAMPERE takes 1,190 seconds to successfully repair the configuration with a relative patch size of 0.0048; IAMPERE-GO takes 2,107 seconds to repair with a relative patch size of 0.741; IAMPERE-MO takes 3,963 seconds to repair with a relative patch size of 0.0048; IAM-Deescalate fails to repair the configuration in the given time limit. The experimental results highlight that solely relying on MaxSAT for fixing real-world misconfigurations can be excessively time-consuming, sometimes even surpassing the allocated time budget. Meanwhile, utilizing GNN independently for these repairs does not significantly reduce the patch size. However, a strategic combination of GNN and MaxSAT facilitates not only a reduction in time spent but also enables the acquisition of smaller or even minimal patches.

As for Test-C, both IAMPERE and its variants successfully repair all 31 misconfigurations, while IAM-Deescalate successfully repairs 24 misconfigurations. All repairs are optimal ones. All tools takes less than 5 seconds for repairing each misconfigurations.

Overall, the results indicate that IAMPERE is competitive in repairing real-world IAM misconfigurations as well as the open-source IAM PE benchmark set.

VIII. Related Work

In addition to the only existing IAM repair tool IAM-Deescalate, and the most recent IAM detector TAC, recent research on the formal verification of cloud access control policies [16], [35], [36] is also relevant to our paper. The AWS team has proposed ZELKOVA [36], which can formally verify several basic security and availability properties of IAM configurations (e.g., whether an arbitrary user can write to a resource) by encoding the problems into Satisfiability Modulo Theories (SMT) formulas. Block Public Access [35] is built on top of ZELKOVA to formally verify access control policies for Amazon Simple Storage Service (S3), ensuring the policies only allow access to trusted entities. Beyond the efforts from the AWS team, Ilia and Oded [16] have proposed an SMT-based bounded model checking approach to formally verify if an IAM configuration has a PE. These approaches focus solely on analyzing and reasoning permissions defined in policy documents, whereas IAMPERE also considers reasoning about the relations among entities and permissions, providing more useful information to reveal broader classes of IAM PEs.

IX. Discussion and Conclusion

Our IAM repair approach is sound, ensuring that the patch produced by IAMPERE is guaranteed to eliminate PEs in the IAM misconfiguration. While IAMPERE aims to generate an approximately minimum patch by leveraging GNN, it does not guarantee the absolute minimum, which makes our approach incomplete. Nonetheless, thanks to our well-designed GNN architecture, informative graph representation and extensive training, our experimental results show that the generated patches are remarkably small. Furthermore, our research highlighted the substantial efficacy of GNN in facilitating MaxSAT solving, indicating a breakthrough with a broader impact that extends far beyond IAM PE repair. This innovation could potentially enhance a wide range of MaxSAT applications, including planning and scheduling [37]–[40], hardware/software verification and repair [41]–[43], bioinformatics [44]–[46], etc. Moreover, our work serves as a tangible demonstration of employing deep learning to bolster automated reasoning.

In conclusion, we introduce IAMPERE, a comprehensive and efficient approach designed to repair a wide range of IAM PEs using an approximately minimal patch. We first formulate the IAM repair problem as a MaxSAT problem, which is inherently challenging. To improve the scalability of MaxSAT solving, we employ GNNs to learn an intermediate patch that is relatively small, though not necessarily minimal. We then apply a MaxSAT solver to the intermediate patch, greedily refining it into an approximately minimal patch. Experimental results on both synthesized and real-world datasets demonstrate that IAMPERE significantly outperforms the baseline in terms of repair effectiveness and efficiency.

Acknowledgment

We would like to thank Alex Cathis for configuration extraction and the anonymous reviewers for their valuable feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and by the Intel RARE center. This work was also supported by a grant from the Army Research Office accomplished under Cooperative Agreement Number W911NF-19-2-0333. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] AWS, "Aws identity and access management (iam)," https://aws.amazon.com/iam/, 2023.

[2] C. One, "Information on the capital one cyber incident," https://www.capitalone.com/digital/facts2019/, 2022.

[3] C. Pernetf, "Research reveals that iam is too often permissive and misconfigured," https://www.techrepublic.com/article/research-iam-permissive-misconfigured/, 2021.

[4] N. Quist, "Unit 42 cloud threat report update: Cloud security weakens as more organizations fail to secure iam," https://unit42.paloaltonetworks.com/iam-misconfigurations/, 2021.

[5] A. Morag, "Cloud misconfigurations: The hidden but preventable threat to cloud data," https://www.infosecurity-magazine.com/opinions/cloud-misconfigurations-threat/, 2021.

[6] Cybersecurity Insiders, "Cloud security report," https://www.isc2.org/Landing/cloud-security-report, 2021.

[7] DivvyCloud, "2020 cloud misconfigurations report," https://divvycloud.com/wp-content/uploads/2020/02/Cloud-Misconfiguration-Report-FINAL.pdf, 2020.

[8] S. Gietzen, "Aws iam privilege escalation – methods and mitigation," https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/, 2018.

[9] G. Kleijn, "Well, that escalated quickly: Privilege escalation in aws," https://bishopfox.com/blog/privilege-escalation-in-aws, 2022.

[10] E. Kedrosky, "Achieving aws least privilege: Understanding privilege escalation," https://sonraisecurity.com/blog/common-methods-aws-privilege-escalation/, 2022.

[11] R. S. Lab, "Pacu: The open source aws exploitation framework," https://rhinosecuritylabs.com/aws/pacu-open-source-aws-exploitation-framework/, 2022.

[12] Salesforce, "Cloudsplaining," https://cloudsplaining.readthedocs.io/en/latest/, 2022.

[13] N. Group, "Principal mapper," https://github.com/nccgroup/PMapper, 2023.

[14] W. Labs, "A graph-based tool for visualizing effective access and resource relationships in aws environments," https://github.com/WithSecureLabs/awspx, 2022.

[15] Y. Hu, W. Wang, S. Khurshid, and M. Tiwari, "Interactive greybox penetration testing on cloud access control with iam modeling and deep reinforcement learning," 2023.

[16] I. Shevrin and O. Margalit, "Detecting multi-step iam attacks in aws environments via model checking," in *The 32nd USENIX Security Symposium*, 2023.

[17] AWS, "Aws service authorization reference: Actions, resources, and condition keys for aws services," https://docs.aws.amazon.com/pdfs/service-authorization/latest/reference/service-authorization.pdf#reference_policies_actions-resources-contextkeys, 2023.

[18] ——, "Security best practices in iam," https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html, 2023.

[19] Xscaler, "Anatomy of a cloud breach: How 100 million credit card numbers were exposed," https://www.zscaler.com/resources/white-papers/capital-one-data-breach.pdf, 2021.

[20] J. Chen, "Iam-deescalate: An open source tool to help users reduce the risk of privilege escalation," https://unit42.paloaltonetworks.com/iam-deescalate/, 2022.

[21] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[22] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[23] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[25] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[26] K. L. McMillan and K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[27] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?" *arXiv preprint arXiv:2105.14491*, 2021.

[28] T. Cai, S. Luo, K. Xu, D. He, T.-y. Liu, and L. Wang, "Graphnorm: A principled approach to accelerating graph neural network training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 1204–1215.

[29] J. You, Z. Ying, and J. Leskovec, "Design space for graph neural networks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 009–17 021, 2020.

[30] F. Bacchus, M. Järvisalo, J. Berg, R. Martins, and A. Niskanen, "Maxsat evaluation 2022," https://maxsat-evaluations.github.io/2022/index.html, 2022.

[31] S. Art, "Iam vulnerable - assessing the aws assessment tools," https://bishopfox.com/blog/assessing-the-aws-assessment-tools, 2021.

[32] F. Bacchus, M. Järvisalo, J. Berg, and R. Martins, "Maxsat evaluation 2021," https://maxsat-evaluations.github.io/2021/organization.html, 2021.

[33] ——, "Maxsat evaluation 2020," https://maxsat-evaluations.github.io/2021/rankings.html, 2020.

[34] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[35] M. Bouchet, B. Cook, B. Cutler, A. Druzkina, A. Gacek, L. Hadarean, R. Jhala, B. Marshall, D. Peebles, N. Rungta *et al.*, "Block public access: trust safety verification of access control policies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 281–291.

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

[37] L. Zhang and F. Bacchus, "Maxsat heuristics for cost optimal planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, no. 1, 2012, pp. 1846–1852.

[38] E. Demirović, N. Musliu, and F. Winter, "Modeling and solving staff scheduling with partial weighted maxsat," *Annals of Operations Research*, vol. 275, pp. 79–99, 2019.

[39] M. Bofill, M. Garcia, J. Suy, and M. Villaret, "Maxsat-based scheduling of b2b meetings," in *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer, 2015, pp. 65–73.

[40] Q. Yang, K. Wu, and Y. Jiang, "Learning action models from plan examples using weighted max-sat," *Artificial Intelligence*, vol. 171, no. 2-3, pp. 107–143, 2007.

[41] A. Morgado, M. Liffiton, and J. Marques-Silva, "Maxsat-based mcs enumeration," in *Hardware and Software: Verification and Testing: 8th International Haifa Verification Conference, HVC 2012, Haifa, Israel, November 6-8, 2012. Revised Selected Papers 8*. Springer, 2013, pp. 86–101.

[42] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[43] B. Salimi, L. Rodriguez, B. Howe, and D. Suciu, "Interventional fairness: Causal database repair for algorithmic fairness," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 793–810.

[44] C.-M. Li, Z. Fang, and K. Xu, "Combining maxsat reasoning and incremental upper bound for the maximum clique problem," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. IEEE, 2013, pp. 939–946.

[45] D. Allouche, I. André, S. Barbe, J. Davies, S. de Givry, G. Katsirelos, B. O'Sullivan, S. Prestwich, T. Schiex, and S. Traoré, "Computational protein design as an optimization problem," *Artificial Intelligence*, vol. 212, pp. 59–79, 2014.

[46] Y. Zhang, H. Zha, C.-H. Chu, and X. Ji, "Protein interaction inference as a max-sat problem," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)-Workshops*. IEEE, 2005, pp. 146–146.