Learning Relationship-Based Access Control Policies from Black-Box Systems

PADMAVATHI IYER and AMIRREZA MASOUMZADEH, University at Albany – SUNY, USA

Access control policies are crucial in securing data in information systems. Unfortunately, often times, such policies are poorly documented, and gaps between their specification and implementation prevent the system users, and even its developers, from understanding the overall enforced policy of a system. To tackle this problem, we propose the first of its kind systematic approach for learning the enforced authorizations from a target system by interacting with and observing it as a black box. The black-box view of the target system provides the advantage of learning its overall access control policy without dealing with its internal design complexities. Furthermore, compared to the previous literature on policy mining and policy inference, we avoid exhaustive exploration of the authorization space by minimizing our observations. We focus on learning relationship-based access control (ReBAC) policy, and show how we can construct a deterministic finite automaton (DFA) to formally characterize such an enforced policy. We theoretically analyze our proposed learning approach by studying its termination, correctness, and complexity. Furthermore, we conduct extensive experimental analysis based on realistic application scenarios to establish its cost, quality of learning, and scalability in practice.

CCS Concepts: • **Security and privacy** → **Access control**; **Authorization;**

Additional Key Words and Phrases: Relationship-based access control, black box, model learning, formal analysis

ACM Reference format:
Padmavathi Iyer and Amirreza Masoumzadeh. 2022. Learning Relationship-Based Access Control Policies from Black-Box Systems. *ACM Trans. Priv. Secur.* 25, 3, Article 22 (May 2022), 36 pages.
https://doi.org/10.1145/3517121

1 INTRODUCTION

The access control policy determines who can access what resources in a system. Understanding such a policy is essential for users to safely and effectively use a system. However, many systems do not provide adequate and accurate documentation of their policies. There are many factors, such as scale, heterogeneity of system components, fast-pace development, and lack of resources, that make it challenging even for developers to understand the overall enforced policy of a system. We note that even in the presence of resources, understanding the enforced policy requires exploring a daunting access space. Given these challenges, we argue that novel techniques should

This material is based upon work supported by the National Science Foundation under Grant No. 2047623.
Authors' address: P. Iyer and A. Masoumzadeh, University at Albany – SUNY, Albany, New York 12222; emails: {riyer2, amasoumzadeh}@albany.edu.

2471-2566/2022/05-ART22 $15.00
https://doi.org/10.1145/3517121

be developed for automated learning of access control policy specification. In particular, in this article, we propose to learn the overall enforced policy of a system by observing its authorization behavior. By treating a system as a black box, we intend to develop a learning technique that is independent of the inner design elements and structures of a system, such as the programming languages and components.

Prior approaches to learning access control policies are largely in the context of *policy mining*, where the goal is to mine a policy in a high-level model from authorization information in a low-level representation (e.g., access control lists or logs). This is useful when migrating to an environment with a new access control model or reconstructing existing policies for improved maintenance. Policy mining has been previously investigated in the context of role-based access control [37, 38], attribute-based access control [20, 27, 47], and relationship-based access control [10, 15, 28]. The common requirement for all policy mining approaches is the availability of an existing policy or a comprehensive access log, which is impractical in our context when an accurate policy is not available in the first place. There has also been a few previously-proposed approaches to learn policies directly from systems [31, 35]. However, these approaches assume that they can exhaustively explore the full access space or that they can reduce the explorable access space based on some expensive manual analysis and reverse engineering approaches. Verifying the correctness of such manual processes themselves will be impractical.

Rather than relying on the availability of authorization information or the ability to fully explore it, we take a novel approach to learning access control policies by interactively observing authorization decisions that a system makes. In particular, we extend the notion of *model learning* [6, 43], a framework for learning formal specification of black-box systems, to the domain of access control policies. Model learning is an active learning paradigm to infer a state-transition model of a system by providing inputs to and observing outputs of the system. In the context of our work, we propose to learn a **deterministic finite automoton** (**DFA**) model that formally characterizes the access control decision-making process in a system, i.e., its access control policy, by selectively submitting access requests to the system and observing the associated decisions. We choose to focus on learning **relationship-based access control** (**ReBAC**) policies. ReBAC is one of the most expressive authorization models for applications with rich data models [21, 24]. A ReBAC policy specifies a set of permitted relationship patterns between the authorized users and protected resources. The objective of our proposed framework is to systematically learn a DFA enforcing a system's ReBAC policy while minimizing the number of access control observations. Learning such a *ReBAC policy DFA* by observing the access request decisions is challenging since an access request can be associated with multiple (permitted and non-permitted) relationship patterns. This article is a significantly extended version of an earlier conference publication [29].

The highlights of our contributions in this article are as follows. We review and contrast the closely related work with this article (Section 2). We propose the notion of *ReBAC policy DFA* as a formal representation of the policy decision-making process in a system that we intend to learn. We present a reference ReBAC policy model and its corresponding ReBAC policy DFA (Section 3). We propose our novel framework for active learning of authorizations (Section 4) and extensively discuss our design of its components and their interactions. The learner component in our framework (Section 5) is responsible for learning the policy based on its authorization observations and processing counterexamples it receives from other components. We propose the mapper component (Section 6) to facilitate the interaction of the learner with the system by translating access control requests/decisions between the abstract domain of relationship patterns (expressed in ReBAC policies) and the concrete domain of users/resources observable from an application. We propose how the equivalence oracle component (Section 7) can test the equivalency of a ReBAC policy DFA hypothesis constructed by the learner against a system. We formally analyze our

framework by proving its termination, correctness, and complexity. For clarity of the presentation, we initially discuss the correctness of our framework when using a simplified, exhaustive mapper approach (Section 8). Then, we present the formal results for our full framework using the proposed mapper (Section 9). Finally, we provide an extensive experimental analysis of a prototype implementation of our framework in terms of learning cost and performance, scalability, and the implementation of the equivalence oracle in practice (Section 10).

2 BACKGROUND AND RELATED WORK

In this section, we first discuss the literature closely related to our work. Then, we briefly discuss the background on a model learning framework from which we adopt ideas in this work.

Policy Mining. Policy mining algorithms produce a high-level policy from existing lower-level permission information. This problem has been extensively investigated in the context of **role-based access control (RBAC)**. Given the **user-permission assignments** (**UPA**), the objective of *role mining* is to find an optimal set of roles R and corresponding **user-role assignments** (**UA**) and **role-permission assignments** (**PA**) that make the policy equivalent to the given UPA [37]. Vaidya et al. defined the basic version of this problem minimizing the number of roles [44]. A variant of this problem also considers minimizing the number of assignments (|UA|+|PA|) [33, 45]. Other (more comprehensive) optimization metrics consist of a cost-based metric that minimizes the overall administration costs due to |UA|, |PA|, |R|, and additional cost due to other business information [18], and the *weighted structural complexity optimization* that aims at minimizing the weighted sum of the number of elements in R, UA, PA, **role hierarchy** (**RH**), and other components of an RBAC system [38].

Subsequently, researchers considered the problem of obtaining **attribute-based access control** (**ABAC**) policies by mining rules based on the attributes associated with users and resources. Xu and Stoller proposed a heuristic approach for mining from an **access control list** (**ACL**) policy and attribute data [47], and a variant of it for mining from operation logs when an ACL is not available (e.g., if the current access policy is encoded in a program) [48]. Later works in this area include an evolutionary, separate and conquer mining approach [36], a *constrained* mining algorithm that limits the maximum weight (size) of each mined rule [25], mining both positive and negative authorization rules [27], an unsupervised learning approach based on k-modes clustering [30], and using natural language processing to extract policies from natural language documents [5, 39]. Cotrini et al. used a machine learning algorithm for subgroup discovery to mine ABAC policies [20]. They also developed a generalized policy mining framework, which can be specialized to produce mining algorithms for a variety of policy languages including RBAC and ABAC [19]; the downside is that the resulting algorithms may achieve lower policy quality than customized algorithms for specific policy languages. Recently, the problem of incremental policy maintenance was presented, which only updates rules that are affected by any permission/attribute changes [9].

Unlike RBAC and ABAC, ReBAC policies deal with relational policies rather than unary predicates on attributes and roles. Bui and Stoller presented a greedy solution for mining ReBAC policies from access control lists and entity relationships [13]. Later, the authors extended their work to a grammar-based evolutionary algorithm [15] and mining from incomplete and noisy permission data [14]. The authors have also proposed combining neural networks and a grammar-based genetic algorithm to support additional policy language features such as set-equality and subset-equal set comparison operators [12]. More recently, they presented a simpler algorithm based on decision trees and its variant that can mine policies with negation conditions [10]. They also extended that work for mining policies when some attribute values are unknown [11]. Iyer and Masoumzadeh proposed a solution for mining ReBAC authorization rules in an evolving system based on rule mining and frequent graph-based pattern mining concepts [28]. Chakraborty and

Sandhu considered the problem of feasibility of ReBAC policy mining in the context of user to user authorizations, which determines whether there exists a ReBAC policy corresponding to the given relationship graph and lower-level authorizations [16].

Compared to the above works, we propose a novel, systematic approach for actively learning access control policies where target systems are treated as black boxes, in order to observe and learn the authorization behavior of the system interactively. In particular, the previous policy mining algorithms assume the provision of complete permission space. But, in our approach, we characterize the enforced ReBAC policy of a system by interacting with its access control engine using a minimal number of access requests. We propose the first of its kind policy learning approach that employs model learning for inferring policies from black-box applications.

Policy Inference. There has been some previous work on inferring policies from web applications in a black-box fashion. To validate the enforcement of access control policies in web applications, Le et al. presented a semi-automated approach to infer those policies [31]. Their approach employs the *RandomTree* classifier on a system's user permission space to infer access rules. Masoumzadeh also proposed an approach to infer the privacy control policy of an **online social network** (**OSN**) to help end users better understand the implicit policies imposed by the system [35]. The motivation was that access to user information is governed by a collection of privacy settings and fixed policies specified by the OSN. But, an OSN such as Facebook is less than transparent about such fixed policies, and consequently some policies are unknown to users. These works, however, assume that they can exhaustively explore the access space first to generate an authorization log. The problem is essentially reduced to access control mining once such an authorization log has been gathered.

Model Learning. Model learning [43] aims at formally characterizing the behavior of a black-box system by providing inputs and observing the resulting outputs. It is emerging as a highly effective bug-finding technique, with applications shown over a wide range of domains such as network protocols, security, legacy software, and banking cards. In the domain of network protocols, researchers have employed combination of model learning and model checking to study different software components in TCP implementation of different operating systems, and their interactions [23]. Black-box testing has been also used to investigate flaws in the TLS protocol implementations [22]. In the security domain, researchers have proposed a black-box differential testing framework called SFADIFF [7] to automatically detect differences between a set of programs with comparable functionality. Model learning has been also utilized to analyze **regular expression** (**RE**) filters and string sanitizers in a black-box manner [8]. Model learning has also been applied for refactoring legacy embedded software [34, 41] and for reverse engineering smartcards and smartcard readers [3, 17]. Another line of research in the software engineering domain, which works toward a similar goal to model learning focuses on generating finite state machines or UML statecharts from scenario-based system requirements represented in the form of UML sequence diagrams [26, 46]. However, such approaches mine patterns based on a set of already-collected sequence diagrams instead of an active learning strategy as in model learning (and our approach). Our proposed framework in this article is the first model learning approach to learning access control policies. In particular, we propose a model learning technique for efficiently observing high-level access patterns from the substantially large access space.

2.1 Background on L* Algorithm and MAT Framework

A foundational work in the model learning area is the approach of a **minimally adequate teacher** (**MAT**) proposed by Angluin. Angluin studied the problem of actively learning a finite-state automaton through observations of its members and nonmembers [6]. Particularly, their work

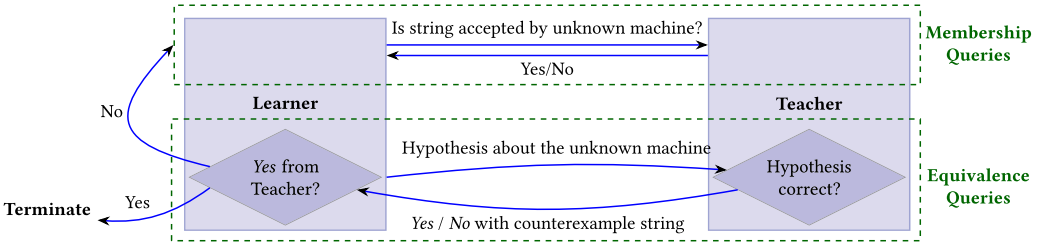


Fig. 1. Mechanism of the MAT Framework.

presents an algorithm, referred to as L^*, for efficiently learning an initially unknown finite machine from any minimally adequate teacher. The MAT framework includes a *learner* component that has to infer the behavior of an unknown machine M by asking queries to a *teacher*. The teacher has knowledge of the machine M. Initially, the learner only knows the input/output alphabet of M. The learner pursues its task by submitting two types of queries to the teacher:

— **Membership Query:** A membership query about a string z asks if z is accepted by automaton M or not. The teacher replies with a *yes* or *no* depending on whether z is in the language of M.
— **Equivalence Query:** With an equivalence query, the learner asks if a hypothesized machine M_H is correct, i.e., whether M_H is equivalent to the unknown machine M. The teacher answers *yes* if this is the case. Otherwise, the teacher answers *no* and provides a *counterexample* that distinguishes M_H and M (i.e., a counterexample is a string in the symmetric difference of the correct machine and the hypothesized machine).

Figure 1 shows interactions between the learner and the teacher in the MAT framework that are triggered in the L* algorithm. The membership and equivalence queries are shown in upper and lower parts of the figure, respectively.

3 POLICY DFA: REBAC POLICY MODEL & ITS DFA REPRESENTATION

In this article, we focus on *ReBAC* policies in which the authorizations are expressed in terms of sequence of relationships between entities (users and resources) in a system. In this section, we provide a reference model for ReBAC based on existing literature [21, 24, 40] and discuss our approach to formally represent such policies to be suitable for our active learning algorithm. Specifically, we discuss the format and evaluation of ReBAC policies in Section 3.1. In addition, we detail the data model used to capture authorization information about users and resources, which is also utilized while making access decisions. In Section 3.2, we propose how we capture a ReBAC policy using a DFA as a formal representation of a machine that enforces the policy. Table 1 summarizes the notations used throughout this article.

3.1 ReBAC Reference Model

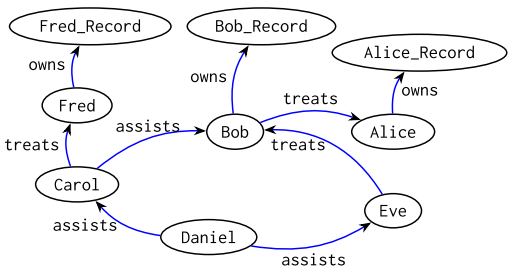
In applications that support ReBAC policies, the data stored in the application can be modeled in the form of a *system graph* [21], where nodes indicate users and resources created in the system, and edges represent relationships between them. ReBAC policies utilize relationship information contained in the system graph to make access control decisions.

Definition 1 (System Graph). Let U and R be the set of users and resources in the system. Let L be the set of possible relationship types among users and resources. The *system graph* is a directed

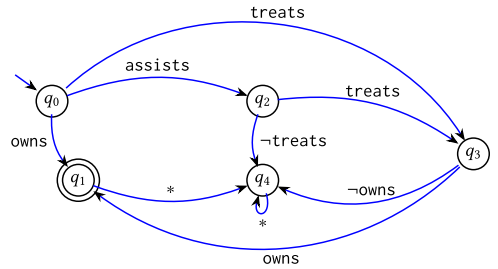
Table 1. List of Notations

Notation	Meaning
G	System graph
U,R,V,E,L	Users, resources, vertices, edges, edge labels
C	All possible access requests in G
$\Pi_{u,r}$	Set of paths between u and r in G
$\mathcal{R}$	Patterns over L of max length N
ϕ	Relationship pattern / ReBAC rule
Φ	Set of relationship patterns / ReBAC policy
Φ_I	ReBAC policy of SUL
M, $\mathcal{L}(M)$	General notation for DFA, language of M

Notation	Meaning
Q,F,δ	States, final states, transitions in M
M_H, M_I	Learner hypothesis DFA, DFA of SUL
O_S, O_E	Prefix-closed, suffix-closed set of patterns
O_τ	Mapping patterns to $\{0, 1\}$
P_ϕ	All relationship patterns in G
P_C	Mapping from P_ϕ to access requests
$(\mathcal{I}_{\widehat{\phi}})$, $\mathcal{I}_\phi$	(Certain) Queried membership patterns
$\mathcal{I}_\tau$	Mapping from $\mathcal{I}_\phi$ to access decisions
.	Concatenation operator



(a) System Graph (Example 1).



(b) ReBAC Policy DFA (Example 2). Special Transition $*$ Matching Any Label in $\{owns, treats, assists\}$.

Fig. 2. Running example for an electronic medical records system.

graph $G(V, E)$ where $V = U \cup R$ is the set of entities and each edge in $E \subseteq V \times V \times L$ indicates a relationship between two entities.

Example 1 (System Graph). Figure 2(a) illustrates a sample subset of a system graph of entities and relationships in an electronic medical records system. The entities include users `Alice`, `Bob`, `Carol`, `Daniel`, `Eve`, and `Fred`, and resources `Alice_Record`, `Bob_Record`, and `Fred_Record`. In this scenario, `Alice` is receiving treatment from `Bob`, who, in turn, is being treated by `Eve`. `Carol` is in charge of providing medical treatment for `Fred`, along with assisting `Bob` as his nurse. `Daniel` happens to be the nurse working under both `Carol` and `Eve`. `Alice_Record`, `Bob_Record` and `Fred_Record` contain all information regarding the medical treatment(s) being received by `Alice`, `Bob`, and `Fred`, respectively.

ReBAC authorization policies grant accesses based on relationship patterns, which characterize different arrangements of labeled edges between the entities in the system graph.

Definition 2 (Relationship Pattern). A *relationship pattern* ϕ is a sequence of relationship labels $\langle l_1.l_2.\ldots.l_n\rangle$, where $l_i \in L$ and $0 \leq n \leq N$. We denote the domain of relationship patterns by $\mathcal{R}$. $\sigma \in \mathcal{R}$ denotes the empty pattern.

In the above definition, N denotes the maximum allowable length of a relationship pattern that is determined by the target system.

In this article, we assume that all authorization rules are about granting the same right (or action). This will help simplify our discussions by *using relationship patterns and authorization rules interchangeably*. Extending the rules to consider an extra component that determines applicable

right is straightforward. Finally, we define a ReBAC policy, which is responsible for regulating access control in the target system, as follows:

Definition 3 (ReBAC Policy). A ReBAC policy, denoted by Φ, is a set of authorization rules $\{\phi_i\}$ where each authorization rule ϕ_i is a relationship pattern.

We alternatively use the term *permitted pattern* to refer to an authorization rule ϕ. Also, we refer to any pattern ϕ' that is not part of authorization policy as a *non-permitted pattern*, i.e., $\phi' \in \mathcal{R} \setminus \Phi$.

We represent an *access request* as tuple $\langle u, r \rangle$, where user $u \in U$ requests to access resource $r \in R$. As discussed earlier in this section, we use an abstract notion of an access right for simplicity; therefore, the right does not need to be specified in an access request. An access request would be *permitted* only if it matches at least one of the rules in the policy. Let $\Pi_{u,r}$ be the set of paths in G from u to r constrained by the maximum allowable length of relationship patterns. Path π matches an authorization rule ϕ if and only if the sequence of edge labels in π matches the sequence of the labels in ϕ. Moreover, we employ the *deny-by-default* strategy. Therefore, if a given access request does not match any authorization rule in the policy then it will be *denied* access to the resource.

Continuing on Example 1, let us provide a ReBAC policy in the context of an electronic medical records system as a running example.

Example 2 (ReBAC Rules). The ReBAC policy enforced by the medical records system consists of the following authorization rules: $\langle owns \rangle$, $\langle treats.owns \rangle$, and $\langle assists.treats.owns \rangle$.

The above policy allows the access of medical records, which are the protected resources in this system, to patients that they belong to (pattern $\langle owns \rangle$), along with the doctors who are treating those patients ($\langle treats.owns \rangle$) and the nurses working under those doctors ($\langle assists.treats.owns \rangle$).

Example 3 (Access Request Evaluation). Based on the system graph in Example 1 and ReBAC policy in Example 2, the following shows list of protected resources and their authorized users in the system:

— `Alice_Record`: `Alice`, `Bob`, and `Carol`.
— `Bob_Record`: `Bob`, `Daniel`, and `Eve`.
— `Fred_Record`: `Carol`, `Daniel`, and `Fred`.

For each of the above authorizations, there exists a path from the user to the resource in the given system graph that matches at least one of the rules in the specified ReBAC policy. For instance, the access request $\langle Bob, Alice\_Record \rangle$ matches the rule $\langle treats.owns \rangle$ given in Example 2.

3.2 ReBAC Policy DFA

Our objective is to infer the access control policy enforced by a system in a black-box manner. We approach this problem by conducting an active learning strategy to infer a formal model of the component in the system that evaluates access requests. In the access control literature, this component is referred to as *policy decision point* [42]. In particular, we employ the concept of *DFA* to represent the policy decision point that evaluates an access request against the system's ReBAC policy. We refer to such a formal model of enforcing ReBAC policies as *ReBAC policy DFA* (or *policy DFA* for short). In the following, we show how we can produce a ReBAC policy DFA based on the reference ReBAC model described in Section 3.1, and how we bridge the expressiveness gap between ReBAC policies and our DFA representation.

Definition 4 (ReBAC Policy DFA). A *ReBAC policy DFA* M is a DFA, denoted by a 5-tuple $\langle Q, L, \delta, q_0, F \rangle$, where Q is a finite set of states, L is the finite set of input symbols called the alphabet (relationship labels in our context), $\delta : Q \times L \longrightarrow Q$ is the state transition function, q_0 is the

initial state, and $F \subseteq Q$ is the set of accepting (final) states. Additionally, M must be constrained such that its language, $\mathcal{L}(M)$, is finite.

An *accepting string* in M starts from initial state q_0 and ends in an accepting state $q_n \in F$. The *language of* M, denoted by $\mathcal{L}(M)$, is the set of all its accepting strings. For brevity of the DFA presentation, we use the notation $\neg l$ to imply $\{l' \in L | l' \neq l\}$, i.e., any relationship label but l. Also, we use $*$ to indicate a set of transitions in the DFA corresponding to every input symbol in L.

Based on the above definition of a ReBAC policy DFA, we observe that any sequence of transitions starting from initial state forms a string of relationship labels, which is equivalent to a relationship pattern (Definition 2). Therefore, each accepting string in a policy DFA can be seen as the relationship pattern ϕ of an authorization rule. As a result, $\mathcal{L}(M)$ represents an authorization policy Φ.

It is important to note that the language of a ReBAC policy DFA, $\mathcal{L}(M)$, must be finite. This is because, the ReBAC model discussed earlier in this section (Definitions 2 and 3) is not as expressive as a DFA. In particular, our policy does not support relationship patterns with repeating sub-patterns as in regular expressions. Therefore, our goal is to learn a *constrained DFA model* that accepts a finite language for representing a ReBAC authorization policy.

The following example illustrates relation between policy DFA and our reference ReBAC model:

Example 4 (Policy DFA Accepting Strings/ReBAC Authorization Rules). Figure 2(b) shows a ReBAC policy DFA corresponding to the policy presented in Example 2. In particular, accepting strings $\langle owns \rangle$, $\langle treats.owns \rangle$, and $\langle assists.treats.owns \rangle$ correspond to the rules in the original policy. In this policy DFA, the access control rule $\langle owns \rangle$ is captured by the sequence of transitions $q_0 \xrightarrow{owns} q_1$ where q_1 is an accepting state. Similarly, the rules $\langle treats.owns \rangle$ and $\langle assists.treats.owns \rangle$ are, respectively, captured by the sequences $q_0 \xrightarrow{treats} q_3 \xrightarrow{owns} q_1$ and $q_0 \xrightarrow{assists} q_2 \xrightarrow{treats} q_3 \xrightarrow{owns} q_1$. Observe that the notation "$\neg treats$" represents two different transitions in this DFA corresponding to labels "*owns*" and "*assists*". The special transition $*$ shown in the figure matches all the labels, representing three different transitions corresponding to the input symbols "*owns*", "*treats*", and "*assists*".

Example 5 (Violation of Constrained DFA Model). Suppose, in Figure 2(b), the transition from state q_2 with the input symbol "*treats*" goes to state q_0 instead of q_3. In this case, any pattern beginning with the sub-pattern $\langle assists.treats \rangle$ will be accepted by this policy DFA. For instance, the patterns $\langle assists.treats.treats.owns \rangle$ and $\langle assists.treats.assists.treats.owns \rangle$ will become authorization rules according to this policy DFA, which is incorrect based on the ReBAC policy specified in Example 2.

4 BLACK-BOX POLICY LEARNING ARCHITECTURE

In this section, we present an overview of our black-box authorization learning architecture and processes. Given a ReBAC authorization system that can be queried for authorization decisions as a black box, our objectives are:

— to correctly infer the underlying authorization rules from the system; and
— to minimize the number of access control queries submitted to the system, since such queries are costly and cannot be exhaustively explored in practice.

In order to accomplish above mentioned objectives, we adopt ideas from the *MAT* framework [6] for query-based learning of finite automata, in particular, the Angluin's L* learning algorithm discussed in Section 2.1. In the rest of this section, we initially describe our objective of learning enforced authorization policy more concretely. Then, we describe our policy learning architecture,

including its components and the challenges in implementing each component. In addition, we present the workflow of our black-box architecture in terms of interactions between the components implementing the membership and equivalence queries and responses.

4.1 Preliminaries

Our objective is to learn the authorization behavior of a system in a black-box manner, particularly by submitting authorization inputs and observing the corresponding authorization decisions. In the access control terminology, this behavior is governed by the **policy decision point** (**PDP**) component in a system. Therefore, in our framework, PDP is considered the **system under learning** (**SUL**). As mentioned in Section 3.2, we will learn a formal model of the policy used by PDP as an automaton called *ReBAC policy DFA*.

We refer to the ReBAC policy enforced in SUL as the *ground-truth policy*, denoted by Φ_I. We denote the corresponding policy DFA by M_I. As our learning process forms a hypothesis policy DFA, we denote it by M_H (and Φ_H for the corresponding policy). In the following, we outline a set of assumed properties about the system graph, which is an input to our learning algorithm, relative to the ground-truth policy in SUL. These properties are required to ensure that the learner algorithm has a fair chance of observing the policy in action (based on the requests corresponding to the system graph).

Definition 5 (Input Properties). Given system graph G and policy Φ_I enforced in SUL, we say a pattern ϕ *applies* to an access request $\langle u, r\rangle$ in G whenever for some $\pi \in \Pi_{s,o}$, π matches ϕ. Then, we expect the following properties to hold in order for our learning algorithm to correctly infer Φ_I:

(1) Every rule in ground-truth policy is *uniquely applicable* to system graph. We say a rule (permitted pattern) $\phi \in \Phi_I$ is uniquely applicable to G whenever there exists at least an access request $\langle u, r\rangle$ in G such that only ϕ applies to $\langle u, r\rangle$. No other rule in Φ_I applies to $\langle u, r\rangle$.
(2) For a non-permitted pattern $\phi' \notin \Phi_I$, one of the following conditions holds:
— ϕ' is not present in G.
— ϕ' exists in G *and* there exists a denied access request $\langle u, r\rangle$ in G where ϕ' applies to $\langle u, r\rangle$.

4.2 Learning Architecture and Workflow

Figure 3 shows the architecture of our black-box approach for learning authorization rules. It comprises of four components, namely, *learner*, *mapper*, *equivalence oracle,* and SUL, where the latter three are contained within a larger abstract component called *teacher*. The label under each arrow shows (in bold) the type of query/request submitted and the response received. The label above each arrow shows an instance of query/response based on our running examples in Section 3.1 (Examples 2 and 3). In the following, we explain each of the components, including its role and assumptions about its design:

System Under Learning (SUL). SUL is the PDP that makes authorization decisions for given access requests based on its enforced policy, i.e., $U \times R \rightarrow \{\texttt{PERMIT}, \texttt{DENY}\}$.

Learner. The goal of the learner component is to actively infer the policy DFA (Definition 4) of ReBAC policy enforced in SUL by interacting with the teacher.

Mapper. The mapper component maps between SUL-understandable domain comprising of access requests and learner-understandable domain of relationship patterns. To be able to produce access requests corresponding to relationship patterns, we assume that the mapper has access to the system graph (Definition 1) that consists of relationships among users and

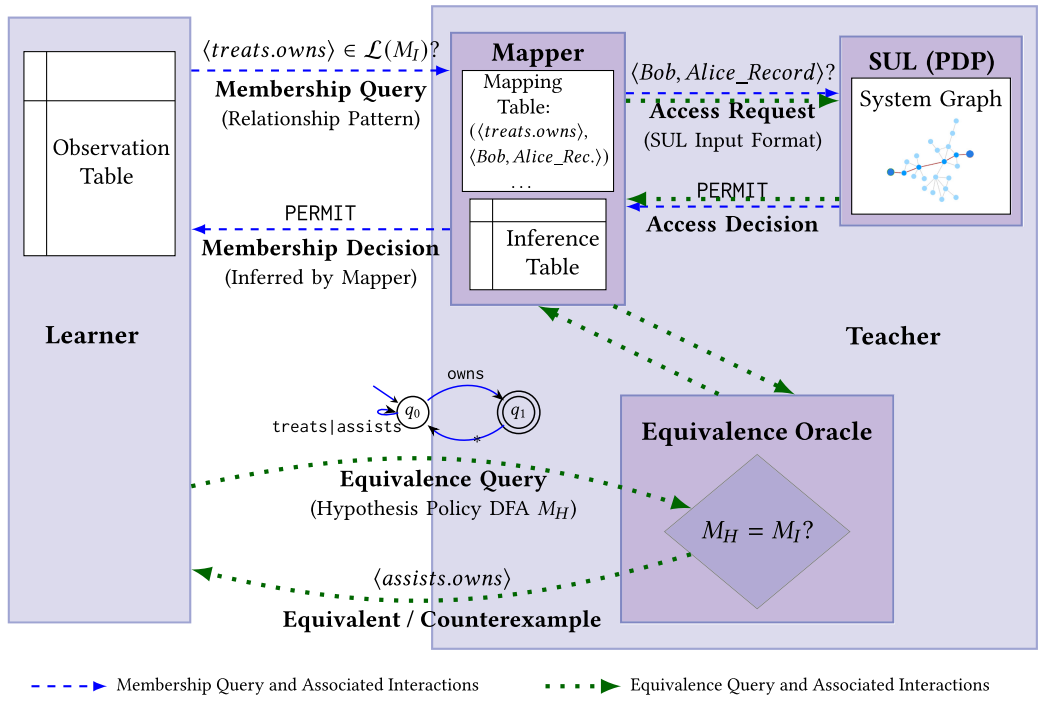


Fig. 3. Proposed Framework for Learning Access Control Policies from Black-Box Systems.

resources. We note that our focus is inferring the authorization policy imposed partly based on system graph; automated inference of the system graph in concrete application domains is of our interest as future work.

Equivalence Oracle. The equivalence oracle component has the important responsibility of evaluating the policy DFA inferred by the learner, and providing counterexamples if there are incorrect authorization rules in the inferred DFA.

Teacher. The teacher component encompasses mapper, equivalence oracle and SUL. The learner submits queries to the teacher and observes teacher's responses for inferring the initially unknown policy DFA.

In our architecture, the learner component has no prior information about the structure of the policy DFA that it has to infer. However, the learner can submit two types of queries to the teacher, namely *membership* and *equivalence* queries, which are defined as follows:

Membership Query asks whether a certain relationship pattern is permitted by SUL.

Equivalence Query checks whether the hypothesis policy DFA inferred so far by the learner correctly represents SUL.

In order to demonstrate the interaction between the components outlined above using membership and equivalence queries, we present the workflow of our learning algorithm at a high level in this section. In Section 5, we describe the working of the learner in detail. The learning procedure works in an iterative manner as follows. The learner submits membership queries about relationship patterns to the teacher. SUL within the teacher accepts only access requests (a pair of user and resource) as inputs. Therefore, the membership query cannot be processed directly by SUL. To tackle this problem, we introduce a *mapper* component between the learner and SUL that accepts

relationship patterns from the learner and produces responses by mapping them to authorization requests and interacting with SUL (upper part of Figure 3). It is important to correctly answer membership queries issued by the learner. However, potentially erroneous response could be generated when the mapper relies on an authorization decision from SUL to respond to the learner. This could happen because an authorization check by SUL tests if a user can access a resource; however, there could be multiple paths between that user and resource in the system graph, out of which only certain paths may be authorized according to the enforced policies. Section 4.3 describes the mapper component in more detail including its strategy for answering membership queries and its challenges.

After a few rounds of the process described above, based on its observations, the learner generates a hypothesis policy DFA and submits it to the *equivalence oracle* as an equivalence query. The equivalence oracle verifies the hypothesis with SUL by interacting with it (through testing authorization decisions of access requests). If the equivalence oracle returns *equivalent*, then the hypothesis is correct and the learner has correctly inferred the policy DFA from the system. Otherwise, the equivalence oracle replies with a counterexample pattern to the learner, and the learning procedure is resumed after processing the counterexample (lower part of Figure 3). It should be noted that the input to the equivalence oracle is a policy DFA whose language comprises a set of relationship patterns, whereas SUL accepts only access requests. The methodology used by the equivalence oracle to verify learner's hypothesis DFA using SUL is discussed in Section 4.4.

4.3 Overview of Mapper Design and Challenges

The mapper acts as a middleman between the learner and SUL. SUL is a policy decision point that can take an access request (i.e., $\langle user, resource \rangle$) as input and provide the corresponding access decision according to the application's enforced policies. However, as we are interested in learning a ReBAC policy DFA (Definition 4), our learner component is interested in membership queries in the form of relationship patterns. Therefore, the mapper is responsible for mapping between the abstract domain of relationship patterns and the concrete domain of access requests based on the given system graph. The existence of a mapper component in the proposed architecture is crucial for the efficiency of our learning approach, since a relationship pattern can, and usually does, correspond to potentially multiple access requests in the system graph. The concept of mapper has been previously shown to be useful in the context of MAT framework [4].

The mapper component will infer the response to a membership query based on its interactions with SUL, i.e., observing SUL's response to individual access requests. As part of this inference process, the mapper produces access requests corresponding to a membership query's relationship pattern and submit them to SUL for evaluation. Moreover, the mapper keeps a record of its current inferences about the responses of membership queries in an *inference table*.

We illustrate the functionality of the mapper component, particularly its role as a middleman between the learner and SUL, in the following example:

Example 6 (Learner, Mapper & SUL Interaction). The upper part of Figure 3 shows a sample membership query (relationship pattern $\langle treats.owns \rangle$), a corresponding access request ($\langle Bob, Alice\_Record \rangle$) that the mapper issues to SUL based on the system graph given in Example 1, and the response it receives from SUL (`PERMIT`). The mapper determines its response to the membership query based on SUL's response and its own inference table, and forwards it to the learner.

A significance of our design of mapper is to avoid submitting exhaustive access requests to SUL for evaluation. This design consideration inevitably results in mapper inferring responses to membership queries based on incomplete information. Some of such responses, thus, could be

uncertain. This uncertain behavior is limited to some positive responses. The mapper addresses those uncertain responses based on feedback it receives from equivalence oracle. We further explain the details of mapper's inference mechanism and its strategy for updating its inferences in Section 6.

Example 7 (Mapper's Uncertain Response). Consider the system graph given in Example 1 and the enforced authorizations specified in Example 3. Suppose the learner submits a membership query for the pattern $\langle assists.assists.owns \rangle$, and the mapper generates the access request $\langle Daniel, Bob\_Record \rangle$ corresponding to the query pattern and submits it to SUL for evaluation. Based on the given authorizations, SUL responds with `PERMIT` decision for the input request. However, there are two paths between Daniel and Bob's record in the system graph out of which only one is permitted as shown below:

$$Daniel \xrightarrow{assists} Eve \xrightarrow{treats} Bob \xrightarrow{owns} Bob\_Record \textbf{: (Permitted Pattern),}$$

$$Daniel \xrightarrow{assists} Carol \xrightarrow{assists} Bob \xrightarrow{owns} Bob\_Record \textbf{: (Non-Permitted Pattern).}$$

Due to incomplete information, the mapper may wrongly infer that the pattern $\langle assists.assists.owns \rangle$ is permitted, and forward that incorrect inference to the learner.

4.4 Overview of Equivalence Oracle and Queries

When the learner submits hypothesis policy DFA to the equivalence oracle, the latter finds if there is any authorization rule misjudged by the learner leading to over-assignment or under-assignment of permissions. The equivalence oracle is responsible to find a counterexample if any discrepancy exists between the hypothesis and SUL. If there is no such counterexample then the equivalence oracle approves the hypothesis generated by the learner. Thus, it provides crucial support to the learner for comprehending the correct set of access control rules from a black-box system.

In order to carry out its task of finding counterexamples, the equivalence oracle employs system graph information from the mapper, along with the ground-truth details about pattern authorizations from the mapper's inference table. Besides, the equivalence oracle may query SUL for access decisions to further validate authorized patterns in the hypothesis. A feature of our architecture is that the counterexamples from the equivalence oracle can also be used by the mapper for rectifying its wrong inferences on pattern authorizations (more details in Section 6).

It is indeed challenging to provide the equivalence oracle in many real-world learning settings. The equivalence oracle can be approximated using conformance testing. However, to the best of our knowledge, there has been no work in the literature on formal conformance testing of ReBAC policies. To cope with these challenges, in this article, we present two solutions for implementing the equivalence oracle based on various degrees of access space exploration. Section 7 explains the operation of the equivalence oracle in detail.

We illustrate operation of the equivalence oracle using an example equivalence query as follows:

Example 8 (Equivalence Query). The lower part of Figure 3 shows a sample equivalence query (hypothesis DFA) submitted by the learner and the associated response from equivalence oracle based on given system graph and authorizations (Examples 1 and 3). The equivalence oracle observes that the pattern $\langle assists.owns \rangle$ is not a rule since the access request $\langle Carol, Bob\_Record \rangle$ that matches the pattern is denied as per the enforced authorizations (recall that we are employing the deny-by-default strategy). However, this pattern is captured as a rule in the hypothesis DFA since the transition "*assists*" followed by transition "*owns*" reaches accepting state q_1, leading to permission over-assignment. Therefore, the equivalence oracle responds with counterexample $\langle assists.owns \rangle$ to learner's hypothesis.

5 LEARNER: INFERRING DFA FROM BLACK-BOX ACCESS CONTROL ENGINE

The goal of the learner is to learn an exact model of unknown ReBAC policy enforced in SUL. The policy enforcement is represented as a ReBAC policy DFA introduced in Section 3.2. In this section, we present detailed working of the learner component for actively learning enforced policy DFA.

5.1 Inference Model

Let M_I be the policy DFA, corresponding to ReBAC policy Φ_I, that is enforced by SUL. The learner does not know about the structure of M_I since it can only interact with the system as a black box. However, it can submit two types of queries to the teacher in the context of MAT framework, namely, *membership* and *equivalence* queries, which are defined as follows:

Definition 6 (Membership Query). The learner submits relationship pattern ϕ of its choice to the teacher and obtains whether $\phi \in \mathcal{L}(M_I)$.

A negative response to the membership query (i.e., $\phi \notin \mathcal{L}(M_I)$) is always correct. However, due to the design of the mapper component, a positive response to the membership query (i.e., $\phi \in \mathcal{L}(M_I)$) can be uncertain (i.e., the correct response might be negative). As we later show, our learner algorithm is still able to function correctly despite such uncertain behavior.

Definition 7 (Equivalence Query). The learner submits hypothesis DFA M_H and the teacher replies with a *yes* if $\mathcal{L}(M_H) = \mathcal{L}(M_I)$, or otherwise with a counterexample pattern $\phi \in \mathcal{L}(M_I) \Delta \mathcal{L}(M_H)$.

In the above definition, Δ denotes the symmetric difference operation. This means that the counterexample can be either a permitted relationship pattern that is missed by M_H (i.e., under assignment), or a non-permitted pattern that is incorrectly authorized by M_H (i.e., over assignment).

In order to facilitate the discussion of our learning approach and its analysis, we present the following simple running example in the context of an electronic medical records system.

Example 9 (Running Example). Figure 4 illustrates the process of actively learning the policy DFA depicted in Example 4 based on system graph in Example 1 and implemented authorizations in Example 3. Here, "o", "t", and "a" are abbreviations for "owns", "treats", and "assists", respectively. Also, we use the wildcard character $*$ to match any element in the set of relationship labels $\{owns, treats, assists\}$. For instance, instead of having three rows $\langle owns.owns\rangle$, $\langle owns.treats\rangle$, and $\langle owns.assists\rangle$, we abbreviate them as $\langle owns.*\rangle$ in the table for brevity. The upcoming examples will discuss the steps of the process.

5.2 Observation Table

Observation table is the primary data structure utilized by the learner to accommodate knowledge about a finite collection of relationship patterns, distinguishing them as accepted or not accepted.

Definition 8 (Observation Table). Observation Table is a triple $\langle O_S, O_E, O_\tau\rangle$, where:

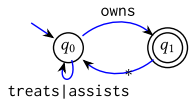
- $O_S \subseteq \mathcal{R}$ is a non-empty finite prefix-closed set of relationship patterns.
- $(O_S \,.\, L)$ is the dynamic part of the table that contains the elements produced from pairwise concatenation of the elements in sets O_S and L.
- $O_E \subseteq \mathcal{R}$ is a non-empty finite suffix-closed set of pattern.
- O_τ is a finite function $((O_S \cup O_S \,.\, L) \,.\, O_E) \rightarrow \{0, 1\}$ that maps patterns resulting from concatenation of patterns in O_S and $(O_S \,.\, L)$ with patterns in O_E to 0 or 1. If the response to a membership query ϕ submitted by the learner is `PERMIT`, then the learner sets $O_\tau(\phi) = 1$ and expects that ϕ corresponds to a rule in Φ_I; else, $O_\tau(\phi) = 0$ (if the response is `DENY`).

O_τ	σ
σ	0
$\langle owns \rangle$	1
$\langle treats \rangle$	0
$\langle assists \rangle$	0

(a) $O_S = O_E = \{\sigma\}$.

O_τ	σ
σ	0
$\langle owns \rangle$	1
$\langle treats \rangle$	0
$\langle assists \rangle$	0
$\langle owns.* \rangle$	0

(b) $O_S = \{\sigma, \langle owns \rangle\}$, $O_E = \{\sigma\}$.



(c) M_{H1}, First Hypothesis of Learner. Counterexample = $\langle assists.owns \rangle$.

O_τ	σ
σ	0
$\langle o \rangle$	1
$\langle a \rangle$	0
$\langle a.o \rangle$	0
$\langle t \rangle$	0
$\langle o.* \rangle$	0
$\langle a.t \rangle$	0
$\langle a.a \rangle$	0
$\langle a.o.* \rangle$	0

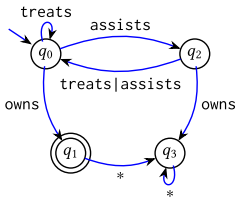
(d) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma\}$.

O_τ	σ	$\langle o \rangle$
σ	0	1
$\langle o \rangle$	1	0
$\langle a \rangle$	0	0
$\langle a.o \rangle$	0	0
$\langle t \rangle$	0	1
$\langle o.* \rangle$	0	0
$\langle a.t \rangle$	0	1
$\langle a.a \rangle$	0	1
$\langle a.o.* \rangle$	0	0

(e) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle\}$.

O_τ	σ	$\langle o \rangle$	$\langle a.o \rangle$
σ	0	1	0
$\langle o \rangle$	1	0	0
$\langle a \rangle$	0	0	1
$\langle a.o \rangle$	0	0	0
$\langle t \rangle$	0	1	0
$\langle o.* \rangle$	0	0	0
$\langle a.t \rangle$	0	1	0
$\langle a.a \rangle$	0	1	0
$\langle a.o.* \rangle$	0	0	0

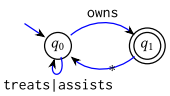
(f) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle, \langle assists.owns \rangle\}$.



(g) M_{H2}, Second Hypothesis of Learner. Counterexample = $\langle assists.assists.owns \rangle$.

O_τ	σ
σ	0
$\langle o \rangle$	1
$\langle t \rangle$	0
$\langle a \rangle$	0
$\langle o.* \rangle$	0

(h) $O_S = \{\sigma, \langle owns \rangle\}$, $O_E = \{\sigma\}$.



O_τ	σ
σ	0
$\langle o \rangle$	1
$\langle a \rangle$	0
$\langle a.o \rangle$	0
$\langle t \rangle$	0
$\langle o.* \rangle$	0
$\langle a.t \rangle$	0
$\langle a.a \rangle$	0
$\langle a.o.* \rangle$	0

(i) M_{H3}, Third Hypothesis of Learner. Counterexample = $\langle assists.owns \rangle$. $O_E = \{\sigma\}$.

O_τ	σ	$\langle o \rangle$
σ	0	1
$\langle o \rangle$	1	0
$\langle a \rangle$	0	0
$\langle a.o \rangle$	0	0
$\langle t \rangle$	0	1
$\langle o.* \rangle$	0	0
$\langle a.t \rangle$	0	1
$\langle a.a \rangle$	0	0
$\langle a.o.* \rangle$	0	0

(j) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle\}$.

O_τ	σ	$\langle o \rangle$	$\langle t.o \rangle$
σ	0	1	1
$\langle o \rangle$	1	0	0
$\langle a \rangle$	0	0	1
$\langle a.o \rangle$	0	0	0
$\langle t \rangle$	0	1	0
$\langle o.* \rangle$	0	0	0
$\langle a.t \rangle$	0	1	0
$\langle a.a \rangle$	0	0	0
$\langle a.o.* \rangle$	0	0	0

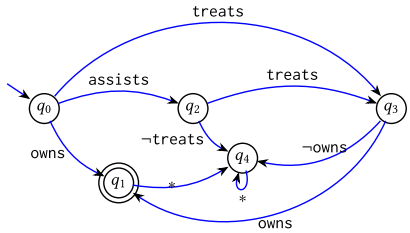
(k) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle\}$.

O_τ	σ	$\langle o \rangle$	$\langle t.o \rangle$
σ	0	1	1
$\langle o \rangle$	1	0	0
$\langle a \rangle$	0	0	1
$\langle a.o \rangle$	0	0	0
$\langle t \rangle$	0	1	0
$\langle o.* \rangle$	0	0	0
$\langle a.t \rangle$	0	1	0
$\langle a.a \rangle$	0	0	0
$\langle a.o.* \rangle$	0	0	0

(l) $O_S = \{\sigma, \langle owns \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle, \langle treats.owns \rangle\}$.

O_τ	σ	$\langle o \rangle$	$\langle t.o \rangle$
σ	0	1	1
$\langle o \rangle$	1	0	0
$\langle t \rangle$	0	1	0
$\langle a \rangle$	0	0	1
$\langle a.o \rangle$	0	0	0
$\langle o.* \rangle$	0	0	0
$\langle t.o \rangle$	1	0	0
$\langle t.t \rangle$	0	0	0
$\langle t.a \rangle$	0	0	0
$\langle a.t \rangle$	0	1	0
$\langle a.a \rangle$	0	0	0
$\langle a.o.* \rangle$	0	0	0

(m) $O_S = \{\sigma, \langle owns \rangle, \langle treats \rangle, \langle assists \rangle, \langle assists.owns \rangle\}$, $O_E = \{\sigma, \langle owns \rangle, \langle treats.owns \rangle\}$.



(n) M_{H4}, Fourth and Correct Hypothesis of Learner.

Fig. 4. Illustration of learning algorithm for inferring policy DFA in Figure 2. ("o", "t", "a", and "$*$" referring to "owns", "treats", "assists", and any label in $L = \{owns, treats, assists\}$, respectively.)

In the above definition, a prefix (suffix)-closed set means that every prefix (suffix) of every member of the set is also a member of the set. Also, the dot operator (".") denotes pairwise concatenation of the elements of two sets. We also use the same notation to indicate concatenation of two singular patterns, and pairwise concatenation of a singular pattern with the elements of a set.

An observation table can be visualized as table using $(O_S \cup O_S \,.\, L)$ as rows and O_E as columns. The entry for row s and column e is $O_\tau(s.e)$. The intuition behind observation table is that the rows labeled by elements of O_S are candidates for states of the automaton being constructed, and rows labeled by elements of $(O_S \,.\, L)$ are employed to establish the transition function. Columns labeled by elements of O_E correspond to distinguishing experiments for the states in the automaton. Thus, an observation table is eventually used to build a deterministic finite-state automaton.

Let $\text{row}(\phi_s)$ denote the row for pattern ϕ_s in the observation table (i.e., $\phi_s \in (O_S \cup O_S \,.\, L)$). Formally, $\text{row}(\phi_s)$ is defined as the finite function f from O_E to $\{0, 1\}$ defined as $f(\phi_e) = O_\tau(\phi_s \,.\, \phi_e)$, where $\phi_e \in O_E$. Accordingly, we define *closed* and *consistent* observation tables as follows:

Definition 9 (Closed Observation Table). An observation table is called *closed* if for every pattern ϕ' in $(O_S \,.\, L)$ there is a pattern ϕ in O_S such that $\text{row}(\phi') = \text{row}(\phi)$.

Definition 10 (Consistent Observation Table). An observation table is called *consistent* if whenever $\phi_1, \phi_2 \in O_S$ and $\text{row}(\phi_1) = \text{row}(\phi_2)$ then $\forall l \in L$, $\text{row}(\phi_1 \,.\, l) = \text{row}(\phi_2 \,.\, l)$.

Checking if an observation table is closed ensures that the transitions lead to states defined based on the table. Checking if an observation table is consistent ensures that the transitions are consistent. Ensuring both these conditions enable us to construct a valid hypothesis policy DFA.

Example 10 (Observation Table). Figure 4 shows several observation tables during the learning process in our running examples (except Figures 4(c), (h), (j) and (n), which are hypothesis policy DFAs). For instance, in the observation table in Figure 4(m), O_S consists of patterns σ (the empty pattern), $\langle o \rangle$, $\langle a \rangle$, $\langle a.o \rangle$, and $\langle t \rangle$, while O_E consists of patterns σ, $\langle o \rangle$ and $\langle t.o \rangle$. In addition, we have $(O_S \,.\, L)$ by concatenating the patterns in sets $O_S = \{\sigma, \langle o \rangle, \langle a \rangle, \langle a.o \rangle, \langle t \rangle\}$ and $L = \{o, t, a\}$. For instance, for pattern $\langle t \rangle \in O_S$ we have the patterns $\langle t.o \rangle$, $\langle t.t \rangle$ and $\langle t.a \rangle$ in the set $(O_S \,.\, L)$. The 0/1 value for an entry $O_\tau(s.e)$ indicates the membership response for the pattern $(s.e)$ produced by concatenating the corresponding row "s" and column "e" indicators. For instance, based on Example 2, pattern $\langle t.o \rangle$ is an authorization rule in the ReBAC policy and hence, $O_\tau(\langle t.o \rangle) = 1$. Note that, in Figures 4(e) and (f), $O_\tau(\langle a.a.o \rangle) = 1$ even though the pattern $\langle a.a.o \rangle$ is not an authorization rule in Example 2. Thus, the entry $O_\tau(\langle a.a.o \rangle)$ is erroneous (due to uncertain membership response for that pattern).

5.3 Learner Methodology

The learner conducts active learning following Algorithm 1, discussed in this section. Later in Section 5.5, we will discuss Algorithm 2 that details our counterexample processing approach. At any time during the learning process, the learner component maintains an observation table $\langle O_S, O_E, O_\tau \rangle$.

At the beginning, learner populates the observation table with $O_S = O_E = \{\sigma\}$, where σ is the empty pattern (Lines 1 and 2). Then, to determine O_τ, the learner submits membership queries to the teacher for σ and each $l \in L$. This initial observation table may or may not be closed and consistent.

In the main loop, the learner checks whether the current table $\langle O_S, O_E, O_\tau \rangle$ is closed and consistent (Line 4). If the table is not closed, the learner looks for $\phi' \in O_S$ and $l \in L$ such that $\text{row}(\phi' \,.\, l) \neq \text{row}(\phi)$ for all $\phi \in O_S$. It then extends O_S by adding the pattern $(\phi' \,.\, l)$ and augments the observation table through membership queries for missing entries (Lines 5–8). If not

ALGORITHM 1: Learning a ReBAC Policy DFA in Black-Box Manner

Result: Policy DFA M_H corresponding to ReBAC policy Φ_I enforced by SUL.

1 Initialize O_S and O_E to $\{\sigma\}$;
2 Construct initial observation table $\langle O_S, O_E, O_\tau \rangle$ through membership queries for σ and each $l \in L$;
3 **repeat**
4 **while** $\langle O_S, O_E, O_\tau \rangle$ *is not closed or not consistent* **do**
5 **if** $\langle O_S, O_E, O_\tau \rangle$ *is not closed* **then**
6 Determine $\phi' \in O_S$ and $l \in L$ such that $\forall \phi \in O_S$, row$(\phi' \,.\, l) \neq$ row(ϕ);
7 $O_S \leftarrow O_S \cup \{(\phi' \,.\, l)\}$;
8 Extend O_τ by issuing missing membership queries for patterns corresponding to $(\phi' \,.\, l) \,.\, O_E$ and $(\phi' \,.\, l) \,.\, L \,.\, O_E$;
9 **if** $\langle O_S, O_E, O_\tau \rangle$ *is not consistent* **then**
10 Determine $\phi_1, \phi_2 \in O_S$, $\phi_e \in O_E$ and $l \in L$ such that row(ϕ_1) = row(ϕ_2) but $O_\tau(\phi_1 \,.\, l \,.\, \phi_e) \neq O_\tau(\phi_2 \,.\, l \,.\, \phi_e)$;
11 $O_E \leftarrow O_E \cup \{(l \,.\, \phi_e)\}$;
12 Extend O_τ by issuing missing membership queries for patterns corresponding to $(O_S \cup O_S \,.\, L) \,.\, (l \,.\, \phi_e)$;
13 Construct hypothesis M_H corresponding to the closed and consistent table $\langle O_S, O_E, O_\tau \rangle$;
14 Issue an equivalence query corresponding to hypothesis DFA M_H to the teacher;
15 **if** *the teacher replies with a counterexample pattern* ϕ **then**
16 $\langle O_S, O_E, O_\tau \rangle \leftarrow$ processCounterexample$(\phi, \langle O_S, O_E, O_\tau \rangle)$;
17 **until** *the teacher replies* equivalent *to the query about hypothesis* M_H;
18 **return** M_H;

consistent, the learner finds $\phi_1, \phi_2 \in O_S$, $\phi_e \in O_E$, and $l \in L$ such that row(ϕ_1) = row(ϕ_2) but $O_\tau(\phi_1 \,.\, l \,.\, \phi_e) \neq O_\tau(\phi_2 \,.\, l \,.\, \phi_e)$. Then, the learner adds the pattern $(l \,.\, \phi_e)$ to O_E and populates O_τ values corresponding to $(O_S \cup O_S \,.\, L) \,.\, (l \,.\, \phi_e)$ through membership queries for missing elements (Lines 9–12).

As soon as the observation table $\langle O_S, O_E, O_\tau \rangle$ becomes closed and consistent, the learner constructs a conjecture policy DFA M_H corresponding to the current table (Line 13). This process is discussed in detail in Section 5.4. The learner then issues an equivalence query to the teacher to determine whether its hypothesis DFA M_H is correct (Line 14). If the equivalence oracle within teacher confirms correctness of the hypothesis and returns *equivalent* then the learner successfully terminates with M_H being the final output (Lines 17 and 18). On the contrary, if the equivalence oracle replies with a counterexample pattern ϕ, then our algorithm calls the *processCounterexample* routine with ϕ and the current observation table as inputs (Lines 15 and 16). The *processCounterexample* routine (discussed later in detail in Section 5.5 and Algorithm 2) resolves the error caused due to the pattern ϕ and returns an updated observation table that addresses the counterexample. The learner then repeats the entire procedure on this new observation table $\langle O_S, O_E, O_\tau \rangle$. The working of the equivalence oracle component is discussed in detail in Section 7.

In summary, the learner issues queries about membership of relationship patterns to the teacher and updates its observation table. Once observation table becomes closed and consistent, the learner submits hypothesis policy DFA to teacher. If the teacher replies with a counterexample, then the learner updates its observation table to account for the erroneous pattern, and repeats the whole process. Else, the learner successfully terminates with right hypothesis.

Example 11 (Closed, Consistent Observation Tables). Figures 4(b), (f), (i), and (m) demonstrates closed and consistent observation tables. Initially, our learning algorithm submits membership

queries for the patterns σ, $\langle o \rangle$, $\langle t \rangle$, and $\langle a \rangle$. The initial observation table shown in Figure 4(a) is not closed since row($\langle o \rangle$) is distinct from row(σ). So, our algorithm moves the pattern $\langle o \rangle$ to O_S and then queries the patterns $\langle o.o \rangle$, $\langle o.t \rangle$, and $\langle o.a \rangle$ to construct the table in Figure 4(b). Similarly, the observation table in Figure 4(l) is not closed since row($\langle t \rangle$) is distinct from all rows in O_S. So, our algorithm moves the pattern $\langle t \rangle$ to O_S and submits membership queries for the missing entries to obtain the table in Figure 4(m).

The observation table in Figure 4(d) is not consistent since row(σ) = row($\langle a \rangle$) but row($\langle o \rangle$) ≠ row($\langle a.o \rangle$) (note that $\sigma.\langle o \rangle = \langle o \rangle$). Thus, our algorithm adds pattern $\langle o \rangle$ to O_E, and queries patterns $\langle t.o \rangle$, $\langle o.*.o \rangle$, $\langle a.t.o \rangle$, $\langle a.a.o \rangle$, and $\langle a.o.*.o \rangle$ to construct the observation table shown in Figure 4(e). Observe that this causes a membership error corresponding to the pattern $\langle a.a.o \rangle$ in the table (which will be resolved during counterexample processing). Similarly, the observation table in Figure 4(e) is not consistent since row($\langle a \rangle$) = row($\langle a.o \rangle$) but row($\langle a.a \rangle$) ≠ row($\langle a.o.a \rangle$) (by replacing $*$ in $\langle a.o.* \rangle$ with a). So, our algorithm adds pattern $\langle a.o \rangle$ to O_E and constructs the observation table shown in Figure 4(f) by submitting membership queries for the missing entries. Note, here, that we could have alternatively added the pattern $\langle t.o \rangle$ to O_E since row($\langle a \rangle$) = row($\langle a.o \rangle$) but row($\langle a.t \rangle$) ≠ row($\langle a.o.t \rangle$).

5.4 Constructing Hypothesis Policy DFA

Given a closed, consistent observation table, we want to build a hypothesis DFA that conforms with the ReBAC policy DFA specification stated in Definition 4. In particular, the language of the constructed hypothesis DFA should be finite.

Let Q be the set of states, q_0 be the initial state, F be the accepting (final) states, and δ be the transition function of a DFA. Then, we construct an initial hypothesis DFA M_H corresponding to a closed, consistent observation table $\langle O_S, O_E, O_\tau \rangle$ and over the alphabet set L as follows:

Q = {row(ϕ): $\phi \in O_S$};
q_0 = row(σ);
F = {row(ϕ): $\phi \in O_S$ and $O_\tau(\phi) = 1$};
δ(row(ϕ), l) = row(ϕ . l).

However, the language of the initial hypothesis DFA constructed as above may not be finite. In particular, considering the hypothesis DFA as a directed graph, we may have reachable cycles on some paths from the initial state to final states. Such cycles will lead to an infinite language for the DFA. We remove those cycles by redirecting them to a dead state in the DFA. We follow a two-step process, namely, identifying the cycles and identifying the nodes' reachability to accepting states. The following steps detect and prune any cycle on the path to an accepting state in hypothesis DFA M_H, where the DFA structure is considered as a graph:

(1) Starting from the initial state q_0 of M_H, traverse the entire DFA in a depth-first manner.
(2) Identify the set of *back edges* B in the depth-first tree, i.e., those edges (u, v) that connect a vertex u to an ancestor vertex v in the tree, including self-loops. The edges in B are responsible for cycles in M_H.
(3) For every back edge $(u, v) \in B$, determine if any of the accepting states of M_H is reachable from v. If true, then mark the edge (u, v) for removal.
(4) After marking all edges that are responsible for creating cycles in the accepting strings, direct each of those marked edges to a dead state in the DFA M_H. To identify a dead state in M_H:
— Determine the connected components in M_H.
— Select the component containing no accepting state and mark its nodes as dead states.
— If no such state exists, then create a dead state.

Example 12 (Constructing Hypothesis Policy DFA). Figures 4(c), (g), (i), and (n) show different hypothesis DFAs (named, M_{H1}, M_{H2}, M_{H3}, and M_{H4}, respectively) that are constructed during the learning process. These hypothesis DFAs are constructed from the closed, consistent observation tables in Figures 4(b), (f), (h) and (m), respectively. To further clarify the process of automaton construction, we focus on how the final hypothesis DFA M_{H4} is constructed from the observation table in Figure 4(m). For each distinct row in O_S corresponding to patterns σ, $\langle o \rangle$, $\langle a \rangle$, $\langle t \rangle$, and $\langle a.o \rangle$, we have states q_0, q_1, q_2, q_3, and q_4, respectively. We have only one final state q_1 corresponding to pattern $\langle o \rangle$ since $O_\tau(\phi.\sigma) = 1$ only for row($\langle o \rangle$), where $\phi \in O_S$. As an example of a transition in DFA M_{H4}, starting from state q_0 corresponding to row(σ) and following label "o" would lead to state q_1 corresponding to row($\langle o \rangle$).

5.5 Processing Counterexamples

Counterexamples are provided by the teacher to the learner when the hypothesis policy DFA M_H does not match ground-truth DFA M_I. In terms of an authorization policy, this means that the learner's hypothesis fails to correctly capture certain authorization rules. This could be caused in our learner due to two reasons:

— faults due to lack of exploring distinguishing accepted patterns by the learner;
— faults induced because of incorrect inference of relationship pattern authorizations by the mapper.

In other words, errors in our hypothesis can be due to either incomplete or incorrect observations about authorization patterns. Our algorithm follows the original L* algorithm to tackle the errors that are due to incomplete observations. However, the errors due to incorrect observations need to be treated differently. Based on these two sources of errors, our learning algorithm deals with two types of counterexamples, which we call `L*` *and* `uncertain` *counterexamples, respectively.*

Definition 11 (`L*` *Counterexample).* An `L*` *counterexample* is about a relationship pattern that *does not exist* in the current observation table. Given a submitted hypothesis M_H and its corresponding table, the received counterexample ϕ is an `L*` counterexample if $\phi \notin ((O_S \cup O_S \,.\, L) \,.\, O_E)$.

Definition 12 (`uncertain` *Counterexample).* An `uncertain` *counterexample* is about a pattern that *exists* in the current observation table. Given a submitted hypothesis M_H and its corresponding table, the received counterexample ϕ is an `uncertain` counterexample if $\phi \in ((O_S \cup O_S \,.\, L) \,.\, O_E)$. Furthermore, it is always the case that $O_\tau(\phi)$ is incorrectly recorded as 1, while it should be 0.

As mentioned in the above definition, the `uncertain` counterexamples are always about relationship patterns that have been incorrectly authorized by the submitted hypothesis. This issue will be further discussed in Section 6.

Algorithm 2 describes our approach to process counterexamples in a hypothesis DFA M_H submitted by the learner. The input to the *processCounterexample* routine are the current observation table $\langle O_S, O_E, O_\tau \rangle$ corresponding to DFA M_H and counterexample pattern ϕ received from the equivalence oracle. The *processCounterexample* routine returns an updated observation table. In the following, we describe the different steps in our counterexample processing algorithm:

Determining Counterexample Type. During counterexample processing, our algorithm first checks if counterexample ϕ exists in current table $\langle O_S, O_E, O_\tau \rangle$ or not (Lines 1 and 13). If it exists, then as mentioned earlier, ϕ is an `uncertain` counterexample. Otherwise, ϕ is an `L*` counterexample.

ALGORITHM 2: processCounterexample(ϕ, $\langle O_S, O_E, O_\tau \rangle$); Processing `uncertain`/L* Counterexample

Inputs: Counterexample pattern ϕ and current observation table $\langle O_S, O_E, O_\tau \rangle$.
Result: Updated observation table $\langle O_S, O_E, O_\tau \rangle$ after processing counterexample ϕ.

```
1  if φ ∈ ((O_S ∪ O_S . L) . O_E) then                    // Process uncertain counterexample
2      O_τ(φ) ← 0;                                       // Correct pattern φ entries in current observation table
3      for i = 1 to len(φ) + 1 do                        // Identify shortest prefix of φ in O_S where suffix in O_E
4          φ_pre ← φ[1 . . . i − 1];                     // Split pattern φ into prefix φ_pre
5          φ_suf ← φ[i . . . len(φ)];                    // and suffix φ_suf
6          if φ_suf ∈ O_E and φ_pre ∈ O_S then
7              break;
8      if φ_pre = σ then
9          Reset ⟨O_S, O_E, O_τ⟩ to the initial table by following steps 1–2 in Algorithm 1;
10     else
11         Remove any pattern from O_S with φ_pre as its prefix;
12         O_E ← {σ};
13 else                                                  // Process L* counterexample
14     Add φ and all of its prefixes to O_S;
15     Extend O_τ to ((O_S ∪ O_S . L) . O_E) through membership queries for missing entries;
```

Processing L* *Counterexample.* To process an L* counterexample, our algorithm adds ϕ along with all its prefixes to O_S (if they do not already exist in O_S). Then, it populates O_τ values corresponding to $(\Phi \,.\, O_E)$ and $((\Phi \,.\, L) \,.\, O_E)$ by submitting membership queries for missing entries (Lines 14 and 15). Here, Φ is the set consisting of counterexample ϕ and any of its prefixes not already present in O_S.

Processing `uncertain` *Counterexample.* This involves the following three steps:

— *Fixing observation table entries.* Our algorithm first corrects the entries in current observation table (Line 2) corresponding to pattern ϕ. In particular, we set $O_\tau(\phi) = 0$ (which has been previously "1" due to the mapper's uncertain membership response). If we visualize observation table as a table with rows $(O_S \cup O_S \,.\, L)$, columns O_E, and $O_\tau(s.e)$ as entry for row s and column e, then multiple entries of the table will be affected by this correction operation.

— *Identifying shortest prefix in O_S.* Then, our algorithm determines the shortest prefix of ϕ that is in O_S such that the suffix is in O_E (Lines 3–7). To this end, our algorithm splits ϕ into a prefix ϕ_{pre} and a suffix ϕ_{suf} and checks whether $\phi_{pre} \in O_S$ *and* $\phi_{suf} \in O_E$, iteratively over the length of ϕ. At the beginning of the loop ϕ_{pre} is empty and $\phi_{suf} = \phi$, and vice-versa at the ending. In the algorithm, $\phi[m \ldots n]$ indicates the substring of ϕ from index m to index n.

— *Removing potential erroneous observations.* Once, we identify the shortest prefix ϕ_{pre}, we add it to the set $(O_S \,.\, L)$ and remove all patterns from O_S that have ϕ_{pre} as the prefix (Lines 10–12). Note that any pattern with ϕ_{pre} as the prefix will be removed from $(O_S \,.\, L)$ as well. Also, we remove all columns (except σ) from O_E. The intuition is that we backtrack to a policy DFA, which did not have the counterexample ϕ, and then remove all states and transitions that succeed the pattern ϕ in the DFA. If the shortest prefix is the empty pattern σ (i.e., there is an error in the start state), then we revert to the initial observation table (Lines 8 and 9).

Subsequently, *processCounterexample* returns to Algorithm 1 with an updated table $\langle O_S, O_E, O_\tau \rangle$. Then, the entire learning process is repeated on the new observation table, i.e., we ensure that the table is closed and consistent, and issue the corresponding equivalence query (hypothesis DFA).

*Example 13 (*L* vs.* uncertain *Counterexamples Processing).* Figure 4(d) illustrates the result of processing an L*-type counterexample, namely $\langle a.o \rangle$, in hypothesis DFA M_{H1} (Figure 4(c)). M_{H1} is not the correct DFA for the policy in Example 2 because the pattern $\langle a.o \rangle$ is not in the policy, but is accepted by M_{H1}. Therefore, the equivalence oracle returns counterexample $\langle a.o \rangle$. To process counterexample $\langle a.o \rangle$, our algorithm adds patterns $\langle a \rangle$ and $\langle a.o \rangle$ to O_S (the empty pattern σ is already in O_S), and queries for patterns $\langle a.t \rangle$, $\langle a.a \rangle$, $\langle a.o.o \rangle$, $\langle a.o.t \rangle$, and $\langle a.o.a \rangle$ to construct the observation table in Figure 4(d).

Figure 4(h) shows the result of processing an uncertain counterexample, namely, $\langle a.a.o \rangle$, in hypothesis DFA M_{H2} (Figure 4(g)). Because of uncertain membership response, the pattern $\langle a.a.o \rangle$, which is not a rule according to Example 2, is accepted by M_{H2}. After fixing the value of $O_\tau(\langle a.a.o \rangle)$, our algorithm determines the shortest prefix of the counterexample that exists in O_S, which is $\langle a \rangle$ in this example, where the suffix $\langle a.o \rangle$ is in O_E. We then add $\langle a \rangle$ to the set $(O_S \,.\, L)$ and remove all rows with pattern $\langle a \rangle$ as the prefix, i.e., row($\langle a.o \rangle$), row($\langle a.t \rangle$), row($\langle a.a \rangle$), and row($\langle a.o.* \rangle$). In addition, all patterns except σ are removed from O_E. Finally, we obtain the observation table shown in Figure 4(h).

Although not illustrated by the above example, we note that our counterexample processing is also capable of detecting erroneous patterns, i.e., processing L*-type counterexamples that are actual rules in the policy but not accepted by the hypothesis DFA.

6 MAPPER: ACCESS SPACE ABSTRACTION

This section describes the design and working of the mapper component that mediates between the learner and SUL, for the learner to infer a policy DFA. The mapper considers concrete domain of access requests when interacting with SUL, while considering abstract domain of relationship patterns when receiving membership queries from the learner.

6.1 Design of the Mapper Component

To be able to perform its function, the mapper component contains two data structures, namely, the *mapping table* and the *inference table*. Informally, the former is used for storing the mapping between relationship patterns and access requests according to the system graph, whereas the latter is used for storing the mapper's current inferences about pattern authorizations based on its interactions with SUL. In the following, we describe each of these data structures in detail:

6.1.1 Mapping Table. The *mapping table* stores the association between the learner's abstract domain of relationship patterns and SUL's concrete domain of access requests based on a system graph. Given the system graph $G(V, E)$, the concrete domain of access requests (used by SUL), denoted by $C \subseteq V \times V$, comprises the collection of access requests $\langle u \in U, r \in R \rangle$. The abstract domain of relationship patterns (used by the learner through membership queries) comprises all possible relationship patterns $\mathcal{R}$ (Definition 2). We formally define the mapping table as follows:

Definition 13 (Mapping Table). The mapping table P is a tuple $\langle P_\phi, P_C \rangle$. $P_\phi \subseteq \mathcal{R}$ is the set of relationship patterns in the system graph. $P_C \subseteq P_\phi \times C$ is a binary relation that maps each pattern $\phi \in P_\phi$ with its corresponding access requests $\langle u, r \rangle \in C$ such that $\exists \pi \in \Pi_{u,r}$, π matches pattern ϕ.

For obtaining access requests $\langle u, r \rangle$ corresponding to a given relationship pattern ϕ, we employ the following procedure. Using a graph traversal strategy such as breadth-first search, we systematically explore system graph $G(V, E)$ for various relationship patterns defined on set of labels L. For every path $\pi \in \Pi_{u,r}$ between user $u \in U$ and resource $r \in R$ encountered during graph traversal, we record the mapping between ϕ (relationship pattern associated with π) and $\langle u, r \rangle$ into the

mapper's mapping table P. Since this procedure is repeated for all V nodes in the directed system graph, the time complexity for obtaining the mapping table containing relationship patterns and their associated access requests is $O(V^3)$. For tackling large graphs with several high-degree nodes, our algorithm is provisioned with a user-specified constraint indicating the maximal length of relationship pattern in SUL's enforced policy. So, the graph traversal procedure is length-limited.

6.1.2 Inference Table. The inference table records the mapper's current inferences about responses to the learner's membership queries. Formally, we define the inference table as follows:

Definition 14 (Inference Table). The inference table $\mathcal{I}$ is a tuple $\langle \mathcal{I}_\phi, \mathcal{I}_{\widehat{\phi}}, \mathcal{I}_\tau \rangle$. $\mathcal{I}_\phi \subseteq \mathcal{R}$ is a set of relationship patterns (corresponding to membership queries) for which the mapper has inferred a response. $\mathcal{I}_{\widehat{\phi}} \subseteq \mathcal{I}_\phi$ indicates those patterns whose membership decision inferred by the mapper is certainly correct (with respect to the ReBAC policy Φ_I enforced by SUL). Finally, function $\mathcal{I}_\tau : \mathcal{I}_\phi \rightarrow \{\texttt{PERMIT}, \texttt{DENY}\}$ maps each $\phi \in \mathcal{I}_\phi$ to its corresponding inferred response by the mapper.

The mapper infers the response to a membership query based on its interactions with SUL and stores them in its inference table. As part of the inference process, the mapper produces an access request $\langle u, r \rangle \in C$ associated with the membership query's relationship pattern $\phi \in \mathcal{R}$, such that $(\phi, \langle u, r \rangle) \in P_C$, and submits it to SUL for determining the access decision. Ideally, the mapper should return correct responses to the learner's membership queries. However, the design of our mapper is such that the access space of SUL is not exhaustively explored. So, it is inevitable that the mapper would make inferences about responses to membership queries based on incomplete information. Thus, some of the mapper's responses to membership queries could be uncertain. This uncertain behavior is restricted to some positive responses. The inference mechanism used by the mapper and its strategy for tackling uncertain responses will be discussed in Section 6.2.

6.2 Working of the Mapper Component

A major design consideration of our mapper is to avoid exhaustive exploration of SUL access space. To this end, the mapper infers its responses to membership queries based on minimal interaction with SUL. This design may result in uncertain inferences since a relationship pattern usually corresponds to multiple access requests in a system graph. The mapper also caches its inferences in the inference table. In the following, we describe various functions of the mapper in our learning architecture, including interacting with the other components and building its inference table:

Interaction with the Learner. At the beginning of our learning process (given in Algorithm 1), the inference table $\mathcal{I}$ is empty, and it is augmented as queries are generated by the learner. Whenever the learner submits a membership query about pattern ϕ, the mapper checks its inference table for decision corresponding to that pattern. Specifically, the mapper returns $\mathcal{I}_\tau(\phi)$ if $\phi \in \mathcal{I}_\phi$. If the mapper does not have the inference, then it will interact with SUL to answer the membership query.

Interaction with SUL. If the record corresponding to learner's membership query ϕ does not exist in its inference table, then the mapper randomly selects an access request $\langle u, r \rangle$ from its mapping table P where $(\phi, \langle u, r \rangle) \in P_C$, and forwards $\langle u, r \rangle$ to SUL. Let Φ be the set of relationship patterns between u and r. Then, based on SUL's response, the mapper updates its inference table as:

— If the mapper receives a DENY decision from SUL, then none of the patterns in Φ can be candidate for a rule, and so the mapper updates its inference table $\mathcal{I}$ with correct responses for those patterns. That is, $\forall \phi' \in \Phi$, $\mathcal{I}$ is augmented with $\mathcal{I}_\tau(\phi') = \texttt{DENY}$ and $\mathcal{I}_{\widehat{\phi}} = \mathcal{I}_{\widehat{\phi}} \cup \{\phi'\}$.

— If the mapper receives a PERMIT decision and $\Phi \setminus \{x \in \mathcal{I}_{\widehat{\phi}} \mid \mathcal{I}_\tau(x) = \texttt{DENY}\} = \{\phi\}$ (i.e., all the relationship patterns in Φ except ϕ are certainly denied), then $\mathcal{I}$ is updated as $\mathcal{I}_\tau(\phi)$=PERMIT and $\mathcal{I}_{\widehat{\phi}} = \mathcal{I}_{\widehat{\phi}} \cup \{\phi\}$.
— Otherwise (i.e., PERMIT and the second condition above does not hold), then $\mathcal{I}$ is updated as $\mathcal{I}_\tau(\phi)$=PERMIT. This $\mathcal{I}_\tau(\phi)$ is not necessarily correct since ϕ can be non-permitted *and* there can be other patterns in Φ responsible for PERMIT. Such uncertain membership inferences are resolved through counterexamples from the equivalence oracle (discussed next).

Interaction with the Equivalence Oracle. When the equivalence oracle returns a counterexample ϕ during evaluation of the learner's conjecture, the mapper updates its inference table entries, i.e., $\mathcal{I}_\tau(\phi)$ and $\mathcal{I}_{\widehat{\phi}}$, with correct membership response corresponding to pattern ϕ. Therefore, if the mapper had provided uncertain membership response about ϕ, where $\phi \in \mathcal{I}_\phi$ but $\phi \notin \mathcal{I}_{\widehat{\phi}}$, then the inference table is lazily updated as $\mathcal{I}_\tau(\phi)$ = DENY when a counterexample from the equivalence oracle arises due to the acceptance of a non-authorized pattern ϕ by the learner's hypothesis. Moreover, $\mathcal{I}$ is updated as $\mathcal{I}_{\widehat{\phi}} = \mathcal{I}_{\widehat{\phi}} \cup \{\phi\}$. Henceforth, during our learning algorithm (Algorithm 1), the mapper's membership response for query pattern ϕ are guaranteed to be correct.

7 EQUIVALENCE ORACLE: VALIDATING INFERRED POLICY DFA

The correctness of a hypothesis policy DFA constructed by the learner is assessed by submitting an equivalence query to the equivalence oracle component. The equivalence oracle needs to verify the validity of the learner's conjecture about access controls enforced in SUL. This process determines if the learner's conjecture contains any over-assignments or under-assignments of access permissions, and allows the learner to obtain a better comprehension of the access control policy. Verifying the hypothesis can be performed using a conformance testing strategy. Our equivalence oracle employs information from the mapper's mapping table and inference table. In addition, it queries SUL for access decisions to further validate authorized patterns in the learner's hypothesis. Note that the equivalence oracle does not have direct knowledge about the enforced access control policies since SUL is a black-box component in the context of this work.

In this section, we propose two strategies for implementing the equivalence oracle based on varying degrees of access space coverage. We provide details of our general algorithm for conformance testing in the context of our first strategy, i.e., complete access space coverage. However, we note that the first strategy is not efficient for many real-world applications. Our next strategy, namely, randomized access space coverage, limits the access space exploration. Our experimental evaluation shows the effectiveness of such strategy in practice.

7.1 Complete Access Space Coverage

For a given hypothesis DFA M_H, the equivalence oracle initially checks the validity of M_H against the ground-truth details about relationship pattern authorizations recorded in the mapper's inference table $\mathcal{I}$ (Definition 14). In other words, for any pattern $\phi \in \mathcal{L}(M_H)$ where the mapper has recorded a correct membership response (i.e., $\phi \in \mathcal{I}_{\widehat{\phi}}$), we verify that $\mathcal{I}_\tau(\phi)$ equals PERMIT. Otherwise, ϕ is returned as a counterexample. An analogous evaluation is performed for patterns not accepted by the hypothesis DFA (i.e., $\forall \phi' \notin \mathcal{L}(M_H)$) against the DENY records in $\mathcal{I}$.

To further validate relationship patterns in the inferred DFA, the equivalence oracle inspects the authorization space of SUL by querying the remaining access requests not already covered by the mapper. We employ the mapper's mapping table P (Definition 13) in order to evaluate relationship patterns in hypothesis DFA based on the authorizations of access requests returned by SUL:

— For a denied request $\langle u, r\rangle$: all paths between user u and resource r based on the system graph should be denied by the hypothesis. Formally, $\forall\phi'$ such that $(\phi', \langle u, r\rangle) \in P_C$, it should hold that $\phi' \notin \mathcal{L}(M_H)$. Otherwise, ϕ' is generated as a counterexample.
— For a permitted request $\langle u, r\rangle$: at least one of the paths between u and r should be permitted according to the learner's hypothesis. Formally, $\exists\phi$ such that $(\phi, \langle u, r\rangle) \in P_C$, $\phi \in \mathcal{L}(M_H)$ should be satisfied. Else, the pattern ϕ is generated as a counterexample.

Note that initially the equivalence oracle utilizes denied access requests to identify the non-permitted patterns, and then permitted access requests to identify the authorization rules. This is because, with a denied request $\langle u, r\rangle$, *all* patterns ϕ' between u and r in G, i.e., $(\phi', \langle u, r\rangle) \in P_C$, must be non-permitted. But, when $\langle u, r\rangle$ is permitted, only one pattern needs to be permitted (i.e., be a rule), making it challenging to reason about permitted patterns. Once, we have identified non-permitted patterns, we use that information to eliminate candidates for a permitted pattern.

It should be noted that the equivalence oracle needs to provide only one counterexample (if any exists) to the learner. Therefore, it can use different exploration strategies for finding counterexamples. In our prototype, we prioritize the length and coverage of the explored patterns. Particularly, during counterexample generation, we consider smaller patterns first. Besides, if two patterns have the same length then the one with larger number of access requests is selected.

The time complexity of an equivalence oracle that uses the complete access space strategy is as follows. Enumerating all access requests will need $O(|V|^2)$, given system graph $G(V, E)$. Assume that we already have the pattern-to-access-request association P_C from the mapper's mapping table (Section 6.1.1). Suppose $b = \mathrm{O}(L^N)$ is the upper bound of the number of different patterns that can exist between any two nodes in G, where L is the set of relationship labels and N is the maximum allowable length of a relationship pattern. As N is typically a small number, we can consider b as a constant factor. Since the equivalence oracle checks the hypothesis DFA against every access request, the number of times that membership of patterns needs to be checked in the language of the hypothesis is $O(b|V|^2)$. The accepted patterns can be enumerated all at once by traversing the hypothesis DFA. Such a traversal, using a graph traversal algorithm, takes $\mathrm{O}(|Q|^2)$ where Q is the set of states in the DFA. Hence, in the worst case, the time complexity for a complete conformance test is $\mathrm{O}(|Q|^2 + b|V|^2)$. Practically, $|Q| << |V|$ and thus the complexity becomes $\mathrm{O}(b|V|^2)$.

7.2 Correctness of Equivalence Oracle

In the following, we show that the proposed approach for equivalence oracle is correct, i.e., it precisely captures misjudged relationship patterns in the learner's hypothesis DFA. Our formal analysis relies on the properties of the system graph relative to the ground-truth policy, outlined in Definition 5 (Section 4.1). Based on the concepts of permitted and non-permitted patterns (Section 3.1), we first define the following concepts in relation to the learner's hypothesis DFA:

Definition 15 (False Positive Pattern). A *false positive pattern* is a non-permitted pattern $\phi' \notin \Phi_I$ that is incorrectly identified as a rule by the learner, i.e., $\phi' \in \mathcal{L}(M_H)$.

Definition 16 (False Negative Pattern). A *false negative pattern* is a permitted pattern $\phi \in \Phi_I$ that is *not* identified as a rule by the learner, i.e., $\phi \notin \mathcal{L}(M_H)$.

Theorem 1 (Correctness of Equivalence Oracle). *The working of the equivalence oracle component in the proposed black-box learning architecture is correct, i.e., it correctly responds with a false positive or false negative pattern as a counterexample to an equivalence query.*

Proof. We consider the cases for identifying false positive and false negative patterns separately, and show that the equivalence oracle correctly identifies the counterexample patterns in each case:

Identifying False Positive Pattern $\phi' \in \mathcal{L}(M_H)$. We rely on the property of non-permitted patterns introduced in Definition 5(2). $\phi' \in \mathcal{L}(M_H)$ implies that $O_\tau(\phi') = 1$ in the learner's observation table. Therefore, ϕ' should exist in the system graph G, since by default the access decision associated with all relationship patterns not present in G is DENY. As a result, the first condition of Definition 5(2) does not hold. According to the second condition of the property, there exists some denied access request $\langle u, r\rangle$ in G such that $(\phi', \langle u, r\rangle) \in P_C$. Since $\langle u, r\rangle$ is denied, the equivalence oracle verifies for all ϕ'_i, $(\phi'_i, \langle u, r\rangle) \in P_C$ (including ϕ'), whether $\phi'_i \notin \mathcal{L}(M_H)$. However, it is given that $\phi' \in \mathcal{L}(M_H)$. Hence, ϕ' is returned as a counterexample by the equivalence oracle.

Identifying False Negative Pattern $\phi \notin \mathcal{L}(M_H)$. We employ Definition 5(1) about unique applicability of authorization rules to the system graph for this case. According to this property, for rule $\phi \in \Phi_I$ there exists some permitted $\langle u, r\rangle$ in G such that only the rule ϕ applies to $\langle u, r\rangle$, i.e., $(\phi, \langle u, r\rangle) \in P_C$ *and* $\forall \phi_i \in \Phi_I$ where $\phi_i \neq \phi$ it holds that $(\phi_i, \langle u, r\rangle) \notin P_C$. For the permitted request $\langle u, r\rangle$, the equivalence oracle determines a ϕ_i that satisfies $(\phi_i, \langle u, r\rangle) \in P_C$ *and* $\phi_i \in \mathcal{L}(M_H)$. Since, we first identify non-permitted patterns and then permitted patterns (Section 7.1), $\forall \phi'_i$ where $(\phi'_i, \langle u, r\rangle) \in P_C$ and $\phi'_i \notin \Phi_I$ (i.e., all non-permitted patterns between $\langle u, r\rangle$), it must be true that $\phi'_i \notin \mathcal{L}(M_H)$; otherwise they would already have been identified as false positive patterns as explained above. Therefore, the equivalence oracle specifically checks whether pattern $\phi \in \mathcal{L}(M_H)$. However, it is given that $\phi \notin \mathcal{L}(M_H)$, and hence the equivalence oracle returns ϕ as a counterexample. □

7.3 Randomized Access Space Coverage

The approach discussed in Section 7.1 is not efficient for many real-world applications since access control queries to a system are costly and hence the complete access space of SUL cannot be explored. As a more practical alternative to the complete access space approach, we propose the randomized equivalence oracle, which tests the correctness of the policy hypothesis DFA using a randomized set of access requests. Given the complete space of access requests C, the randomized approach selects a random subset $C_{sub} \subseteq C$ and verifies the learner's conjecture M_H against that.

The randomized strategy for testing the conformity of the learner's conjecture is similar to the complete coverage approach. It follows the same methodology as discussed in Section 7.1 to evaluate relationship patterns in the learner's hypothesis DFA by utilizing permitted and denied access requests. Nevertheless, instead of exploring the complete access space, the number of test cases is reduced by a factor of $|C| \,/\, |C_{sub}|$. However, compared to the complete coverage approach, testing the learner's conjecture based on a random subset of access space can induce errors during learning. This is because the equivalence oracle no longer has knowledge of the complete policy enforced in SUL and it is inherent that it may not be able to correctly examine the learner's hypothesis for misjudged authorization rules. Hence, the randomized method is generally more efficient for large applications with huge number of users and resources that are tolerant to small learning inaccuracy. We experimentally investigate the effectiveness of the randomized equivalence oracle in Section 10.5.

8 LEARNING USING AN EXHAUSTIVE MAPPER

In Section 6, we discussed the design and working of our mapper that minimizes the number of access requests submitted to SUL. We refer to the mapper discussed in Section 6 as the *proposed mapper*. In this section, we propose a naive implementation of the mapper (in terms of its strategy for evaluating the learner's membership queries) in our learning architecture, namely, the *exhaustive mapper*. As evident from its name, this kind of mapper exhaustively explores SUL access space to answer membership queries issued by the learner. This simplified mapper lays the foundation for our formal analysis of the proposed learning framework (which will be studied in Section 9).

In particular, we show that the working of our framework using the exhaustive mapper is more similar to the MAT framework (i.e., the Angluin's L* algorithm explained in Section 2.1) than that using the proposed mapper. Therefore, in this section, we formally establish the correctness of our learning framework using the exhaustive mapper by applying the theoretical results from the Angluin's L* algorithm.

8.1 The Exhaustive Mapper

The proposed mapper employs a *single-match* strategy to answer a membership query, i.e., it infers a membership response based on SUL's decision for "only one" access request matching the queried pattern. However, the exhaustive mapper employs an *all-match* strategy, i.e., it evaluates "all" access requests associated with a queried pattern. This strategy ensures that the exhaustive mapper produces consistent and correct membership responses during the entire learning process.

Definition 17 (Exhaustive Mapper). The exhaustive mapper is a special version of the proposed mapper (Section 6) with mapping table $\langle P_\phi, P_C \rangle$ and inference table $\langle \mathcal{I}_\phi, \mathcal{I}_{\widehat{\phi}}, \mathcal{I}_\tau \rangle$, where $I_\phi = I_{\widehat{\phi}}$. We calculate $\mathcal{I}_\tau$ for a pattern ϕ by querying SUL for $\{\langle u, r \rangle\}$ such that $(\phi, \langle u, r \rangle) \in P_C$. The value of $\mathcal{I}_\tau(\phi)$ will be `PERMIT` only if $|\{\langle u, r \rangle\}| > 0$ *and* SUL returns `PERMIT` for all cases; else it will be `DENY`.

Note that according to the above definition, there will be no uncertainty in membership responses.

Claim 1 (Determinism of Exhaustive Mapper Responses). *The value of $\mathcal{I}_\tau(\phi)$ for a given membership query ϕ does not change throughout the learning process.*

The different strategies used in the exhaustive mapper and the proposed mapper impact the cost of exploring SUL access space and learning iterations. The exhaustive mapper needs to evaluate all access requests associated with a membership query pattern. Thus, in total it submits $\Theta(V^2)$ access requests to SUL, where V is the set of entities in the given system graph. Such an exhaustive exploration of SUL is very costly in practice. On the other hand, the proposed mapper submits only one access request per membership query. Due to the all-match strategy employed by the exhaustive mapper, the membership query responses are always correct and no uncertainties is introduced in the learning process. Thus, at the time of equivalence test, there will be no `uncertain` counterexamples in this case (only `L*` counterexamples can happen). On the other hand, the single-match strategy followed by the proposed mapper induces additional cost to our learning process due to uncertain membership responses. Specifically, the learner may have to perform more iterations in Algorithm 1 due to backtracking when processing an `uncertain` counterexample (Section 5.5).

As discussed above, we emphasize that the exhaustive mapper is mainly explored deeply to facilitate later formal analysis. Implementing it in practice would not be a very feasible option.

8.2 Correctness of Learning Framework using Exhaustive Mapper

Our framework works *correctly* if the learned policy DFA is an acceptor for the unknown ReBAC policy Φ_I enforced by SUL. The learner reaches such a DFA after rounds of membership and equivalence queries about relationship patterns to a MAT teacher who has knowledge of Φ_I. In this section, we prove that when using an exhaustive mapper, our learner is able to infer the correct and minimal DFA representing policy Φ_I, by applying the correctness results from the L* algorithm.

Consider two learning frameworks, F and F', corresponding to our approach and Angluin's approach [6], respectively. In F, our learner (Algorithm 1) interacts with the teacher involving the exhaustive mapper and the equivalence oracle. In F', the learner uses the L* algorithm and interacts with a teacher that correctly answers membership and equivalence queries. In order to

show that both F and F' behave the same in terms of correctness, we prove that: (i) the learner algorithm in F and F' behaves the same, (ii) the membership query responses in F are correct, and (iii) the equivalence query responses in F are correct. We showed (iii) in Section 7.2 (Theorem 1). In Theorem 2, we show (ii) that the exhaustive mapper's responses to the learner's membership queries are always certainly correct according to the SUL's enforced policy. Then, in Theorem 3, we show (i) and prove the correctness of F by employing the results about F'.

Theorem 2 (Correctness of Exhaustive Mapper Responses). *During a complete course of our learning algorithm, for any ϕ, $\mathcal{I}_\tau(\phi) = \texttt{PERMIT}$ if and only if $\phi \in \Phi_I$; otherwise, $\mathcal{I}_\tau(\phi) = \texttt{DENY}$.*

Proof. Based on Claim 1, the mapper's response, i.e., $\mathcal{I}_\tau$, is deterministic. Let $\mathcal{I}_\tau(\phi) = \texttt{PERMIT}$. Then, according to Definition 17, ϕ exists in the system graph and all applicable $\langle u, r\rangle$ requests (where $(\phi, \langle u, r\rangle) \in P_C$) are permitted. By contradiction, assume $\phi \notin \Phi_I$. This contradicts with Definition 5(2) since none of the applicable requests $\langle u, r\rangle$ are denied. Therefore, $\mathcal{I}_\tau(\phi) = \texttt{PERMIT} \implies \phi \in \Phi_I$. On the other hand, let $\phi \in \Phi_I$. Therefore, all applicable requests must be permitted. Therefore, it follows Definition 17 that $\phi \in \Phi_I \implies \mathcal{I}_\tau(\phi) = \texttt{PERMIT}$. □

We borrow the following result regarding framework F' from the Angluin's work [6, Theorem 1]:

Lemma 1 (Correctness and Minimality of Hypothesis DFA [6]). *If $\langle O_S, O_E, O_\tau\rangle$ is a closed, consistent observation table, then the corresponding hypothesis DFA M_H is consistent with the finite function O_τ. Any other DFA consistent with O_τ but inequivalent to M_H must have more states.*

Theorem 3 (Correctness and Minimality of Learner using Exhaustive Mapper). *The learner in framework F infers a correct and minimal DFA representation of ground-truth policy Φ_I enforced in SUL by interacting with the teacher involving the exhaustive mapper and the equivalence oracle.*

Proof. In framework F, since there is no uncertainty in the membership query responses, the learner processes only L⋆ counterexamples. Therefore, the learner algorithm in both frameworks F and F' behaves the same. Additionally, we showed the correctness of equivalence and membership query responses in framework F in Theorems 1 and 2, respectively. Thus, our learning framework F behaves like the Angluin's framework F' in terms of correctness. Based on Lemma 1, framework F' is correct and always produces a minimal DFA consistent with the current observations. Hence, once the equivalence oracle confirms the correctness of the policy DFA, the same correctness and minimality results applies to our framework F. □

9 FORMAL ANALYSIS OF PROPOSED BLACK-BOX LEARNING FRAMEWORK

In this section, we build on our analysis established for the exhaustive mapper (Section 8) to present the theoretical analysis of our framework involving the proposed mapper. We first distinguish the major challenges in studying the proposed mapper in accounting for the uncertain membership responses and the errors in the learning caused by that. We then formally demonstrate the termination (Section 9.1), correctness (Section 9.2), and time complexity (Section 9.3) of our learning approach using the proposed mapper.

As discussed in Section 8.2, our framework using the exhaustive mapper behaves similar to the Angluin's L* algorithm [6]. Angluin showed that, as the learner progresses, the number of states increases monotonically up to the size (number of states) of the initially unknown machine, and this concept was employed to prove the termination of L*. However, in case of learning using the proposed mapper, along with increase in the number of states as in the exhaustive mapper, the

number of states in the inferred DFA may also decrease during uncertain counterexample processing. This is because, for the proposed mapper we additionally need to account for uncertain counterexamples due to potential errors in membership queries (Section 5.5), compared to the exhaustive mapper whose membership responses are deterministic and may only deal with L* counterexamples (Section 8.1).

9.1 Termination of Learning Framework using Proposed Mapper

In this section, we show that our learning algorithm involving the proposed mapper terminates in a finite number of iterations. Our algorithm terminates when no counterexamples can be found in the inferred policy DFA. Therefore, we show that the number of L* and uncertain counterexamples are bounded during a complete learning process. We initially prove the following lemma that relates the two types of counterexamples, L* and uncertain, that can happen in our framework:

Lemma 2 (Non-Repeating Nature of uncertain Counterexamples). *When a counterexample $\widehat{\phi}$ of type either* uncertain *or* L* *is received from the equivalence oracle, then $\widehat{\phi}$ will* not *be generated as an* uncertain *counterexample in the subsequent iterations of our learning algorithm.*

Proof. Based on the counterexample $\widehat{\phi}$ received from the equivalence oracle, the mapper updates its inference table $\mathcal{I}$ with correct responses about pattern $\widehat{\phi}$, i.e., updates the entry $\mathcal{I}_\tau(\widehat{\phi})$ with correct authorization about $\widehat{\phi}$ and $\mathcal{I}_{\widehat{\phi}} = \mathcal{I}_{\widehat{\phi}} \cup \widehat{\phi}$. Therefore, hereafter, the membership query responses inferred by our mapper corresponding to the relationship pattern $\widehat{\phi}$ are guaranteed to be correct. □

We adopt the following lemma about the working of the L* algorithm from the Angluin's work [6], which will be later used in showing the termination of our architecture using the proposed mapper.

Lemma 3 (Processing L* Counterexample [6]). *Processing an* L* *counterexample increases the number of states in the learner's current conjecture by at least one.*

Theorem 4 (Termination of Learner Using Proposed Mapper). *The learning approach involving the proposed mapper, given in Algorithm 1, successfully terminates.*

Proof. At the end of each iteration of the main loop in Algorithm 1 the learner submits an equivalence query for hypothesis DFA M_H. The algorithm terminates when the response to the equivalence query is positive. Such a response implies that *no* L* *or* uncertain counterexamples can be identified for M_H. Therefore, in the following, we show that the number of L* and uncertain counterexamples are bounded during the course of our learning algorithm.

Processing L* counterexamples increases the number of states in the learner's inferred DFA M_H (Lemma 3). The number of states in the inferred policy DFA is bounded by $O(l \times r)$, where r is the number of rules in the policy Φ_I enforced by SUL and l is the length of the longest rule pattern in Φ_I. As a result, the number of L* counterexamples is bounded during the course of the algorithm.

The number of uncertain counterexamples depends on the number of errors due to uncertain membership responses during the course of the learning algorithm. Let η be the set of non-permitted patterns that apply to at least one permitted access request, i.e., for every $\phi' \in \eta$ there exists some permitted $\langle u, r\rangle$ in system graph G such that $(\phi', \langle u, r\rangle) \in P_C$. Recall that for any uncertain counterexample $\widehat{\phi}, O_\tau(\widehat{\phi})$ is "1" instead of "0". This means that while $\widehat{\phi}$ is non-permitted, it has been erroneously observed as permitted based on an applicable request. Thus, $\widehat{\phi} \in \eta$. Therefore, the number of uncertain counterexamples is bounded by the size of η, which is itself bounded

by the size of $\mathcal{R}$ (patterns over L of maximum allowable length). Moreover, based on Lemma 2, an uncertain counterexample can occur at most once during an entire learning process. □

9.2 Correctness and Minimality of Learning Framework using Proposed Mapper

In order to show the correctness of our learning framework using the proposed mapper, we utilize our theoretical results regarding the correctness and minimality using the exhaustive mapper, discussed in Section 8.2. The major difference when using the proposed mapper instead of exhaustive mapper is that we need to account for uncertain membership responses and the errors caused by those during the learning process. In our learning framework, the membership errors are resolved through processing the counterexamples received from the equivalence oracle. During counterexample processing, the learner updates its observation table corresponding to the erroneous entry and the entries affected by the membership of the counterexample pattern (i.e., removing relevant rows and columns in Algorithm 2, Lines 3–12). Fixing the erroneous entries in the observation table is relatively straightforward. However, it is not trivial to show that our uncertain counterexample processing fixes the additional errors created during the learning process (e.g., extra rows/columns), and does not affect the expected working of the L* algorithm.

Theorem 5 (Correctness and Minimality of Learner using Proposed Mapper). *Our learning framework involving the proposed mapper produces a correct, minimal policy DFA corresponding to the ground-truth policy Φ_I enforced in SUL.*

Proof. Recall that there can only exist a finite number of uncertain counterexamples (Theorem 4). Thus, during the course of Algorithm 1, we will eventually obtain a hypothesis DFA where all uncertain counterexamples have been resolved. We first show that after processing the final uncertain counterexample, the framework with the proposed mapper behaves the same as with the exhaustive mapper. Suppose the equivalence oracle responds with the final uncertain counterexample $\widehat{\phi}$ during a run of our learning process. According to our uncertain counterexample processing (Algorithm 2), the learner first corrects the observation table entries corresponding to $O_\tau(\widehat{\phi})$ (Line 2). Then, it determines patterns ϕ_{pre} and ϕ_{suf} ($\phi_{pre}.\phi_{suf} = \widehat{\phi}$) such that ϕ_{pre} is the smallest prefix of $\widehat{\phi}$ that exists in O_S and ϕ_{suf} is its associated suffix that exists in O_E (Lines 3–7). If ϕ_{pre} is the empty string σ, then our algorithm reverts back to the initial observation table (Line 9). Subsequently, the learner with the proposed mapper will behave the same as with the exhaustive mapper, since $\widehat{\phi}$ was the last uncertain counterexample for the current learning process and the proposed mapper will not produce uncertain responses anymore.

On the other hand, if the smallest prefix ϕ_{pre} is a non-empty pattern, then we remove all patterns from O_S that contain ϕ_{pre} as their prefix and also remove all columns, except σ, from O_E (Lines 10–12). We show that the rows corresponding to other patterns in O_S that do not contain ϕ_{pre} as their prefix could not have been affected by the membership error in row(ϕ_{pre}), i.e., those patterns were not added to O_S because of the error in row(ϕ_{pre}). During the process of building a closed, consistent observation table, the learner adds a pattern to O_S when the row corresponding to that pattern is distinct from all rows in O_S, i.e., the observation table is not closed (Algorithm 1, Lines 5–8). However, in the presence of a membership error, such a distinction could have been incorrectly observed. Therefore, we show that such an incorrect distinction is *not* possible for a row whose corresponding pattern in (O_S . L) does not contain ϕ_{pre} as its prefix. Let us refer to such patterns as ϕ_{sl}. Without loss of generality, assume that ϕ_{pre} corresponds to the row that makes the observation table not closed. Thus, before adding ϕ_{pre} to O_S, row(ϕ_{sl}) needed to be equal to some row($\phi_s \in O_S$). Therefore, for any ϕ_{sl}, adding ϕ_{pre} to O_S would not have made row(ϕ_{sl}) suddenly distinct. Moving ϕ_{pre} to O_S might also make the observation table not consistent. The

learner would add a column to the observation table in such a situation (Algorithm 1, Lines 9–12). This new column could make row(ϕ_{sl}) and row(ϕ_s) distinct. However, this distinction cannot be attributed to the error in row(ϕ_{pre}); such a distinction is bound to happen regardless, if row(ϕ_{sl}) and row(ϕ_s) are not supposed to refer to the same state in the ground-truth DFA.

Therefore, our counterexample processing only removes rows corresponding to the patterns with ϕ_{pre} as prefix, since other rows could have not been possibly affected by the membership error. Thereafter, the learner with the proposed mapper will behave the same as with the exhaustive mapper as discussed before. Finally, the correctness and minimality of our framework using the proposed mapper follows those of the case using the exhaustive mapper (Theorem 3). □

9.3 Time Complexity Analysis

We borrow the following complexity result of the L* algorithm from the Angluin's work [6, Theorem 6], which we will utilize in the time complexity analysis of our proposed framework.

Lemma 4 (Time Complexity of L* [6]). *The running time of the L* algorithm is bounded by a polynomial in the number of states of the minimum DFA for the initially unknown finite machine and the maximum length of any counterexample provided by the teacher during the running of L*.*

Consider the possible case that during the entire learning process no uncertainty is introduced by the mapper. Let $t_0, t_1, \ldots, t_{n-1}, t_n$ be the sequence of closed, consistent observation tables during the course of the learning algorithm. Note that in that case there will be no `uncertain` counterexamples.

However, unlike the above special case, uncertainty in the membership query responses, and hence `uncertain` counterexamples, can happen in a regular course of the learning process. According to our counterexample processing (Algorithm 2), processing an `uncertain` counterexample may decrease the number of states in the hypothesis DFA. In the worst case, the counterexample processing will take the learner back to the initial DFA with only one state (i.e., the start state) corresponding to the empty string σ. Therefore, in terms of the worst running time, a learning process may follow the above mentioned sequence of observation tables all the way until the last step, where an erroneous membership response is received. In such a learning sequence, observation tables $t'_1, t'_2, \ldots, t'_{n-1}$ are the same as $t_1, t_2, \ldots, t_{n-1}$, respectively. However, $t'_n \neq t_n$ since there is an erroneous membership response in the last step. As mentioned earlier, processing the `uncertain` counterexample that will be generated as the result of submitting hypothesis corresponding to t'_n, will take the learner back to observation table t_0 in the worst case. Then, the learner has to reiterate the sequence to eventually obtain table t'_n, which will be now the same as t_n. Since, the only counterexample in the hypothesis is resolved, the net effect is running the L* algorithm and obtaining the correct final DFA corresponding to the table t_n.

Now, suppose that instead of one `uncertain` counterexample, we encounter multiple `uncertain` counterexamples. Based on the above argument, in the worst case, the L* algorithm will be repeated for all `uncertain` counterexamples. Since, the number of `uncertain` counterexamples is bounded by η (discussed in the proof of Theorem 4), the worst case complexity of our learning algorithm is the complexity of L* (see Lemma 4) multiplied by a factor of O(η).

10 EXPERIMENTAL ANALYSIS OF PROPOSED LEARNING FRAMEWORK

In this section, we discuss the prototype implementation of our learning framework and compare its performance with other learning approaches. We perform two kinds of assessments, one for the learning phase and the other for equivalence oracle. The learning phase focuses on interaction between learner, mapper and SUL, and we perform experiments under different learning setups to evaluate: (1) the learning cost, i.e., the number of queries and access requests produced to infer

Table 2. Learning Cost Comparison between Our Framework Using the Proposed Mapper, Its Exhaustive Alternative, and an Offline Learning Method

App.	$\|U\|$	$\|R\|$	$\|E\|$	$\|L\|$	Offline	Exhaustive Mapper				Proposed Mapper			
					#Acc.Req.	$\|Q\|$	*#Acc.Req.*	*#Membshp.*	*#Equiv.*	$\|Q\|$	*#Acc.Req.*	*#Membshp.*	*#Equiv.*
EHR	500	480	23,800	8	240,000	8	780,576	590,544	6	8	40,862	820,420	10
Elgg	420	560	24,186	4	235,200	5	717,587	456,712	5	5	15,670	652,704	8

access rules (Section 10.2), (2) the performance in accurately learning the SUL policy (Section 10.3), and (3) the scalability of our approach when the application information is dynamic (Section 10.4). During learning phase assessment, the equivalence oracle is implemented based on complete access space coverage (Section 7.1), so that our assessment is independent of the equivalence oracle's performance. Finally, we assess the feasibility and performance of implementing the randomized equivalence oracle (Section 10.5).

10.1 Setup, Applications, and Baselines

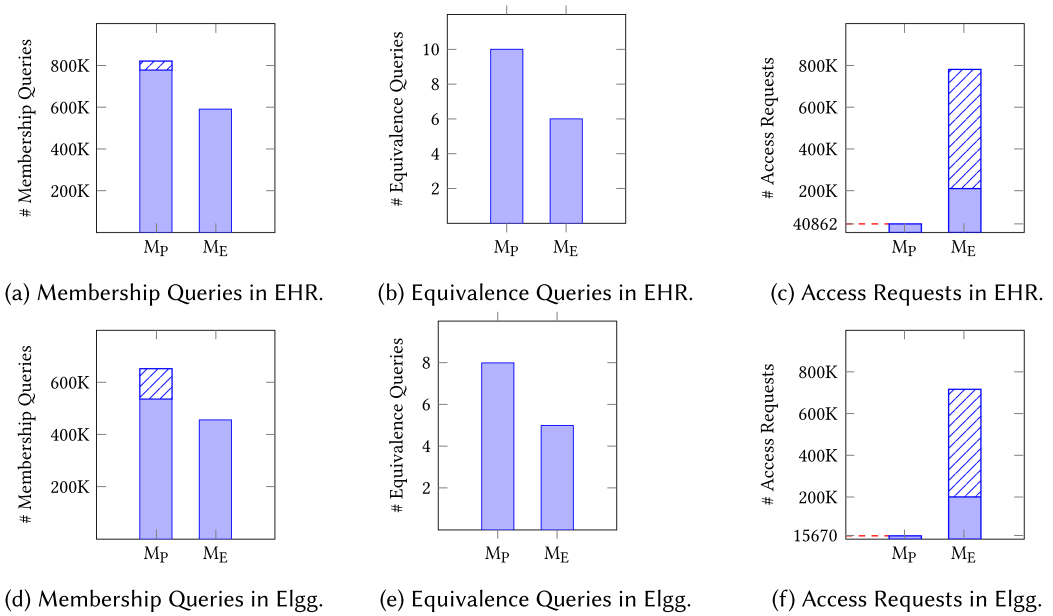
We have implemented a prototype of our learning framework in Java, and performed all our experiments on a 64-bit Windows 10 machine using Intel Core i7-7700 processor and 16 GB of RAM. We execute every experiment 10 times and report the average result over the 10 runs. For evaluating the correctness of the learned authorization rule set, we utilized the access control policy provided by each target system as the ground-truth for comparison. We emphasize that such a ground-truth policy is only used for performance evaluation, and not during the learning process. Also, we constrain the length of the learner queries to 5 based on the maximum length of relationship patterns in the ground-truth policies.

We experimented with two SULs: a simulated SUL in the health-care domain protecting electronic health records, called *EHR*, and an open-source social network application, called *Elgg* [1]. The left part of Table 2 shows the system graph configurations for both applications, where $|U|$, $|R|$, $|E|$, $|L|$ indicate the number of users, resources, edges, and edge labels, respectively. **EHR** regulates accesses by doctors, nurses, and agents to patients' medical records. We adapted the policy specification for EHR from [24]. The system graph was randomly generated by taking into account the domain-specific constraints in place (e.g., the *assists* edge can be between only two users). **Elgg** allows users to add friends, create posts, and comment on their friends' posts. The friend relation in Elgg is directional. To create users and their friend network, we utilized random subset of the relationships data from a social network dataset called soc-Pokec [32]. To generate resources like posts and comments, we simulated random user interactions with the application by employing *UI.Vision RPA* [2]. Our goal is to learn the default visibility policy as applied to users' posts and comments.

We evaluate our proposed learning framework (using the proposed mapper, presented in Section 6) against two baselines. The first baseline is the same learning framework but using the exhaustive mapper (Section 8). As the second baseline we consider an offline learning (i.e., policy mining) approach, in which the learner explores the complete authorization space of SUL to mine the precise policy.

10.2 Learning Cost Assessment

The right part of Table 2 shows the results for the offline learning and our learner using each kind of mapper. In each case, we report number of states in the final policy DFA ($|Q|$), membership queries (*#Membshp.*), equivalence queries (*#Equiv.*), and access requests submitted to SUL (*#Acc.Req.*). Moreover, Figure 5 visually compares the results for the two mappers. The proposed mapper issues more queries than the exhaustive mapper (about 40% extra membership queries and 60% more



(a) Membership Queries in EHR. (b) Equivalence Queries in EHR. (c) Access Requests in EHR.

(d) Membership Queries in Elgg. (e) Equivalence Queries in Elgg. (f) Access Requests in Elgg.

Fig. 5. Visual Learning Cost Comparison, Corresponding to Table 2, When Using Proposed (M_P) vs. Exhaustive (M_E) Mappers for EHR (5(a), 5(b), 5(c)) and Elgg (5(d), 5(e), 5(f)) Applications. Diagonal Fill-Pattern Indicating Duplicates.

equivalence queries). However, our ultimate goal is to reduce the number of access requests submitted to SUL since those are costly to process. The advantage of our framework is demonstrated by the substantial reduction in the number of access requests, i.e., 94% for EHR and 97% for Elgg, in the proposed mapper case compared to that of the exhaustive mapper. Also, the offline learning requires about 83% and 93% more access requests, for EHR and Elgg respectively, than that using the proposed mapper. Note that more access requests are reported for the exhaustive mapper than the offline learning since some access requests may be repeated in the former case (our implementation does not cache the query results); the repeated access requests are shown by diagonal fill-pattern in Figures 5(c) and (f). Furthermore, for the proposed mapper scenario, the number of access requests submitted to SUL is much less than the number of membership queries issued by the learner. This is due to the design of our proposed mapper, which responds to membership queries based on its inferences and interacts with SUL only when necessary. Moreover, note that there are some repeated membership queries (shown by diagonal fill-pattern in Figures 5(a) and (d)), caused due to backtracking during counterexample processing, that do not lead to additional access requests.

10.3 Learning Performance Assessment

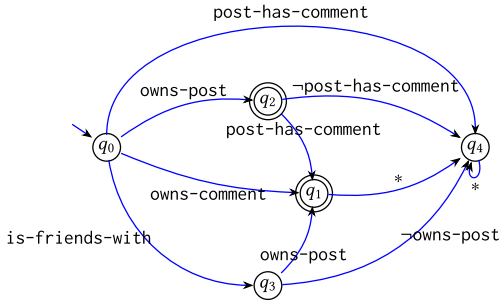
We evaluate the quality of the learned policies against the ground-truth both syntactically (based on policy rules) and semantically (based on permission assignments resulting from policies). Figure 6(a) shows the number of rules in the ground-truth ($|\Phi_I|$) and learned ($|\Phi_H|$) policies as well as our precision and recall in each application scenario. Moreover, we show syntactic similarity of the learned policy and the ground-truth. We calculate syntactic similarity as the average, over rules ϕ in ground-truth, of the syntactic similarity between ϕ and the most similar rule in our learned policy. We measure the syntactic similarity of two rules based on the similarity of their relationship patterns. We were able to learn the accurate policy in the case of EHR. However, the

App.	$\|\Phi_I\|$	$\|\Phi_H\|$	Prec.	Rec.	Syn.Sim.
EHR	8	8	1.0	1.0	1.0
Elgg	5	4	1.0	0.8	0.9

(a) Syntactic Assessment wrt. Policy Rules.

App.	FPR	FNR	Accur.
EHR	0.0	0.0	1.0
Elgg	0.0	0.63	0.87

(b) Semantic Assessment wrt. Permissions.



(c) Policy DFA Inferred from Elgg Application.

Fig. 6. Learning Performance of Our Framework (Using the Proposed Mapper).

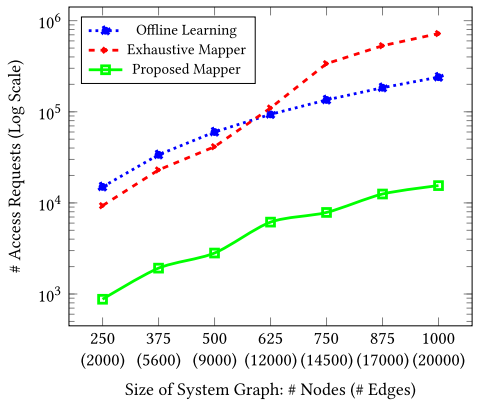
learned policy for Elgg misses one rule (resulting in 0.8 recall). The inferred policy DFA for Elgg is depicted in Figure 6(c). In this case, rules such as *users can access comments on their posts* were learned correctly. But, as expected, our approach was not able to learn a rule in the ground-truth that had a conjunctive condition. This rule states that *users can access comments on others' posts only if they are friends with both owners of the posts and comments*. This is because our current policy model (Section 3.2) and learner algorithm do not support rules that contain conjunctive patterns. Supporting such rules will be in our future work. Figure 6(b) shows the quality of our learned policies measured by permission over-assignment (false positive rate or FPR), permission under-assignment (false negative rate or FNR), and accuracy. Our results show no permission over-assignment in either application. The relatively high false negative rate in the case of Elgg is due to the missing rule in the learned policy as discussed earlier, which results in under-assignments and a reduced overall accuracy.

10.4 Scalability Assessment

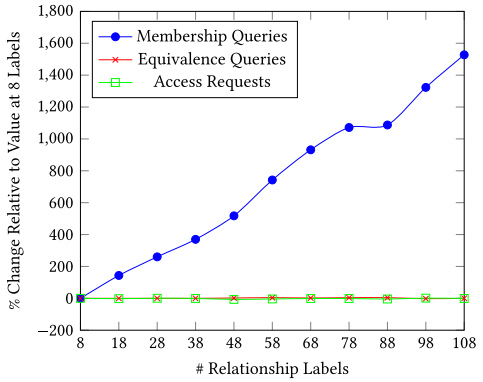
To investigate the suitability of our proposed framework for real-world applications, we examine its scalability and how its cost contrasts against baselines as the target SUL grows. In particular, we consider the growth in terms of the size of the system graph and the set of relationship labels.

System Graph Scalability in Elgg. For examining the performance of our algorithm with respect to different sizes of the system graph, we obtain random samples of various sizes from the soc-Pokec relationships dataset, and simulate them on the Elgg application. Then, with regards to each of the network sizes, we generate a system graph as explained in Section 10.1 and execute our learning approach. Figure 7(a) demonstrates the number of access requests (represented in the log scale) submitted to SUL in Elgg based on different sizes of system graphs. The x-axis depicts the graph growth in terms of number of nodes as well as number of edges (in brackets), while maintaining certain domain integrity constraints. As shown in the figure, the number of access requests for the offline and the exhaustive mapper cases are an order of magnitude larger than that for the proposed mapper. Also, note that the exhaustive becomes more than the offline after a certain point. This is because as the system graph size increases, the number of duplicate access requests generated by the exhaustive mapper also increases leading to larger total number of access requests compared to the offline.

Relationship Labels Scalability in EHR. We study the effect of the size of the relationship labels set (i.e., set of alphabets for policy DFA) on the learning cost of the proposed framework for the EHR application. For each experiment, we add a number of relationship labels to the learner's



(a) Impact of System Graph Size on # Access Requests (Elgg).



(b) Impact of Increase in Size of Relationship Set on #Queries/Requests Using Proposed Mapper (EHR).

Fig. 7. Scalability Evaluation on Learning Cost.

alphabet set L without changing the system graph (so, patterns formed from the added labels do not exist in the system graph) and execute our learning approach. Our observations are shown in Figure 7(b). The x-axis indicates the size of relationship labels set, which is initially "8" since the EHR policy contains 8 labels. The y-axis indicates the percentage change (increase/decrease) in the number of membership queries, equivalence queries, and access requests during a complete run of the learner compared to that of the initial size of "8" labels. According to our results, the number of membership queries increases almost linearly, but the number of equivalence queries and access requests remain almost constant. For the access requests, this is because of the mapper's inference mechanism that returns DENY decision to the learner, if a membership pattern does not exist in the system graph, without consulting SUL. For the equivalence queries, this is because there is no additional uncertainty in mapper's membership responses due to the added relationship labels, and so the extra labels do not cause backtracking during counterexample processing.

10.5 Equivalence Oracle Assessment

We evaluate the performance of our learning framework in presence of the randomized equivalence oracle (discussed in Section 7.3) depending on the amount of available test cases. Test cases are randomly selected access requests of users and resources and associated decisions. Figure 8 demonstrates our observations about the accuracy of the learned policy for different sizes of equivalence oracle test set for EHR. Here, the x-axis represents percentage of user access space considered for equivalence oracle with total access requests submitted to SUL in parentheses; the total also includes access requests due to membership queries. As shown in the figure, with only 40% of access requests provided to equivalence oracle, we achieve an accuracy of about 0.99. Interestingly, with just 1% of access requests, equivalence oracle attains 0.5 accuracy, which manifests the power of our model learning approach for inferring the authorization behavior.

Figure 8 also demonstrates the significant performance of our learning framework compared to that of a state-of-the-art offline algorithm from the literature that mines ReBAC policy using an access log [28]. As the input access log to the offline algorithm, we provided the same access requests and responses that were seen by our framework. Those were seen as part of the membership and equivalence queries in our framework, and their total number is indicated in parentheses on the x-axis of the figure. Our results show that only when more than 80% of the access space is

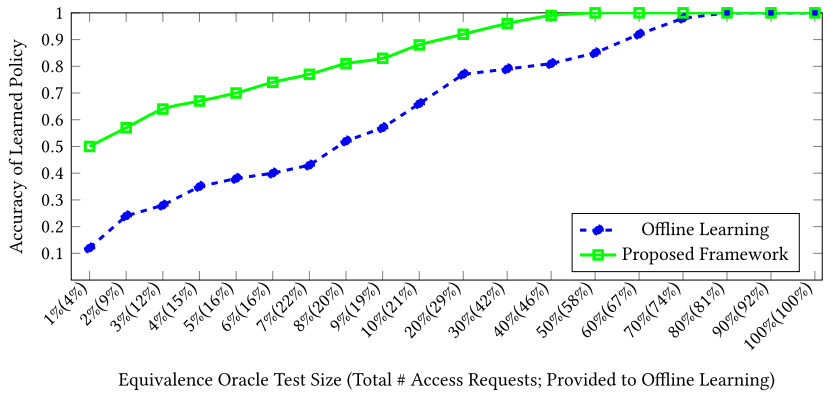


Fig. 8. Learning performance of proposed framework versus an offline learning approach [28] for EHR.

provided, the offline algorithm performs similar to our approach in terms of the accuracy of the mined policy. Moreover, the rate at which the offline algorithm looses accuracy as it is provided with less data is much faster than that of our proposed algorithm. For example, for 20% of access space (involving 10% of equivalence oracle test cases) our proposed approach attains about 0.9 accuracy, while the offline algorithm could reach only around 0.65 accuracy.

11 CONCLUSION

We proposed a first-of-its-kind approach to learn the overall authorization behavior of a system by considering the target system's access control engine as a black box. Particularly, we proposed to learn a DFA model, called *ReBAC policy DFA*, characterizing the ReBAC policy enforced by a system by interacting with its access control engine using a minimal number of access requests. To facilitate such active investigation, we abstracted the large access space of a target system into relationship patterns that are expressed in ReBAC policies, using single-match observations. While this minimalist strategy introduces uncertainty in our learning process, our theoretical analysis proves that the proposed framework learns a correct and minimal DFA model corresponding to the ReBAC policy of a system. Furthermore, our prototype implementation results show that our framework can correctly learn policies as long as the ground-truth policy is expressible in our current ReBAC policy DFA model. Our experiments also demonstrate that our proposed randomized strategy for implementing equivalence oracle is feasible and allows our framework to attain significantly better learning accuracy compared to an offline learning approach that is provided the same data for training. We formally analyzed the cost of our framework through algorithmic complexity of the learner. Our experimental results show the significant cost advantage of our proposed framework compared to two baseline learning setups. In particular, we demonstrate that the proposed framework issues significantly lower number of access control requests to a system, reducing the overhead of learning on the target system in practice. Moreover, we showed that our learning process is scalable across different sizes of users/resources as well as sets of relationship labels in a target system.

Among interesting directions for future work is investigating strategies for more efficient implementations of the equivalence oracle. One such strategy would be to leverage any existing background knowledge about the ground-truth policy to reduce the tested space by the equivalence oracle. We also plan to support modeling and learning more expressive ReBAC policies in our framework that could exist in real-world applications, as evidenced by our experimental results.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and helpful suggestions.

REFERENCES

[1] 2004. Elgg Social Networking Engine. Retrieved August 1, 2021 https://elgg.org/.
[2] 2016. UI.Vision RPA. Retrieved August 1, 2021 https://ui.vision/rpa.
[3] Fides Aarts, Joeri De Ruiter, and Erik Poll. 2013. Formal models of bank cards for free. In *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 461–468.
[4] Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits Vaandrager. 2015. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods in System Design* 46, 1 (2015), 1–41.
[5] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. 2018. A deep learning approach for extracting attributes of ABAC policies. In *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies*. 137–148.
[6] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106.
[7] George Argyros, Ioannis Stais, Suman Jana, Angelos D. Keromytis, and Aggelos Kiayias. 2016. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1690–1701.
[8] George Argyros, Ioannis Stais, Aggelos Kiayias, and Angelos D. Keromytis. 2016. Back in black: Towards formal, black box analysis of sanitizers and filters. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. IEEE, 91–109.
[9] Gunjan Batra, Vijayalakshmi Atluri, Jaideep Vaidya, and Shamik Sural. 2021. Incremental maintenance of ABAC policies. In *Proceedings of the 11th ACM Conference on Data and Application Security and Privacy*. 185–196.
[10] Thang Bui and Scott D. Stoller. 2020. A decision tree learning approach for mining relationship-based access control policies. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 167–178.
[11] Thang Bui and Scott D. Stoller. 2020. Learning attribute-based and relationship-based access control policies with unknown values. In *Proceedings of the International Conference on Information Systems Security*. Springer, 23–44.
[12] Thang Bui, Scott D. Stoller, and Hieu Le. 2019. Efficient and extensible policy mining for relationship-based access control. In *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*. 161–172.
[13] Thang Bui, Scott D. Stoller, and Jiajie Li. 2017. Mining relationship-based access control policies. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. ACM, 239–246.
[14] Thang Bui, Scott D. Stoller, and Jiajie Li. 2018. Mining relationship-based access control policies from incomplete and noisy data. In *Proceedings of the International Symposium on Foundations and Practice of Security*. Springer, 267–284.
[15] Thang Bui, Scott D. Stoller, and Jiajie Li. 2019. Greedy and evolutionary algorithms for mining relationship-based access control policies. *Computers & Security* 80 (2019), 317–333.
[16] Shuvra Chakraborty and Ravi Sandhu. 2021. Formal analysis of rebac policy mining feasibility. In *Proceedings of the 11th ACM Conference on Data and Application Security and Privacy*. 197–207.
[17] Georg Chalupar, Stefan Peherstorfer, Erik Poll, and Joeri De Ruiter. 2014. Automated reverse engineering using lego®. In *Proceedings of the 8th USENIX Workshop on Offensive Technologies (WOOT 14)*.
[18] Alessandro Colantonio, Roberto Di Pietro, and Alberto Ocello. 2008. A Cost-Driven Approach to Role Engineering. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC'08)*. 2129–2136.
[19] Carlos Cotrini, Luca Corinzia, Thilo Weghorn, and David Basin. 2019. The next 700 policy miners: A universal method for building policy miners. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 95–112.
[20] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC rules from sparse logs. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy*. IEEE, 31–46.
[21] Jason Crampton and James Sellwood. 2014. Path conditions and principal matching: A new approach to access control. In *Proceedings of the 19th ACM Symposium on Access Control Models and Technologies*. ACM, 187–198.
[22] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of TLS implementations. In *Proceedings of the USENIX Security 15*. 193–206.
[23] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *Proceedings of the International Conference on Computer Aided Verification*. Springer, 454–471.
[24] Philip W. L. Fong. 2011. Relationship-based access control: Protection model and policy language. In *Proceedings of the 1st ACM Conference on Data and Application Security and Privacy*. ACM, 191–202.
[25] Mayank Gautam, Sadhana Jha, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2017. Poster: Constrained policy mining in attribute based access control. In *Proceedings of the 22nd ACM on Symposium on Access Control Models and Technologies*. ACM, 121–123.

[26] David Harel and Hillel Kugler. 2002. Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science* 13, 01 (2002), 5–51.
[27] Padmavathi Iyer and Amirreza Masoumzadeh. 2018. Mining positive and negative attribute-based access control policy rules. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*. ACM, 161–172.
[28] Padmavathi Iyer and Amirreza Masoumzadeh. 2019. Generalized mining of relationship-based access control policies in evolving systems. In *Proceedings of the 24th ACM on Symposium on Access Control Models and Technologies*. ACM, 135–140.
[29] Padmavathi Iyer and Amirreza Masoumzadeh. 2020. Active learning of relationship-based access control policies. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*. 155–166.
[30] Leila Karimi and James Joshi. 2018. An unsupervised learning based approach for mining attribute based access control policies. In *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 1427–1436.
[31] Ha Thanh Le, Cu Duy Nguyen, Lionel Briand, and Benjamin Hourte. 2015. Automated inference of access control policies for web applications. In *Proceedings of the 20th ACM on Symposium on Access Control Models and Technologies*. ACM, 27–37.
[32] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Retrieved from http://snap.stanford.edu/data.
[33] H. Lu, J. Vaidya, and V. Atluri. 2008. Optimal Boolean Matrix Decomposition: Application to Role Engineering. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. 297–306.
[34] Tiziana Margaria, Oliver Niese, Harald Raffelt, and Bernhard Steffen. 2004. Efficient test-based model generation for legacy reactive systems. In *Proceedings of the 9th IEEE International High-Level Design Validation and Test Workshop*. IEEE, 95–100.
[35] Amirreza Masoumzadeh. 2015. Inferring unknown privacy control policies in a social networking system. In *Proceedings of the 14th ACM Workshop on Privacy in the Electronic Society*. ACM, 21–25.
[36] Eric Medvet, Alberto Bartoli, Barbara Carminati, and Elena Ferrari. 2015. Evolutionary inference of attribute-based access control policies. In *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 351–365.
[37] Barsha Mitra, Shamik Sural, Jaideep Vaidya, and Vijayalakshmi Atluri. 2016. A survey of role mining. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–37.
[38] Ian Molloy, Ninghui Li, Yuan Alan Qi, Jorge Lobo, and Luke Dickens. 2010. Mining roles with noisy data. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*. ACM, 45–54.
[39] Masoud Narouei, Hamed Khanpour, and Hassan Takabi. 2017. Identification of access control policy sentences from natural language policy documents. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 82–100.
[40] Syed Zain R. Rizvi, Philip W. L. Fong, Jason Crampton, and James Sellwood. 2015. Relationship-based access control for an open-source medical records system. In *Proceedings of the 20th ACM on Symposium on Access Control Models and Technologies*. ACM, 113–124.
[41] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of legacy software using model learning and equivalence checking: An industrial experience report. In *Proceedings of the International Conference on Integrated Formal Methods*. Springer, 311–325.
[42] Annie W. Sokol. 2010. A Report on the Privilege (Access) Management Workshop. NIST Interagency/Internal Report (NISTIR).
[43] Frits Vaandrager. 2017. Model learning. *Communications of the ACM* 60, 2 (Jan. 2017), 86–95.
[44] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. 2010. The role mining problem: A formal perspective. *ACM Transactions on Information and System Security (TISSEC)* 13, 3 (2010), 1–31.
[45] Jaideep Vaidya, Vijayalakshmi Atluri, Qi Guo, and Haibing Lu. 2009. Edge-RMP: Minimizing administrative assignments for role-based access control. *Journal of Computer Security* 17, 2 (2009), 211–235.
[46] Jon Whittle and Johann Schumann. 2000. Generating statechart designs from scenarios. In *Proceedings of the 22nd International Conference on Software Engineering*. 314–323.
[47] Zhongyuan Xu and Scott D. Stoller. 2014. Mining attribute-based access control policies. *IEEE Transactions on Dependable and Secure Computing* 12, 5 (2014), 533–545.
[48] Zhongyuan Xu and Scott D. Stoller. 2014. Mining attribute-based access control policies from logs. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 276–291.

Received August 2021; revised January 2022; accepted February 2022