COL106 : 2021-22 (Semester I)
Project: Module 2

Satwik Banchhor Chirag Bansal Venkata Koppula

August 27, 2021

Notations

Important: The question marked with a ♠ is the *lab-submission* problem. It is to be submitted via Moodle by 11:59PM on the day that you have your lab.
The order of nodes in a linked list is same as the order of insertion, i.e. first (respectively last) node refers to the node in the linked list which was inserted the earliest (respectively latest). For two strings a, b, we denote a.`concat`(b) by $a + b$ (that is, if $a =$ "Hello", and $b =$ "World", then $a + b =$ "HelloWorld").

1 Introduction : Authenticated Data Structures

The focus of this module is one of the main building blocks of any cryptocurrency: a *blockchain*. Blockchain and cryptocurrencies are usually discussed together, but blockchains can be used in various other scenarios. For this module, we will do the following:

- first, we will avoid the term 'blockchain' for now, and instead call it an *authenticated list*. Looking ahead, we will also consider other authenticated data structures.
- study a toy scenario where such authenticated data structures can be useful.

1.1 Toy Scenario

2 Assignment Questions

- `public classData`: This class has the following attributes:
  - `public String value`: a string representing the value of the `Data` object.
- `public class Node`: This class has the following attributes:
  - `public Node previous`: pointer to another node.
  - `public Node next`: pointer to another node.
  - `public String dgst`: a string representing the digest.
  - `public Data data`: the data contained in the node. [1]
- `public class AuthList`:This class has the following attributes:
  - `private static final String start_string`: a 64−character string representing the starting digest for all authenticated lists (`AuthList` objects).

[1] (venkata) instead of making the data a String, we can have it as an abstract datatype. This will be useful when we replace this with a Merkle Tree.

- `private Node lastnode`: represents the last node present in the authenticated list.
- `private Node firstnode`: represents the first node present in the authenticated list.

The class has the following methods:

- `public static boolean CheckList(AuthList current, String proof) throws AuthenticationFailedException`: returns `True` if all the nodes in this `AuthList` are valid and `current.lastnode.dgst = proof` otherwise raises `AuthenticationFailedException`. Here a node u is considered valid if the following property holds:

$$u.\texttt{dgst} = \begin{cases} CRF64.\texttt{Fn}(\texttt{AuthList.start\_string} + \text{“\#”} + u.\texttt{data.value}) & \text{if } u.\texttt{previous} = \texttt{null} \\ CRF64.\texttt{Fn}(u.\texttt{previous.dgst} + \text{“\#”} + u.\texttt{data.value}) & \text{if } u.\texttt{previous} \neq \texttt{null} \end{cases}$$

  where $CRF64$ is an instance of the class `CRF` (from the previous assignment) with `outputsize` $= 64$.
  [2]

- `public void InsertNode(Data datainsert, String proof) throws AuthenticationFailedException`: checks the authenticity of the list and the `String proof`, inserts a new node in the list with `data` corresponding to `datainsert`, and updates the digests of the nodes such that the updated list is also valid. Returns the digest of the last node of the updated list.
- `public String DeleteFirst(String proof) throws EmptyListException, AuthenticationFailedException`: checks the authenticity of the list and the `String proof`, deletes the first node (inserted earliest) of the list, and updates the digests of the nodes such that the updated list is also valid. Returns the digest of the last node of the updated list.
- `public String DeleteLast(String proof) throws EmptyListException, AuthenticationFailedException`: checks the authenticity of the list and the `String proof`, deletes the last node (last node) of the list, and updates the digests of the nodes such that the updated list is also valid. Returns the digest of the last node of the updated list.
- `public static Node RetrieveList(AuthList current, String proof, String data) throws AuthenticationFailedException, DocumentNotFoundException`: checks the authenticity of the list and the `String proof`, returns the first `Node` u of all the nodes v present in the `AuthList current` having v.`data.value` = `data`. If there is no such node u then raises `DocumentNotFoundException`.
- `public void AttackList(AuthList current String new_data) throws EmptyListException`: modifies `value` of the `data` field of the first node of the list and updates the digests of the nodes such that the updated list is also valid. Raises `EmptyListException` if the list is empty.

- `class StackNode`: This class has the following attributes:
  - `public Data data`: the data contained in the node.
  - `public String dgst`: a string representing the digest.
- `public class AuthStack`: This class has the following attributes:
  - `private static final String start_string`: a 64−character string representing the starting digest for all `AuthList` objects.

[2] (venkata) define validity of the chain.

– `private StackNode top`: `StackNode` at the top of the stack.

This class has the following methods:

– `public static boolean CheckStack(AuthStack current, String dgst) throws AuthenticationFailedException`: returns `True` if all the nodes in this `AuthStack` are valid and `current.top.dgst` = `proof` otherwise raises `AuthenticationFailedException`. Here a node u is considered valid if the following property holds:

$$u.\texttt{dgst} = \begin{cases} CRF64.\texttt{Fn}(\texttt{AuthList.start\_string} + \text{“\#”} + u.\texttt{data.value}) & \text{if } u \text{ is the only element of the stack} \\ CRF64.\texttt{Fn}(v.\texttt{dgst} + \text{“\#”} + u.\texttt{data.value}) & \text{if } v \text{ is the } \texttt{StackNode} \text{ below } u \end{cases}$$

where $CRF64$ is an instance of the class `CRF` (from the previous assignment) with `outputsize` = 64.

– `public String push(Data datainsert, String proof)throws AuthenticationFailedException`: checks the authenticity of the stack and the `String proof`, pushs a new node in the list with `data` corresponding to `datainsert`, and updates the digests of the nodes such that the updated stack is also valid. Returns the digest of the last node of the updated stack.

– `public String pop(String proof) throws EmptyStackException, AuthenticationFailedException`: checks the authenticity of the stack and the `String proof`, pops a node from the stack, and updates the digests of the nodes such that the updated stack is also valid. Returns the digest of the last node of the updated stack.

– `public String GetTop(String proof) throws AuthenticationFailedException`: checks the authenticity of the stack and the `String proof`, and returns the top node of the stack.

– `public static StackNode RetrieveStack(AuthStack current, String dgst, String data) throws AuthenticationFailedException DocumentNotFoundException`: checks the authenticity of the stack and the `String proof`, returns the first `Node` u of all the nodes v present in the `AuthStack current` having v.`data.value` = `data`. If there is no such node u then raises `DocumentNotFoundException`.

2.1 Exercises

Exercise 2.1. ***Authenticated lists***

Implementing methods of `AuthList` *(except* `AttackList`*): Given the design of an authenticated linked list, complete its implementation by writing the methods of the class* `AuthList` *described above.*

Exercise 2.2. ♠ ***Authenticated stacks***

Implementing methods of `AuthStack`*: Given the basic structure of an authenticated stack, complete its design and implementation by writing the methods of the class* `AuthStack` *described above.*

Exercise 2.3. ***Optional Question: Attack authenticated lists***

Implementing `AttackList`*: Given an authenticated list modify the data of the first node to a given string and appropriately update the digests such that the list passes the authentication test.*

2.2 Instructions

- Do not change the accessibility, names or signatures of the attributes and methods in the driver code. You may add your own attributes and methods to any of the above classes as and when required.
- The default constructor is used to instantiate objects of all the above classes. It is your responsibility to ensure appropriate initialization of the attributes of a newly created object.