Synthesizing and Analyzing Attribute-Based Access Control Model Generated from Natural Language Policy Statements

Mahmoud Abdelgawad
m.abdelgawad@colostate.edu
Department of Computer Science,
Colorado State University
Fort Collins, Colorado, USA

Indrakshi Ray
indrakshi.ray@colostate.edu
Department of Computer Science,
Colorado State University
Fort Collins, Colorado, USA

Saja Alqurashi
saja.alqurashi@colostate.edu
Department of Computer Science,
Colorado State University
Fort Collins, Colorado, USA

Videep Venkatesha
videep@rams.colostate.edu
Department of Computer Science,
Colorado State University
Fort Collins, Colorado, USA

Hosein Shirazi
hshirazi@sdsu.edu
Management Information Systems,
San Diego State University
San Diego, California, USA

ABSTRACT

Access control policies (ACPs) are natural language statements that describe criteria under which users can access resources. We focus on constructing NIST Next Generation Access Control (NGAC) ABAC model from ACP statements. NGAC is more complex than RBAC or XACML ABAC as it supports dynamic, event-based policies, as well as prohibitions. We provide algorithms that use *spaCy*, a NLP library, to extract entities and relations from ACP sentences and convert them into the NGAC model. We then convert this NGAC model into *Neo4j* representation for the purpose of analysis. We apply the approach to various real-world ACP datasets to demonstrate the feasibility and assess scalability. We demonstrate that the approach is scalable and effectively extracts the NGAC ABAC model from large ACP datasets. We also show that redundancies and inconsistencies of ACP sentences are often found in unclean datasets.

CCS CONCEPTS

• **Security and privacy** → **Access control**.

KEYWORDS

Cybersecurity, Attribute-Based Access Control (ABAC), Next Generation Access Control (NGAC), Natural Language Processing (NLP)

ACM Reference Format:
Mahmoud Abdelgawad, Indrakshi Ray, Saja Alqurashi, Videep Venkatesha, and Hosein Shirazi. 2023. Synthesizing and Analyzing Attribute-Based Access Control Model Generated from Natural Language Policy Statements. In *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT '23), June 7–9, 2023, Trento, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3589608.3593844

SACMAT '23, June 7–9, 2023, Trento, Italy

ACM ISBN 979-8-4007-0173-3/23/06.
https://doi.org/10.1145/3589608.3593844

1 INTRODUCTION

Access control policies (ACPs) are vital to protect resources against unauthorized access while at the same time guaranteeing the resources' accessibility to legitimate users. ACPs are typically part of security requirements, often written in natural language. ACPs articulate strict rules describing how access is managed, who may access resources, and under what conditions. For instance, a doctor can change a patient's record in the healthcare system, but the nurse can only view the patient's record. ACP statements are manually translated into a formal representation (i.e., security model) [4] following a standard framework such as Discretionary Access Control (DAC), Mandatory Access Control (MAC), Role Based Access Control (RBAC), and Attribute-Based Access Control (ABAC). A security model is executed by a security mechanism (software and hardware) to implement the controls (decisions to accept or deny an access request) imposed by the ACP sentences.

Researchers have worked on automated extraction of ACP sentences from security requirements and identifying access control elements corresponding to RBAC and XACML ABAC using machine learning and Natural Language Processing (NLP) techniques [1, 5, 7, 8]. Our work focuses on constructing a NIST Next Generation Access Control (NGAC) model from a set of ACP sentences and formally assessing the generated model's quality. The quality of a model is characterized by a set of correctness properties (namely, completeness, consistency, and minimality).

NGAC is expressed using entities and relations, where the entities correspond to users, user attributes, resources, resource attributes, and policy classes. It supports four relation types: obligations, prohibitions, associations, and assignments. Obligations are event-based rules that specify what must be done post-access. Prohibitions allow the support for negative policies. Association specifies which user attributes allow one to access resources, where the resources that the user can access are described by the resource attributes. Assignment allows it to support attribute hierarchies, user (resource) to user (resource) attribute assignment, and user (resource) attribute to the policy class mapping.

We provide algorithms for extracting the various relations and entities in NGAC. Extracting a model from ACP sentences has its own challenges. For instance, the NGAC model generated may not have all the information. ACPs may have inconsistencies – the access control decision may be both deny and allow for some specific cases. ACPs may have redundancies. We propose a set of correctness properties that the generated model must satisfy. Completeness ensures that all statements are covered in the NGAC model. Consistency ensures that the policy evaluation process produces either allow or deny a decision but not both. Minimality ensures the absence of redundancies in policies. We demonstrate how to check for the satisfaction of these properties.

We leverage NLP techniques to address the problem of extracting policy elements from ACP sentences. The NLP models, such as *spaCy* [2], have advanced significantly using deep learning techniques. The *spaCy* model provides various linguistic features, including Part-of-Speech (POS) tagging, tokenization, lemmatization, syntactic dependency parsing, named entity recognition, and sentence segmentation. These linguistic features help in processing ACP sentences. We use the *spaCy* model to extract entities in the form of (*subject, predicate, object*) from ACP sentences and identify the meaning (the relation between entities) based on the syntactic dependency.

The NGAC model so-generated is then formalized using *Neo4j* [6], a scalable graph database that provides high-performance. *Neo4j* provides many features, including Labelled Property Graph (LPG), Cypher (a SQL-like query language), and Graph Processing Engine (GPE), which consists of a set of graph traversal algorithms. We use *Neo4j* to represent the extracted NGAC elements and relations as a graph with nodes and edges. We demonstrate how to formulate the queries to check for the satisfaction of completeness, consistency, and minimality properties.

2 PRELIMINARY

2.1 Next Generation Access Control (NGAC)

The NGAC security model [3] consists of basic elements and relations. The basic elements include users, processes, resources, operations, access rights, user attributes, resource attributes, and policy classes. For simplicity, this paper ignores processes. The NGAC relation types are assignment, association, prohibition, and obligation. The assignment relation is a directional link that connects policy elements, users to user attributes, user attributes to user attributes, resources to resource attributes, resource attributes to resource attributes, user attributes to policy class, and resource attributes to policy class. The association relation defines the access rights between the user attributes and the resource attributes. The prohibition relations negate access rights between user attributes and resource attributes. The obligation is an event-pattern relation that triggers the execution of the pattern when the event occurs. The obligation relation is essential for constructing a dynamic NGAC security model. The NGAC connects the basic elements by the assignment relation in a hierarchical construction. For instance, consider the NGAC security model shown in Figure 1, which consists of two users, $U = \{Alice, Bob\}$, two resources $R = \{Resume, Contract\}$, three user attributes $UA = \{Staff, HR, Company\}$, two resource attributes $RA = \{HR\text{-}Docs, Files\}$, and one policy class $PC = \{File\ System\}$. The access rights set has read and write $AR = \{read, write\}$. The NGAC security model comprises 9 assignment relations illustrated as solid arrows and two association relations shown as dotted-lines.

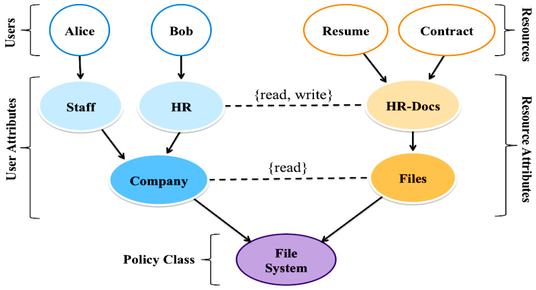

Figure 1: NGAC Security Model Example

Figure 1 demonstrates all types of assignment relations; user to user attribute ($Alice \rightarrow Staff$), user attribute to user attribute ($Staff \rightarrow Company$), resource to resource attribute ($Resume \rightarrow HR\text{-}Docs$), resource attribute to resource attribute ($HR\text{-}Docs \rightarrow Files$), user attribute to policy class ($Company \rightarrow File\ System$), and resource attribute to policy class ($Files \rightarrow File\ System$). It also shows association relations with various access rights. For instance, ($HR - \{read, write\} - HR\text{-}Docs$) shows read and write association relation whereas ($Company - \{read\} - Files$) presents only read association relation. This NGAC security model example does not demonstrate prohibition or obligation relation. Our approach extracts entities from ACP sentences to fill the NGAC element sets, along with extracting the relations to construct the NGAC security model in the exact graph representation as Figure 1.

2.2 *SpaCy* Model

The *spaCy* model provides various linguistic features, including tokenization, lemmatization, Part-of-Speech (POS) tagging, syntactic dependency parsing, and Named Entity Recognition (NER). Briefly, the tokenizer splits a text into meaningful segments known as tokens. The POS labels each token as a noun, verb, adjective, or adverb. The parser builds a syntactic dependency for each sentence as a navigable tree. The lemmatizer converts a word to the base or dictionary form (i.e., lemma); for example, the base form of the word "prohibiting" is "prohibit". The NER identifies various named and numeric entities, such as companies, locations, organizations, and products. We use these linguistic features to extract entities in the form of (*subject, predicate, object*) and identify the meaning (the relation between these entities based on the syntactic dependency). For instance, consider the ACP sentence "An HCP creates patients." tokenized by *spaCy* model as shown in Figure 2. This tokenization shows, in the bottom, the POS as the determiner "An", the proper noun "HCP", the verb "creates", and the noun "patients". It also shows the syntactic dependency tree that explains the relationship between the root verb "creates", the subject "HCP", and the object "patients". This tokenized ACP sentence is used to generate the form (*"HCP", "create", "patient"*) which is required to construct NGAC security model.

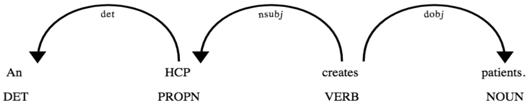

Figure 2: Tokenized ACP Sentence

2.3 Graph Database (Neo4j)

The *Neo4j* [6] is a high-performance graph store and it is schema-free. It organizes the data as nodes, relationships, and properties. A *node* is an unique entity with a set of properties. A *relationship* is a directed edge between two nodes and has a set of properties. A *property* is a key-value pair that characterizes a node or an edge. Cypher is a declarative query language similar to SQL but designed for *Neo4j*. It has keywords that accomplish data functions such as *MATCH*, *CREATE*, *DELETE*, and *RETURN*. For instance, creating an assignment relation to assigning the user *Alice* to the user attribute *HCP* requires three Cyphers. Two Cyphers create nodes u and ua as follows: (i) $CREATE\ (u : U\{name : \text{“}Alice\text{”}\})$ and (ii) $CREATE\ (ua : UA\{name : \text{“}HCP\text{”}\})$ that create two nodes, u type U and labeled with “Alice” and ua type UA and labeled with “HCP”, respectively. The third Cypher retrieves these two nodes and then creates an edge e of type *Assignment* and labels it with *assign_to* shown below.

$$MATCH\ (u : U),\ (ua : UA)$$

$$WHERE\ u.name = \text{“}Alice\text{”}\ AND\ ua.name = \text{“}HCP\text{”}$$

$$CREATE\ (u) - [e : Assignment\{assign : \text{“}assigned\_to\text{”}\}] - > (ua);$$

The *Neo4j* automatically builds a graph representation for this assignment relation as shown in Figure 3. We utilize *Neo4j* to represent the extracted NGAC elements and relations as a graph with nodes and edges, which help us to verify completeness, consistency, and minimality.

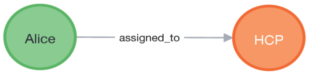

Figure 3: Assignment Relation on *Neo4j*

3 NGAC EXTRACTION APPROACH

3.1 Approach Description

We leverage the existing NLP techniques to extract entities (subject, predicate, object) from ACP sentences and identify the meaning (the relation between entities) based on the syntactic dependency to generate the NGAC security model. The NGAC security model extraction approach consists of four steps. (i) Apply *SpaCy* model to ACP sentences. (ii) Identify the type of relation. (iii) Extract NGAC basic elements and relations. (iv) Test whether the elements and relations are extracted correctly. Figure 4 illustrates the cross-functional flowchart of these steps. The steps are described below.

Apply *SpaCy*: This step starts with loading and splitting the ACP file into ACP sentences. It then applies the *SpaCy* model against each sentence for linguistic features. This process iterates over all ACP sentences until the end of the ACP file. Each ACP sentence linguistically featured (tokenized) is sent to the second process for identifying relation type.

Relation Type Identification: This step examines the ACP sentence's tokens and tags to identify the subject, predicate, and object. It also examines the syntactic dependency of the ACP sentence to identify the relation type as one of the following.

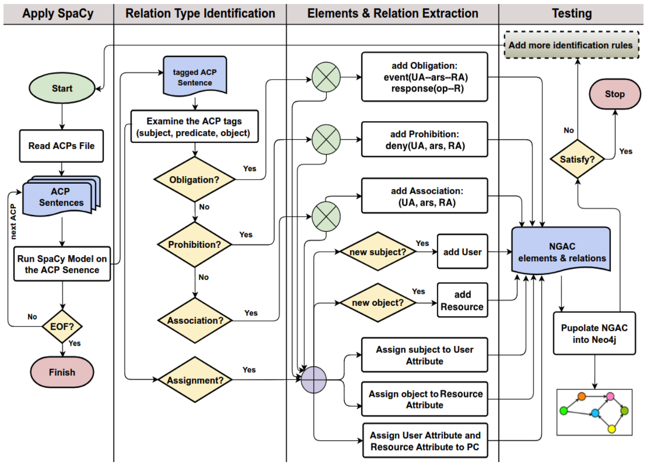
Apply SpaCy
Relation Type Identification
Elements & Relation Extraction
Testing
Add more identification rules
Start
tagged ACP Sentence
Read ACPs File
Examine the ACP tags (subject, predicate, object)
add Obligation: event(UA--ars--RA) response(op--R)
Stop
add Prohibition: deny(UA, ars, RA)
No
Yes
Satisfy?
Obligation?
ACP Sentences
next ACP
No
add Association: (UA, ars, RA)
Yes
Prohibition?
Yes
new subject?
add User
NGAC elements & relations
Run SpaCy Model on the ACP Senence
No
Yes
new object?
add Resource
Yes
Association?
No
EOF?
Yes
Pupolate NGAC into Neo4j
Finish
Yes
Assignment?
Assign subject to User Attribute
Assign object to Resource Attribute
Assign User Attribute and Resource Attribute to PC

Figure 4: Cross-functional Flowchart of the NGAC Extraction

- **Obligation**: If the syntactic dependency includes an adverbial clause, the ACP sentence is marked as an obligation relation with two parts (event and response). The part containing the root verb, the independent clause, represents the response, while the other part, the conditional clause, corresponds to the event.
- **Prohibition**: If the syntactic dependency contains a negation pattern (e.g., a user cannot view, or a user is not allowed to view an object), the ACP sentence is marked as a prohibition relation. If the negation pattern is not found, the root verb is checked against a list of prohibition words (e.g., prohibit, deny, disallow, forbid, prevent, inhibit). The ACP sentence is marked as a prohibition relation if the root verb matches any.
- **Association**: If the ACP sentence is not marked as obligation or prohibition, it is examined as an association relation. Syntactic dependency is used to identify the association relation of the sentence entities in the form (subject, predicate, object), whether the sentence is active or passive.
- **Assignment**: If the syntactic dependency shows that the ACP sentence describes a subject or object (e.g., Tom and Alice are users, or the last name can be public information), it is marked as an assignment relation.

Elements & Relation Extraction: This step forms the relations and entities of the NGAC security model from the relation type identified by the step Relation Type Identification. It then inserts each NGAC relation into the set where it belongs. For instance, the ACP sentence *“a user cannot edit his account number”* is marked as prohibition; thus, it will be formed and stored as *“deny(user–edit–account_number)”*.

This process also creates new elements, namely, subjects and objects, if the entities are new. It then assigns each element to the attribute set in which it belongs: a subject is assigned to user attribute UA sets and an object is assigned to resource attribute RA sets. If an element does not belong to any existing user attribute or resource attribute, a new attribute set is created to contain this element.

Testing: This step converts the NGAC elements and relations to a graph with nodes and edges using the Neo4j graph database. We call it the NGAC graph. The Testing process populates the NGAC sets generated (U, UA, R, RA) as nodes and creates the relationships among these nodes using the NGAC relations generated

($Assignment$, $Association$, $Prohibition$). It then assigns the user attribute set UA and resource attribute set RA to a policy class PC to form a complete NGAC graph. The $Obligation$ relation is the mechanism that makes the NGAC security model dynamic, changing the PC structure at runtime. For instance, it removes or assigns a user from a user attribute, associates user attributes to a resource attribute, and changes a prohibition configuration through administrative obligation triggered by an access event.

Our work considers the initial configuration of the NGAC security model. Thus, the $Obligation$ relations are omitted from the NGAC graph. However, the $Obligation$ relations are identified to extract new entities, such as users and resources, which stay part of the NGAC graph. The Testing process uses the NGAC graph to derive testcases. Each testcase represents a path in the NGAC graph that connects a user node U to a resource node R. The edges that link the user attributes UA and resource attribute RA to policy class PC are ignored because they do not represent relations extracted from ACP sentences.

We use the Cypher Query Language (specifically, MATCH statement) to generate such paths. The Testing process automatically scripts these paths as testcases. The testcases then execute against the ACP sentences to ensure that the ACP sentences are extracted in the NGAC security model correctly. If testcases fail to satisfy the ACP sentences, new identification rules are added, and the entire process starts over.

3.2 Implementation

For simplicity, we selected 8 ACPs from the *iTrust* dataset [8] to demonstrate possible scenarios for constructing the policy class shown in Table 1. Sentences 1, 2 and 4 demonstrate a conflict of ACPs (grant and deny for a doctor to create patients). Sentence 4 also exemplifies a negative ACP that does not use the word *"Not"* but instead uses a prohibition word (prevent). Sentences 5 and 6 demonstrate various privileges assigned to user attributes (doctors and nurses). Sentence 7 shows an obligation relation, while sentences 8 illustrates an assignment relation. To demonstrate users accessing resources, we added 3 sentences, shown in Table 1 sentences 9-11, that assign three users (Bob, Alice, and Jack) as a doctor, nurse, and administrator, respectively. We also added 2 sentences (12 and 13) to demonstrate resources assigned to resource attributes.

iTrust: **Access Control Policies**	
1	An HCP creates patients.
2	Doctor is an HCP.
3	Nurse is an HCP.
4	A doctor is prohibited from creating patients.
5	Doctors can update the patient record.
6	Nurses can only view the patient record.
7	Whenever a HCP changes a patient record, an email must be sent to the administrator.
8	The administrator can assign a patient.
Users	
9	Bob is a doctor.
10	Alice is a nurse.
11	Jack is an administrator.
Resources	
12	JohnSmithRecord is a patient record.
13	John is a patient.

Table 1: Access Control Policies and Users

3.2.1 ***NGAC Extraction.***

We use *SpaCy* model (*en_core_web_lg*) for linguistic features, including tokenization, POS tagging, lemmatization, and syntactic dependency. The *en_core_web_lg* model size is large (560 MB); however, it provides high-level accuracy for tokenization, syntactic dependency, and POS. The *SpaCy* model is applied to the ACP sentences shown in Table 1 for tokenization. For each ACP sentence, the *SpaCy* model results in the POS and syntactic dependency tree. The *Relation Type Identification* algorithm, Algorithm 1, uses the POS tags and syntactic dependency tree to determine the relation type - whether assignment, association, prohibition, or obligation. Algorithm 1 first examines whether the ACP tokenized sentence is an obligation relation type (lines 1 to 7). If the syntactic dependency tree (i.e., *SDTree*) contains an adverbial clause tag "advcl", then it means that the ACP sentence has two parts (independent clause and conditional clause). The *SpaCy* model allows us to process each part as a syntactic dependency subtree. The independent clause subtree is processed as a response clause, while the conditional clause subtree is treated as an event clause. Each subtree is passed to the element and relation extraction algorithm (Algorithm 2) to sequence the sentence's elements into the *"subject–predicate–object"* format. The obligation identification process results in NGAC format as follows:

$$Obligation(Event : (UA - \{ars\} - RA) \longrightarrow Response : (op - R))$$

Algorithm 1: Relation Type Identification

```
   Input  : tokenized ACP : POS, SDTree
   Output : Relation Type : {Assign, Assoc, Prohib, Obligat}
   /* Check if the sentence is obligation                      */
1  if SDTree contains "advcl" tag then
2      eventClause ← ConditionClause(ACP)
3      responseClause ← IndependentClause(ACP)
4      event ← RelationExtraction(eventClause)
5      response ← RelationExtraction(responseClause)
6      return Obligat(event, response)
7  end
   /* Check if the sentence is prohibition                     */
8  else if SDTree contains "neg" tag OR
     the root verb ∈ the prohibition word list then
9      denyClause ← RelationExtraction(ACP)
10     return Prohib(denyClause)
11 end
   /* Otherwise the sentence is association                    */
12 else
13     accessRightClause ← RelationExtraction(ACP)
14     return Assoc(accessRightClause)
15 end
```

If the ACP sentence is not an obligation, it is then examined whether it is a prohibition relation (lines 8 to 11). Algorithm 1 checks that if the *SDTree* contains a negation "neg" tag. If the *SDTree* does not include the negation tag, the Algorithm 1 searches for prohibition word over the *SDTree.* We support Algorithm 1 with a list of prohibition words, such as *prohibit, deny, disallow, forbid, restrict, inhibit, prevent, revoke, discard, and decline.* The prohibition word list contains 26 words that are selected carefully from various English language dictionaries. If the ACP sentence is a prohibition, it is passed to the element and relation extraction algorithm (Algorithm 2) to form the sentence's elements as *"subject–predicate–object"*. The prohibition identification process results in this form:

$$Prohibition(deny(UA - \{ars\} - RA))$$

If the ACP sentence is neither obligation nor prohibition, then it is considered as an association relation type that has this format:

$$Association(UA - \{ars\} - RA)$$

Algorithm 2 receives a tokenized ACP that includes POS tags and dependency (i.e., *dep*) tree or sub-tree as input. It then iterates over each token in the ACP to extract the root verb, subjects, and objects. If the token's dependency is *"ROOT"* and the POS tag is *"VERB"*, it means the token points to the root verb of the sentence and is extracted as a *"predicate"*. If the token's dependency is *"subj"* and the POS tag is *"PROPN"* or *"NOUN"*, it means the token points to a subject and is extracted and added to a *"subjects"* list. The *"subjects"* list carries two types of elements: proper noun-subjects, which represent users, and common noun-subjects, which represent user attributes. We assume that an ACP sentence may contain one or many subjects and objects but only one predicate. Thus, we treat them as a unique element. Suppose the token's dependency is *"obj"* and the POS tag is *"PROPN"* or *"NOUN"*; it means that the token points to an object and is extracted and added to a *"objects"* list. After the iteration loop ends - all tokens have been examined - Algorithm 2 returns the results as a tuple *(subjects, predicate, objects)* to be used in the other processes. The assignment relation is processed differently since it occurs whenever a new subject and object are found in an ACP sentence, regardless of whether the sentence is an association, prohibition, or obligation. For each ACP sentence, the assignment extraction algorithm, Algorithm 3, searches for subjects and objects using POS tags as they are *"PROPN"* or *"NOUN"* and connected to the root verb, *"ROOT"*. Suppose the ACP sentence consists of a single subject and object. For this case, Algorithm 3 first extracts the subject and object. If the subject is a proper noun-subject, it creates an assignment relation user to user attribute ($U - UA$) and adds it to the assignment set (*assignments*). If the subject is a common noun-subject, it creates an assignment relation user attribute to user attribute ($UA - UA$) and adds it to the assignment set (*assignments*). Similarly, if the object is a proper noun-object, it creates an assignment relation resource to resource attribute ($R - RA$) and adds it to the assignment set (*assignments*), and if the object is a common noun-object, it creates an assignment relation resource attribute to resource attribute ($RA - RA$) and adds it to the assignment set (*assignments*). Algorithm 3 processes four types of assignment relations, user-to-user attribute, user attribute-to-user attribute, resource-to-resource attribute, and resource attribute-to-resource attribute. Algorithm 3 then creates an assignment relation to connect the proper noun-subject found to its subject attribute set and another assignment relation that connects the resource to its resource attribute set. At the final step, Algorithm 3 assigns every user to user attribute and resource to resource attribute. For instance, the user *Jack* assigned to the user attribute *administrator* and the resource *John* to the resource attribute *patient*.

Algorithm 2: Element & Relation Extraction

```
Input  : $tokenized\ ACP:\ POS,\ dep$
Output: $formation : tuple([subjects], predicate, [objects])$
/* Iterate over each token in the ACP */
foreach $token \in ACP$ do
    /* check the verb's dependency and POS */
    if $dep$ = "ROOT" AND POS = "VERB" then
        $predicate \leftarrow token.text$
    end
    /* check the subject's dependency and POS */
    else if $dep$ = "subj" AND POS = "PROPN" OR "NOUN" then
        $subjects \leftarrow add(token.text)$
    end
    /* check the object's dependency and POS */
    else if $dep$ = "obj" AND POS = "PROPN" OR "NOUN" then
        $subjects \leftarrow add(token.text)$
    end
end
return ($subjects,\ predicate,\ objects$)
```

Algorithm 3: Assignment & Relation Extraction

```
Input  : $tokenized\ ACP:\ POS,\ dep$
Output: $assignments : assign(U \rightarrow UA), assign(R \rightarrow RA),$
        $assign(UA \rightarrow UA'), assign(RA \rightarrow RA')$
/* Iterate over each token in the ACP */
foreach $token \in ACP$ do
    /* add subject to U set */
    if $dep$ = "subj" AND POS = "PROPN" then
        $U \leftarrow add(token.text)$
    end
    /* add subject to UA set */
    if $dep$ = "subj" AND POS = "NOUN" then
        $UA \leftarrow add(token.text)$
    end
    /* add object to R set */
    if $dep$ = "obj" AND POS = "PROPN" then
        $R \leftarrow add(token.text)$
    end
    /* add object to RA set */
    if $dep$ = "obj" AND POS = "NOUN" then
        $RA \leftarrow add(token.text)$
    end
    /* assign user to user attribute */
    $assign(U \rightarrow UA)$
    /* user attribute to user attribute */
    $assign(UA \rightarrow UA')$
    /* assign resource to resource attribute */
    $assign(R \rightarrow RA)$
    /* resource attribute to resource attribute */
    $assign(RA \rightarrow RA')$
end
```

3.2.2 *NGAC Testing.*

We connect the NGAC elements and relations extracted as a graph with nodes and edges. We populate these elements and relations into *Neo4j* graph database and represent them as nodes and edges respectively. The *Neo4j* graph database allows us to generate paths from the NGAC graph. These paths are scripted as testcases which will be executed against the ACP sentences. As shown in Figure 5, the 13 sentences shown in Table 1 are extracted as NGAC security model and populated into the *Neo4j*. We first created a node for each NGAC element- user, user attribute, resource, resource attribute, and policy class. These elements are then connected by creating the extracted NGAC relations. Figure 5 shows 3 users U's (nodes colored with light blue), 4 user attributes UA's (nodes colored with dark blue), 2 resources R's (nodes colored with light orange), 2 resource attributes RA's (nodes colored with dark orange). The UA's nodes contain U's nodes, and the RA's contain R's nodes through assignment relations. These assignment relations are represented as light blue arrows labeled *"assign_to"*. Figure 5 illustrates 7 assignment relations. It also shows 5 association relations that are represented as green arrows labeled with the words *create, update, view, change, and assign*, which correspond to access rights extracted from the root verbs of the first 8 ACP sentences. It also shows 1 prohibition relation that is represented as a brown arrow labeled with the word *deny[create].*

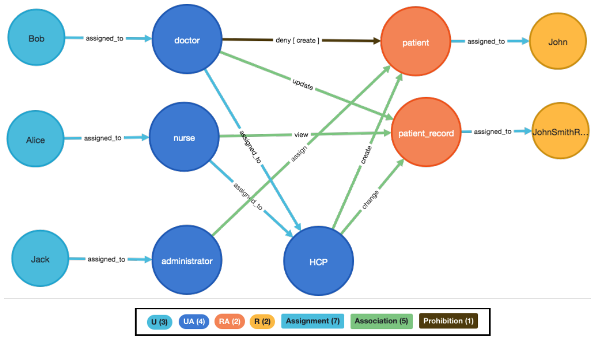

Figure 5: NGAC Represented By Using Neo4j

We use the NGAC graph to generate paths, from U’s nodes to R’s node, that matches the ACP sentences. For instance, the path that starts from the user *Bob*, to the user attribute *doctor*, the resource attribute *patient*, and the resource *John*, should match three sentences (4,9, and 13) from Table 1. This path is presented as follows:

$$(Bob \longrightarrow doctor \xrightarrow{deny[create]} patient \longrightarrow John)$$

We use *Neo4j* Cypher path matching (i.e., *MATCH* statement) to generate all possible paths starting from every U user attribute node to every R resource attribute node. The *MATCH* statement is written as follows:

$$\begin{aligned} MATCH\ paths = (u:U) - [&: Assignment|\ Prohibition| \\ &Association * 1..4] - (r:R) \\ RETURN\ \ distinct(\ paths\ )\ \ &ORDER\ BY\ \ length(paths); \end{aligned}$$

This *MATCH* statement generates 8 paths that cover the graph, Figure 5, from U to R. For instance, the path starts from user “Jack” as an administrator, shown in left-bottom of Figure 5, and assign a patient name “John”, is represented as:

```
[{name:"Jack"},{assign:"assigned_to"},{name:"administrator"},
{name:"administrator"},{ars:"assign"},{name:"patient"},
{name:"patient"},{assign:"assigned_to"},{name:"John"}]
```

These paths are a sequence of nodes and relations representing how a user node U reaches a resource node R through *Assignment* and *Association* relations. We use these paths to script testcases that will be executed against each ACP sentence’s content. For instance, when the testcases are executed against the sentence number 1 (*“An HCP creates patients”*), at least one testcase has to successfully pass. This testcase matches the word *“HCP”* as a user attribute node, *“create”* as an access right node (intermediate node), and the word *“patient”* as a resource attribute node. If all testcases fail to match these three words, this ACP sentence has not been extracted completely. This testing process is executed for each ACP sentence.

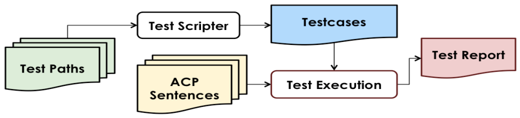

Figure 6: Testing Process

The testing process, presented in Figure 6, starts with loading the paths generated, and then a test scripter converts them into *Pyhton* code testcases. For each path, the test scripter fetches the user from the first node, the access right from an intermediate node, and the object from the last node. It then converts them as *Regular Expression*. For example, consider the following path:

$$[\{“name” : “HCP”\}, \{“ars” : “create”\}, \{“name” : “patients”\}]$$

The scripter will fetch the “HCP”, “create”, and “patient” words and then form a regular expression as:

$$“. * HCP. * create. * patient. * ”$$

The dot refers to any character; the star means zero or many. As a result, this regular expression verifies that from the beginning of the sentence, ignore a series of characters until reaching the “HCP” word, ignore another series of characters until reaching “create", and another series of characters until reaching the word “patients”. Similarly, the scripter converts all paths into regular expressions. Each regular expression is then passed as an argument to a match function (*search(pattern, ACP_Sentence)*) that searches for a match over a string variable that presents an ACP sentence. The case sensitivity of letters is ignored. The result of this function is asserted to be not equal to *“None”*. The match function and the assertion are wrapped up in a testcase defined with a unique name. The testcases generated are then encased into a class (i.e., test-suite).

The test execution first loads the ACP sentences. It then passes each ACP sentence to the test-suite, executes it, and reports the test result into a file. The test result is analyzed based on the following test criterion:

$$“for\ each\ ACP\ sentence,\ at\ least\ one\ testcase\ should\ pass”$$

The 8 paths generated from the graph, Figure 5, were scripted and executed against the first 8 ACP sentences presented in Table 1, which form part of the policy class. The result of this execution is summarized in Table 2, which presents the ACP sentence and the number of testcases that it successfully passed. Table 2 shows that all ACP sentences have at least one testcase that they passed. This indicates that the extraction of NGAC from the ACP sentences presented in Table 1 is complete. However, the generated paths show that the ACP sentences suffer from *inconsistencies*. In other words, the ACP sentences express privilege conflicts. For instance, the third path denies a doctor from creating a patient but the seventh path allows a doctor to create a patient. The exact use case occurs for the nurse user: the fourth path grants a nurse only viewing privilege for a patient record, whereas the tenth path grants her the privilege to change a patient record. This is because doctors and

Access Control Policy Sentence	# of TCs Passed
An HCP creates patients.	1
Doctor is an HCP.	1
Nurse is an HCP.	1
A doctor is prohibited from creating patients.	1
Doctors can update the patient record.	1
Nurses can only view the patient record.	1
Whenever a HCP changes a patient record, an email must be sent to the administrator.	1
The administrator can assign a patient.	1

Table 2: Testing Results

nurses were assigned to the user attribute *HCP*, which has access rights to create a patient and change a patient record.

4 ANALYSIS

We collected policy documents from 7 educational institutes, including the University of Massachusetts Amherst, the University of Arizona, Georgia State University, Colorado State University, New York University, the University of Denver, and Harvard University to analyze correctness and scalability. Table 3 illustrates the datasets' names (abbreviations) and sizes (number of ACP sentences).

Institute	Dataset Name	# of ACPs
University of Massachusetts Amherst	MA	2
University of Arizona	AZ	9
Georgia State University	GA	20
Colorado State University	CSU	62
New York University	NYU	83
University of Denver	DU	105
Harvard University	Harvard	249

Table 3: Datasets Name and Size

4.1 Correctness

We analyzed the approach's correctness in terms of *completeness*, *minimality*, and *consistency*.

The *completeness* demonstrates that *"does the generated NGAC security model cover all ACP sentences?"* Testing the generated NGAC graph against the ACP sentences answers this question. Table 4 illustrates the results of testing each NGAC graph generated against its dataset along with the accuracy; the number of testcases passed over the number of testcases generated from the graph. The accuracy of NGAC security model completeness varies from one dataset to another, with an average of 83.56%, which points to the fact that the NGAC graph does not cover all ACP sentences. We investigated this incompleteness by looking at each ACP sentence that was not covered. We found that these ACP sentences are grammatically correct but either awkward or missing one of the main sentence's elements (subject, predicate, object). Cleaning the ACP sentences noticeably influences the NGAC security model extraction.

Dataset	# of TCs	# of TCs Passed	Accuracy(%)
MA	16	14	87.50
AZ	31	23	74.19
GA	248	234	94.35
CSU	162	146	90.12
NYU	271	215	79.34
DU	440	388	88.18
Harvard	509	359	70.53

Table 4: Testing Datasets Result

The *minimality*, i.e., non-redundancy, refers to the fact that an access right given to a user attribute to access a resource attribute should not exist for two different paths in the NGAC graph. We checked the minimality for each dataset by generating two sets of paths; one set has all paths from user attributes UA to resource attributes RA without distinguishing them, and the other set includes distinguishing paths (using *"DISTINCT"* keyword in the *Neo4j* Cypher to generate paths). These sets' differences result in redundant paths, reflecting ACP sentences' redundancy. The result shows that all NGAC security models are non-redundant except those generated for the GA dataset. Investigating that, we found that this dataset contains two types of policies (education and employment policy), causing some user attributes UA access to the same resource attributes RA. Unclean ACP dataset may also cause a redundancy issue.

The *consistency* of the NGAC security model verifies that, for a particular resource attribute, access rights given to one user attribute UA_1 and prohibited in another user attribute UA_2, and a user or user attribute is assigned for both UA_1 and UA_2, that is, granting and denying a user or a user attribute to access the same resource. In such a case, the model is inconsistent. The inconsistency always occurs between association and prohibition relations, granting a user attribute to access a resource in one place and denying it from accessing the same resource in another place. We conducted consistency verification to detect inconsistency. We first separated the paths into two sets; one set carries paths that are association relations, and the other one consists of paths that are prohibition relations. We then exercised that for every user attribute, matching the access rights of paths with association relations to the denied access rights of paths with prohibition relations to which the same user attribute and resource attribute is assigned. We used *regular expression* for matching values of the access rights and denied access rights. We did not find any inconsistency in the NGAC security models.

4.2 Scalability

We analyzed the approach's scalability regarding the *space complexity* and *time complexity*.

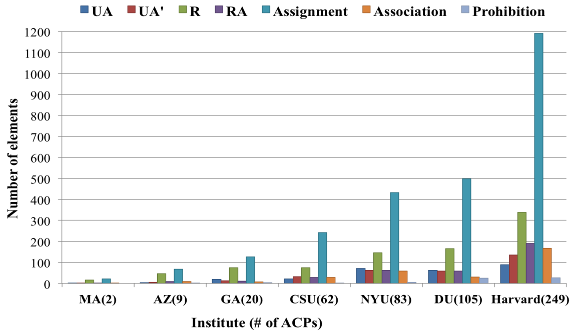

Figure 7: Space Complexity of NGAC Security Model

Figure 7 shows the number of NGAC basic elements and relations extracted from each dataset. Note that the number of assignment relations increases when the number of ACP sentences increases. The MA dataset contains two ACP sentences, and the number of assignment relations extracted is 22, whereas the Harvard dataset consists of 249 ACP sentences and generates 1191 assignment relations. The *space complexity* for the extraction process depends on the syntactic dependency tree generated by the *spaCy* model for an ACP sentence. It is $O(nm)$, where n is the syntactic dependency tree's depth, and m is the number of calls of the extraction function at each level. The *space complexity* for populating the NGAC elements and relations into the *Neo4j* is $O(V + E)$, where V is the number of vertices representing the NGAC basic elements, and E is the number of edges representing the NGAC relations.

The *time complexity* for the extraction process also depends on the syntactic dependency tree generated by the *spaCy* model for each ACP sentence. As shown in Figure 8, the blue line, it took less than 5 seconds to extract elements from 2 ACP sentences of the MA dataset, whereas it took 35 seconds when extracting elements from the 249 ACP sentences of the Harvard dataset. The extraction process is specifically tree search; thus, the *time complexity* is linear-logarithmic time $O(n\ log\ n)$, where n is the syntactic dependency tree's depth of an ACP sentence. We noticed that extracting elements from compound ACP sentences take longer than simple sentences. This is because the syntactic dependency tree size is larger for the compound sentence compared to that of the simple sentence. Figure 8, the red line, represents the *time complexity* for the testing process. The testing process took 3 seconds when the testcases were executed against the MA dataset and it took 58 seconds when the testcases were executed against the Harvard dataset. The *time complexity* for the testing process is quadratic $O(n^2)$.

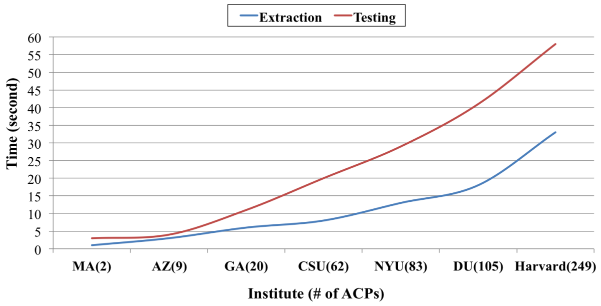

Figure 8: Time Complexity of NGAC Security Model

5 RELATED WORK

Earlier works [1, 5, 7, 8] focus mostly on the automatic identification of ACP sentences from natural language documents using NLP and extracting security elements corresponding to RBAC and XACML ABAC.

X. Xiao *et al.* [8] introduced a novel approach to extract ACP sentences from natural language documents which they refer to as Text2Policy. The Text2Policy uses linguistic analysis (a shallow parsing) to parse and annotate words and phrases in the sentences and then use this annotation to extract the subject, action, and object from each sentence. The authors use the extracted elements to generate eXtensible Access Control Markup Language (XACML), a machine-enforceable language designed to implement ABAC.

J. Slankas *et al.* [7] conducted various works on identifying ACP sentences and extracting ACP elements. They use inductive reasoning to find and extract access control rules to construct the RBAC security model. They also use semantic role labeling (SRL), using supervised machine learning that detects semantic arguments associated with a verb (i.e., a predicate). The approach was evaluated with various documents, conference management, education, and healthcare. The result shows the approach identified ACP sentences that exist in 47% of the sentences with a precision of 81% and recall of 65%. It also shows that the approach extracted access control roles from those identified ACP sentences with an average precision of 76% and an average recall of 49%.

M. Narouei *et al.* [5] also conducted various works using an SRL semi-supervised machine learning models to identify role in ACP sentences to construct RBAC model and to identify ACP and extract elements to construct ABAC model. The authors show that the ACP identification was achieved with an average recall and precision of 90%, whereas their previous work of ACP identification was achieved with an average F1 score of 75 percent, which improved by 15 percent.

M. Alohaly *et al.* [1] presented a deep learning approach, a Convolutional Neural Network (CNN), to extract attributes of subjects and objects to construct the hierarchical structure of an ABAC security model. For evaluation, the approach was applied to various ACP datasets, iTrust, Collected ACP, IBM App, and CyberChair. The results show that approach achieved an average - F1-score of 0.96 when extracting the attributes of subjects and 0.91 when extracting the objects' attributes. The approach was limited to synthetic data the authors collected and injected with policy element attributes gathered from the Web content.

6 CONCLUSION

We proposed an approach to extract the NGAC basic elements and relations from ACP sentences and verify the resulting NGAC model. We used *spaCy* for the extraction, and *Neo4j* to construct the NGAC security model which is analyzed for correctness properties. The approach was applied to various real-world ACP datasets demonstrating feasibility and scalability. Future work includes using various NLP transformer models, such as, BERT and GPT-3, for extraction and analysis.

ACKNOWLEDGMENTS

This work was supported in part by funding from NSF under Award Numbers ATD 2123761, CNS 1822118, NIST, ARL, Statnett, AMI, NewPush, and Cyber Risk Research.

REFERENCES

[1] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. 2019. Automated extraction of attributes from natural language attribute-based access control (ABAC) policies. *Cybersecurity* 2, 1 (2019), 1–25.

[2] Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. https://spacy.io/ Accessed: 2023.

[3] American National Standards Institute. 2020. *Information Technology - Next Generation Access Control (NGAC).* Information Technology. ANSI, New York, NY. Accessed: 2023.

[4] Xin Jin, Ram Krishnan, and Ravi Sandhu. 2012. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI.* Springer, Berlin, Heidelberg, 41–55.

[5] Masoud Narouei, Hassan Takabi, and Rodney Nielsen. 2018. Automatic extraction of access control policies from natural language documents. *IEEE Transactions on Dependable and Secure Computing* 17, 3 (2018), 506–517.

[6] Neo4j. 2012. Neo4j - The World's Leading Graph Database. http://neo4j.org/ Accessed: 2023.

[7] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. 2014. Relation extraction for inferring access control rules from natural language artifacts. In *Proceedings of the 30th annual computer security applications conference.* ACM, New York, NY, USA, 366–375.

[8] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. 2012. Automated Extraction of Security Policies from Natural-Language Software Documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12).* Association for Computing Machinery, New York, NY, USA, Article 12, 11 pages.