CS 5430
Access Control Analysis

Philip Ayoub (pja66) Nikolas Lillios(ncl44)

Fall 2021

1 Overview

This document outlines our implementation of the Access Control Analysis Algorithm (ACAA). This algorithm was implemented in Java for a number of reasons. First, is Java's space and time superiority compared to Python. The next is its ability to be run on a Linux Operating System. The final reason, and perhaps the most prudent, was our combined experience with the language. Our algorithm consists of 1 Main file that contains 5 fields and 5 helper functions. The primary goal of our design was to create a system that prioritizes speed over space efficiency. Thus many of our implementation choices sacrifice memory by storing duplicate values for the ability to constant and linear execution times.

2 Fields

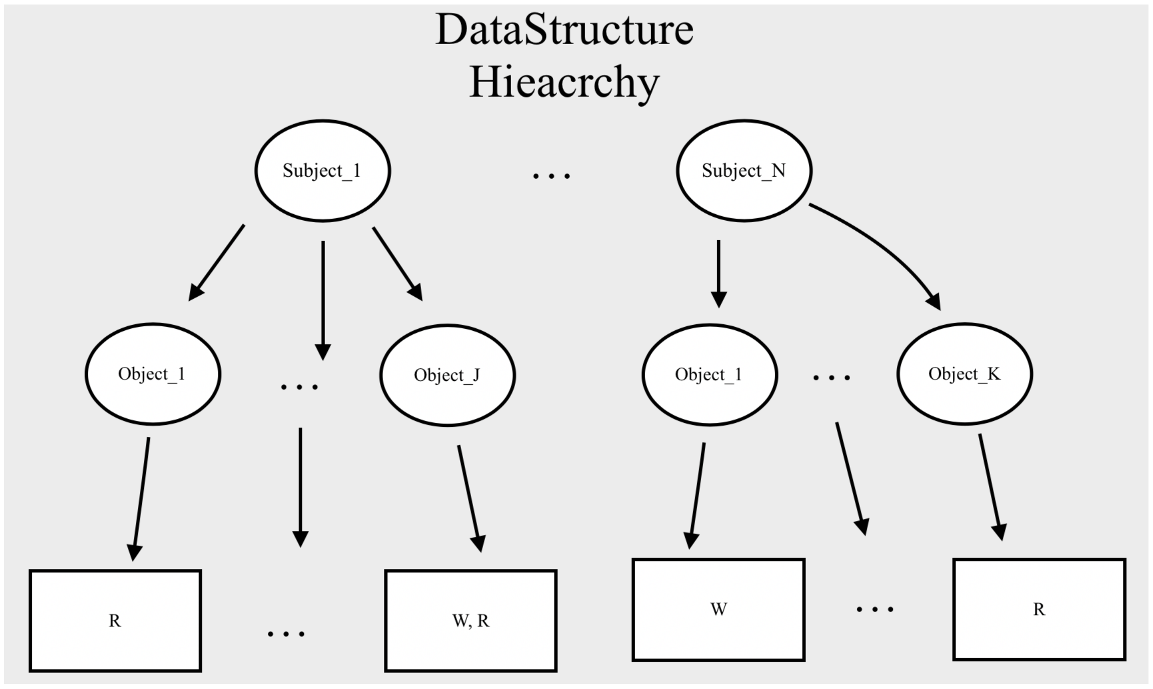
2.1 PrivMatrix

2.1.1 Data Structure

The heart of a ACAA is the data structure that represents the relationship between subjects, objects, and assigned privileges. We choose to use a *Access Control Matrix* to implement the given *Authorization Triples* (S, O, p). To represent the *Access Control Matrix* we used nested Hash Maps because of there O(1) look up and placement times.

HashMap(String, HashMap(String, List of Strings))

The first *String* represents a given *Subject*. The second *String* represents a given *object* and the *List of Strings* represents the list of privileges that are associated with that *subject* and *object*. Together all these values exhibit a primary key that represents every Authorization Triple given in the input file. Strings were an obvious data type for the *Subjects* and *Objects* since they represent the names of principles and files respectively. A list was chosen because you aren't aware during the execution of the program how many privileges a particular *Subject-Object* relationship will hold. Thus we decided to not over allocate space with a set length array. Some may criticize that Lists have a O(n) placement time but since you can only have a max of 2 *privileges*, the placement time is actually O(1).



2.1.2 Alternative Implementations

- **Access Control Lists:** We could have chosen to use *Access Control Lists* instead of a *Access Control Matrix*. However, *Access Control Lists* would be very inefficient. For comparable memory requirements to our implementation, the look up and placement times would be linear.
- **Capabilities:** Just like *Access Control Lists* the memory needed would be comparable to our implementation but the look up and placement times would have been much worse.
- **Other Access Control Matrix Implementations:** There are a number of ways to implement a *Access Control Matrix*. You can use a 3D array where each cell holds a list of privileges. You can do the same implementation with lists. You can also use the same data structure we implemented but with a different hierarchy (ie. Objects, Privileges, Subject). Each of these have there strengths and weakness. Using a 3D matrix is more space efficient then nested HashMaps, but has O(n*m), where n is the number of Subjects m the number of Objects, look up and placement time. In addition, there are various trade offs for using different hierarchy levels.

2.2 SubjectsSeen

2.2.1 Data Structure

SubjectsSeen is a field that stores the names of the *Subjects* that have appeared in past commands. This information is recorded for the purposes of command validation. To implement this field we used a HashSet because of its O(1) look up and placement time. The *String* represents the name of a *Subject*.

HashSet(String)

2.2.2 Alternative Implementations

An Alternative implementation to storing the names of the *subjects* is to search the subjects in the *Access Control Matrix* every command. We thought that the time wasted on cycling though the *Access Control Matrix* every potential command greatly out weighted the extra memory needed to store the names of the encountered subjects. Another option would have been to use a list or an array. However, the extra space a set uses, since its load factor is smaller than one, we thought was worth it to achieve the advantages of the O(1) look up time as opposed to O(n) with lists and arrays.

2.3 ObjectsSeen

2.3.1 Data Structure

ObjectsSeen is a field that stores the names of the *Objects* that have appeared in past commands. This information is recorded for the purposes of command validation. To implement this field we used a HashSet because of its O(1) look up and placement time. The *String* represents the name of an *Object*.

HashSet(String)

2.3.2 Alternative Implementations

An Alternative implementation to storing the names of the *Object* is to search the subjects in the *Access Control Matrix* every command. We thought that the time wasted on cycling though the *Access Control Matrix* every potential command greatly out weighted the extra memory needed to store the names of the encountered subjects. Another option would have been to use a list or an array. However, the extra space a set uses, since its load factor is smaller than one, we thought was worth it to achieve the advantages of the O(1) look up time as opposed to O(n) with lists and arrays.

2.4 TakeMap

2.4.1 Data Structure

TakeMap will store a map for each subject which will map to the other subjects that it is authorized to prerform a take on. It is implemented with a hashmap for efficient retrieval and storage. Both will be O(1).

HashSet(String)

2.5 OutputFile

2.5.1 Data Structure

OutputFile, which represents a temporary output file, is a List of Strings that holds the values that will be converted into a .txt file once the program completes. Each cell of the *List* represents a line that will exist in the **output.txt** file. List[0] is the first line, List[1] is the second and so on.

List(String)

2.5.2 Alternative Implementations

Another implementation of this would be to use an array. However this would require keeping a counter which would require memory additional memory. Since for our use the placement time is O(1), it made more sense to just keep things simple and use an *arraylist*.

3 Methods

3.1 InputFile

InputFile is a pretty straight forward method. It uses various built in libraries to transform the input.txt file into a *list of Strings*. Each cell represents 1 line of the input file. Thus if there are 20 lines in the input file, the *list* will be of size 20.

3.2 Concatenate

Concatenate is a key helper method that allows our other methods to preform efficiently. *concatenate* takes in one cell, which is equivalent to 1 line of the input file, of the *list of Strings* return by the *InputFile* method. It then returns a list divided into potentially 4 parts. Since there are only 2 types of *commands*, *Add* and *Query*, and they both consist of 4 values separated by a comma and a space we can divide the important information into a List with 4 cells. For instance, *concatenate* will take in a line ”Add, S1, O1, R” and returns list = Add, S1, O1, R. It uses built in methods to remove spaces and spilt on commas. This is a key method that helps in command validation as discussed next.

3.3 ValidInstruction

The main goal of *ValidInstruction* is to determine the validity of a potential Command. Each line in the input file can be two things: a comment or a command. Comments are just lines that are not syntactically correct Commands. For all intensive purposes comments can be ignored, however commands need to be processed and executed. This methods filters on 4 main constraints. The first is the size of the list that is returned by concatenate. If this list isn’t of size 4 then we know that this command is not syntactically correct by definition, and is thus a comment. The next is if the first cell of the list isn’t ”Add” or ”Query”. The third is if cell 4 isn’t ”R, W, or T.” Finally, the last area that is analysis is if there is a object where there should be a subject and vice versa. If all of these test cases pass, then we know that we have a properly written command and can go forward and execute it.

3.4 OutputFileGen

OutputFileGen takes as an input a *list of strings*, or in our case, the field *OutputFile*, and transforms it into the required output form, a txt file with the name ”output”.

4 Commands

4.1 Before every Command

Each line is sent through *Concatenate* and *ValidInstruction* to deem the category, *comment* or *command*, of the line. Both these functions take O(1) time.

4.2 Add

4.2.1 Overview

Add is one of the main instructions in this system. An *add* essentially places the privilege stated in the instruction into the *Access Control Matrix*. The algorithm uses multiple checks to make sure it is not placing duplicate information into the matrix.

4.2.2 Asymptotic Complexity

- Time: O(1) for inserting a new privilege to the PrivMatrix.
- Space: O(n*m*C), where *n* is the number of *Subjects*, and m is the number of Objects in the *privMatrix*. *C* is the load Factor that is greater than 1.

4.3 Query

4.3.1 Overview

Query is the other main instruction in this system. When triggered, the algorithm attempts to determine if there exists a sequence of take operations that would result in the specified subject having the specified permission on the given object. This is done by storing a HashMap of the take authorizations for each subject. Then, when a query is attempted, our analyzer will check all of the take authorizations for that subject. It will then repeat this process recursively for each other subject that it has take authority over to find the set of subjects S whose permissions could possibly be transferred to the given query target subject. At each subject along this search, the algorithm will check all of their permissions and compare to the target permission. If the queried object and permission is in this set it returns ”YES”, if it will continue running until it finds the target permission and returns ”YES” or the algorithm terminates because it has visited every subject where it will return ”NO”.

4.3.2 Asymptotic Complexity

- Time: worst case, O(nm), where *n* is the number of subjects in the *privMatrix* and m is the number of Objects in the *privMatrix*. This is because when searching the subject tree for take relationships, each subject can be visited at most once by depth first search. At each subject, we must check for all of their permissions to see if it matches the target permission. Because up to every n subject could have permissions for every m object, this will require up to nm constant time operations to complete the query algorithm.
- Space: worst-case, O(m+n), where *n* is the number of subjects and and m is the number of Objects in the *privMatrix*. This space complexity is due to the algorithm will store up to m *Objects* and n *Subjects* because it could store each subject up to once either in the seen subject set or the to be visited subjects on the stack.

5 UGCLab Running Times

Here we will provide the running times of our implementations of ’add’ and ’query when run on the UGCLab computers. These times were measured using java.lang.System.nanoTime() which is an accurate way to measure elapsed time because it is never adjusted to sync to wall clock time. We used to provided test file to test the running time of these implementations and then computed the average operation time for each of the two commands.

Command	Avg. Running Time (millisecs)
Add	0.012455
Query	0.383474

6 Test File

There are around 20 permutations of input lines:

{Add, Quote YES, Quote NO} x {R,W,T} x {Command, Comment 1, Comment 2, Comment 3} + {Comment 4}

Where:

- Command = Syntactically correct command
- Comment 1 = Misspelled/No *Add* or *Quote*
- Comment 2 = Misspelled/No R, W, T
- Comment 3 = *Object* where there should be a *Subject* belongs and vice versa
- Comment 4 = A plain text comment

Our test file, included in the .zip file, includes tests for each of these permutations. More comments are provided in the input.txt file.