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

Satwik Banchhor Chirag Bansal Venkata Koppula

September 1, 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: There are two compulsory exercises, and one optional exercise. Exercise 2.2 (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: an *authenticated linked list*. We will first 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.

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

*Author names listed in alphabetical order. Many thanks to Pratik Kedia, Akshay Mattoo for help with this module.

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.

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 $\texttt{dgst}_i$ (resp. $\texttt{data}_i$) be the `dgst` (resp. `data`) on $\texttt{Node}_i$.

The client first downloads the last node $\texttt{Node}_k$, and checks that $\texttt{proof} = \texttt{dgst}_k$. 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.

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

Finally, it checks that $\texttt{dgst}_1$ is equal to the CRF output on $\texttt{start\_string}$ + "#" + $\texttt{data}_1$. 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 one check that the `dgst` of the last node is equal to the `proof` stored by the client?

Where are authenticated linked lists used? Authenticated linked lists, or its extensions, are widely used in practice. Below, we list a few such cases:

1. The toy-scenario described above is used in a lot of client-server based interactions.
2. It is perhaps the most fundamental entity in any cryptocurrency.
3. Authenticated linked lists can be used to build other authenticated data structures such as *authenticated stacks* (see the *Optional Reading* included at the end of Section 2.2 to understand why authenticated stacks are useful). You must implement an authenticated stack (this is the lab-submission question for this week).

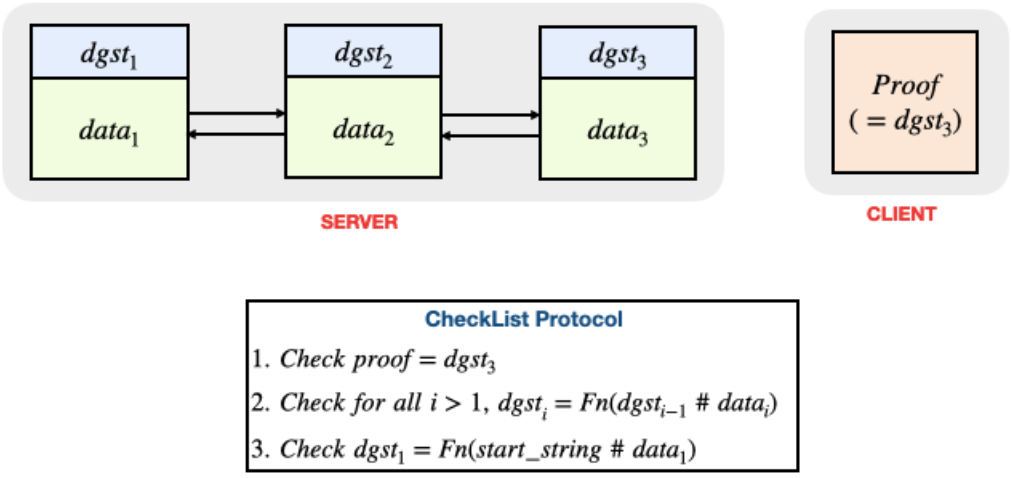

Figure 1: In the above figure, we have a server with three items stored in an authenticated linked list. The client only needs to store the `proof`. The `CheckList` protocol is described above.

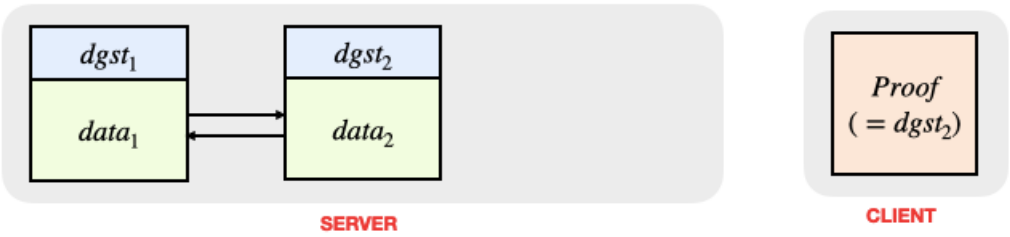

Figure 2: Next, we delete the last node from the authenticated list. Note that the server now contains two items, and the proof stored by the client is also updated.

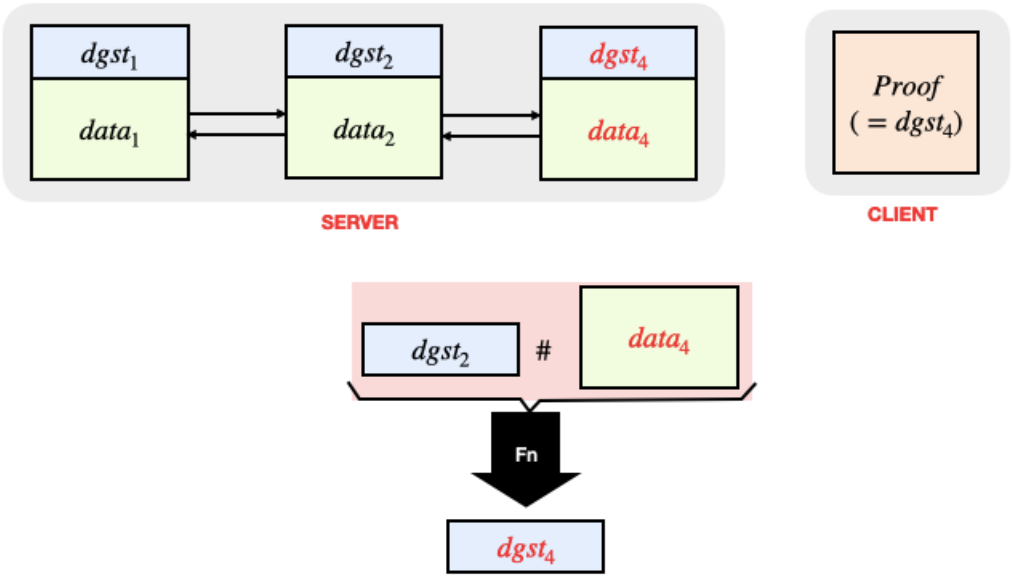

Figure 3: Finally, we insert a new item ($\texttt{data}_4$) to the authenticated list. The server now contains three data items, where the new digest is computed using $\texttt{dgst}_2$ and $\texttt{data}_4$. The client's proof is also updated.

2 Assignment Questions

You must solve all the problems below, preferably before your lab session next week. The lab evaluation questions will build upon these assignment questions, and therefore it is strongly recommended that you try these questions beforehand.

2.1 Authenticated Lists

You are given the following classes:

- `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.

- `public class AuthList` :This class has the following attributes:
  - `public static final String start_string`: a string representing the starting digest for all authenticated lists (`AuthList` objects).
    This string should be equal to your IIT Delhi entry number.
  - `public Node lastnode`: represents the last node present in the authenticated list.

– `public 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:

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

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

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, Data 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` $=_d$ `data`. If there is no such node u then raises `DocumentNotFoundException`. Here equality of two `Data` objects ($=_d$) is defined as the equality of all their attributes: ($d_1 =_d d_2 \iff d_1.\texttt{value} = d_2.\texttt{value}$).

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 following methods of the class* `AuthList`

(a) `InsertNode`*: this method takes as input an object* `datainsert` *of class* `Data`*, and a string* `proof`*.*

- *It first checks that the list is not tampered with. This is done using* `CheckList` *(and this is where the string* `proof` *is used).* [2]
- *If the above check passes, then create a new object of class* `Node`*. This object has four attributes — a pointer to the previous node, a pointer to the next node. Set these appropriately. It has an attribute of class* `Data`*. This is set to* `datainsert`*. Finally, it has a string attribute named* `dgst`*. This is computed using the CRF on an appropriate input.*

[2] Note that `CheckList` is a static method.

- *Finally, add the above* `Node` *object to the list, and update the* `proof` *appropriately.*

When inserting the first node, you must use the `start_string`***. This must be your IITD entry number.***

(b) `DeleteLast`*: this method takes only one input - string* `proof`*.*

- *It first checks that there is no tampering, and this is done using* `CheckList` *(this is where the string* `proof` *is used).*
- *If the check passes, then the method deletes the last node of* `AuthList`*. This updates the* `lastnode` *attribute of the* `AuthList`*.*
- *Finally, the method returns a string, which is the new proof that is stored by the client. This should be the* `dgst` *of the last node of the authenticated list.*

(c) `DeleteFirst`*: This method is similar to* `DeleteLast`*. It first checks the list, then deletes the first node in the list (note that we have a doubly linked list), and finally sends back the new proof. An important distinction here is that* ***the*** `dgst` ***of all nodes needs to be updated appropriately.***

(d) `RetrieveNode`*: This static method takes as input the list, an object* `data` *of class* `Data`*, and a string* `proof`*.*

- *It first checks the list using* `CheckList`*.*
- *If the check passes, it starts with the earliest node on the linked list (that is,* `firstnode`*), and checks if the node's data value is same as the* `value` *attribute of* `data`*. If so, it returns this node, else it moves to the next node.*
- *If no such node is found, it throws an exception.*

Useful predefined functions:

- `Fn`*: Compressing function defined in the* `CRF` *class in the "Includes" directory.*
- `CheckList`*: To check the authenticity of an* `AuthList` *and a string* `proof`*. This function is defined in the* `AuthList` *class.*

2.2 Authenticated Stacks

- `public 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 authenticated stacks (`AuthStack` objects).
    This string should be equal to your IIT Delhi 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.
- `public String push(Data datainsert, String proof) throws AuthenticationFailedException`: checks the authenticity of the stack and the `String proof`, pushes 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 StackNode GetTop(String proof) throws AuthenticationFailedException`: checks the authenticity of the stack and the `String proof`, and returns the top node of the stack.

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`*.*

- `CheckStack`*: This method will be similar to* `CheckList`*. It is a static method that takes an* `AuthStack` *object and a string* `proof`*. It first checks that the* `proof` *is identical to the* `dgst` *of the top* `StackNode`*. Next, it checks the* `dgst` *of each node (using the* `data` *of the current node, and the* `dgst` *of the node below it).*
- `push`*: This method will be similar to the* `InsertNode` *method in* `AuthList`*.*
- `GetTop`*: This method first checks the stack, and if the check passes, it returns the top node of the stack. Note that it does not remove the top node.*
- `pop`*: This method first checks the stack, and then removes the top node of the stack. It returns an updated* `proof`*. Is this method similar to* `DeleteFirst` *of* `AuthList` *or* `DeleteLast` *of* `AuthList`*?*

Note: Your implementation should be minimalistic, and not contain extra attributes/methods that are not needed. For instance, you don't need to store the bottom-most element of the stack, so your `AuthStack` ***should not have the corresponding attribute.***

Useful predefined functions:

- `Fn`*: Compressing function defined in the* `CRF` *class in the "Includes" directory.*

Optional Reading: Why are authenticated stacks useful?

In class, we discussed that stacks are useful for handling function calls. There is a special stack (called the call stack*) in the memory, and whenever a function is called, the function attributes, local variables, return address etc are pushed onto this stack. The return determines the program control flow after the function call is executed (that is, where the program must go after the function evaluation is completed). A popular security exploit is to alter the return address stored in the call stack, potentially forcing a malicious program to be executed. You can read more about it here. Authenticated stacks can be used to detect such tampering.*

2.3 Optional Question: Attack authenticated lists

Clearly, the security of our authenticated list is closely related to the size of the digest. In this optional exercise, you will show a tampering attack on the authenticated linked list.

- `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.

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.*

Each test case is associated with a digest size between 5 *and* 15*. In each test case, you are given a sequence of strings. You must first insert these strings in the authenticated linked list, and compute the corresponding* `proof`*. Then, you must alter the data on the first node, replacing it with the string* `new_data` ***without being detected****.*

Useful predefined functions:

- `Fn`*: Compressing function defined in the* `CRF` *class in the "Includes" directory.*

3 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.
- **Submission instructions for lab-submission problem:** You must create a directory whose name is your entry number, followed by "Module2" (for example, if your entry number is "xyz120100", then the folder name should be "xyz120100Module2"). The directory must contain `StackNode.java` and `AuthStack.java`. Finally, compress this directory, and upload it on Moodle. The file name should be `entrynumModule2.zip` (that is, `xyz120100Module2.zip` in the above example).

Acknowledgements

This is Version 2 of the module + supporting code (updated `CheckList`in `AuthList.java`, changed `readme` and access specifiers of `lastnode` and `firstnode` in version 0). Thanks to Aditya Agrawal, Vansh Kachhwal, Sreemanti Dey and Aditya Singh for finding the above typos.