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

Satwik Banchhor Chirag Bansal Venkata Koppula

August 27, 2021

Notations

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").

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.

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

Suppose you have a lot of documents on your mobile phone (referred to as the *client* for this document), and you want to store them on a server. You would like to perform the following basic operations:

- `AddDocument`: add new documents to the server
- `RetrieveDocument`: retrieve some document stored on the server

This is very simple - you can have a (doubly) linked list on the server. Each node in the linked lists contains one document. Every time you want to add a new document, you can insert a new node (containing the new document) to the end of your linked list. And every time you want to retrieve a document, you can go through the linked list, and retrieve the document (if available).

However, we would like to have an additional feature — we want to make sure that no one can tamper with your documents. That is, we want to make sure that if someone alters any of the documents on the server's linked list, then **the client should be able to detect this tampering**. Therefore, we also want to have a third operation: `CheckDocuments`. This operation ensures that if the documents on the server are modified, then the client can detect it.

A Naive Solution: First, let us consider a naive way of supporting all three operations:

- The client stores a copy of all documents, in the same order as they were inserted in the server's linked list.
- `AddDocument` and `RetrieveDocument` will be same as before. For `CheckDocuments`, the client will use its copy of documents to make sure that the server list was not modified.

Clearly, this will ensure that all tampering is detected. But it is also quite wasteful — the client needs to store the entire list to make sure that there is no tampering. Is there a better way to ensure that the server's list has not been tampered? Yes! There is an efficient way of addressing this issue, and the client only needs to store a 64 character 'proof'. The main idea is to use a collision-resistant function (CRF).

Pause here, and think about how to use a CRF to detect tampering.

A Simple Solution using CRFs: For simplicity, let us assume that each document is a long string of characters. Here is a simple solution: the client can concatenate all the documents, compute the CRF on this concatenated string (let `proof` be the 64-character string that is output by the CRF), and store `proof` on the client-side. Every time the client needs to perform `CheckDocuments`, it downloads the entire list of documents from the server, concatenates them, and checks if the output of CRF applied on the downloaded set (let us call this CRF output as `proof-downloaded`) is identical to `proof`. If it is not, then the client knows that some document has been tampered. If `proof` and `proof-downloaded` are identical, then we can assume that there was probably no tampering.

F.O.F.Y.

Suppose `proof`= `proof-downloaded`. Why can we conclude that there was probably no tampering?

Hint: In order to modify the documents such that the tampering is not detected, the adversary needs to find a collision for the CRF, and this is hard.

A Better Solution: In the above solution, the client needs to download the entire list on his/her device, concatenate the documents, and then apply the CRF on the concatenated string. Here, we we discuss a better solution where the client only needs to download one node at a time, do some processing on it, and then discard it. This is achieved by storing a short `dgst` at each node of the linked list. Therefore, each node of the linked list now contains the document (which we will refer to as the `data`), as well as a short `dgst`.

Pause here, and think about how you can store a meaningful `dgst` *at each node, such that tamper-detection can be performed by downloading/processing one node at a time.*

Computing the `dgst` in each node: Let `start_string` be some fixed string that the client chooses before adding any documents to the linked list (this can be any arbitrary string, say the client's name). Let $\texttt{data}_1$ be the first document that the client wishes to store on the server. The client first computes $\texttt{dgst}_1$ as the CRF output on `start_string`+ "#" + $\texttt{data}_1$. [1] The first node in the authenticated linked list will have $\texttt{data}_1$ as the data, and $\texttt{dgst}_1$ as its `dgst`. The client will store $\texttt{dgst}_1$ as the `proof`.

Next, suppose the client wishes to add $\texttt{data}_2$ to the server. It will first check that there is no tampering of the linked list on the server (this checking procedure is described in the paragraph below). Suppose it finds no tampering. Then it creates a new node, whose `data` is set to $\texttt{data}_2$, and the `dgst` is set to CRF output on $\texttt{dgst}_1$ + "#" + $\texttt{data}_2$. Let us call this CRF output as $\texttt{dgst}_2$. The client's proof is updated to $\texttt{dgst}_2$.

Deleting items from the authenticated linked list is also easy. As with insert, we will first check that there was no tampering of the linked list. Once that check passes, we simply delete the last node, and update the client's proof.

[1] Recall, the '+' operator here denotes concatenation.

What should be the client's new proof after the last node is deleted?

Checking the authenticated linked list In order to check the auth. linked list on the server, the client uses `proof` (which is stored on the client's device), and downloads each node of the linked list, one by one, starting with the last node. Suppose there are k nodes in the linked list, where $\texttt{Node}_i$ is the i^{th} node. Let $\texttt{dgst}_i$ be the `dgst` on $\texttt{Node}_i$, and $\texttt{data}_i$ the `data` on $\texttt{Node}_i$.

The client first downloads the last node $\texttt{Node}_k$, and checks that $\texttt{proof} = \texttt{dgst}_i$. If not, it outputs False. Else, for $i = k$ to 2, it does the following:

1. Download $\texttt{Node}_{i-1}$, and check that CRF output on $\texttt{dgst}_{i-1}$ + "#" + $\texttt{data}_i$ is equal to $\texttt{dgst}_i$. If not, it outputs False.

Finally, it checks that $\texttt{dgst}_1 = \texttt{start\_string}$ + "#" + $\texttt{data}_1$. If all these checks pass, it outputs True, else it outputs False.

F.O.F.Y.

Why is it necessary to perform all these k checks? Why does it not suffice to have just check that the `dgst` of the last node is equal to the `proof` stored by the client?

2 Assignment Questions

- `public class Data`: 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. [2]
- `public class AuthList`:This class has the following attributes:
  - `private static final String start_string`: a string representing the starting digest for all authenticated lists (`AuthList` objects). This string should be equal to your entry number.
  - `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:

  Predefined:
  - `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:

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

[2] (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.

where $CRF64$ is an instance of the class `CRF` (from the previous assignment) with `outputsize` = 64.
[3]

To be implemented:

- `public String 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 RetrieveNode(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 string representing the starting digest for all `AuthStack` objects. This string should be equal to your entry number.
  - `private StackNode top`: `StackNode` at the top of the stack.

  This class has the following methods:
  - `public static boolean CheckStack(AuthStack current, String proof) 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:

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

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

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

- `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.

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.