Berkeley, Fall 2017, CS261, Week 3
Integrity for Outsourced Data Structures

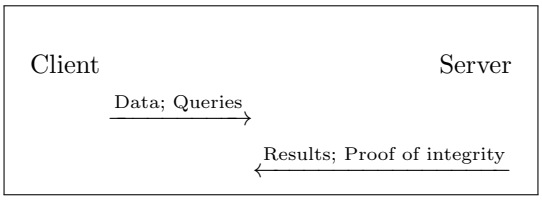
Scriber: Yuncong Hu

August 31, 2017

1 Introduction

This lecture mainly discusses how to outsource computation and show the integrity in the outsourced database model(ODB, an example of the well-known Client-Server model) using Merkle Hash Tree [1]. Compare to the last two lectures which is about how to outsource computation to remote server such as Amazon with encrypted data and keep the computation secret, in this class we focus on the case where the outsourced data is public and we only concern about the integrity.

Let's imagine there are some clients and some server. Clients will upload the data to the server because clients do not have enough space to store all the data, and then clients will send some operations and queries to the server. How could clients verify the response from the server is correct without having trusted servers?

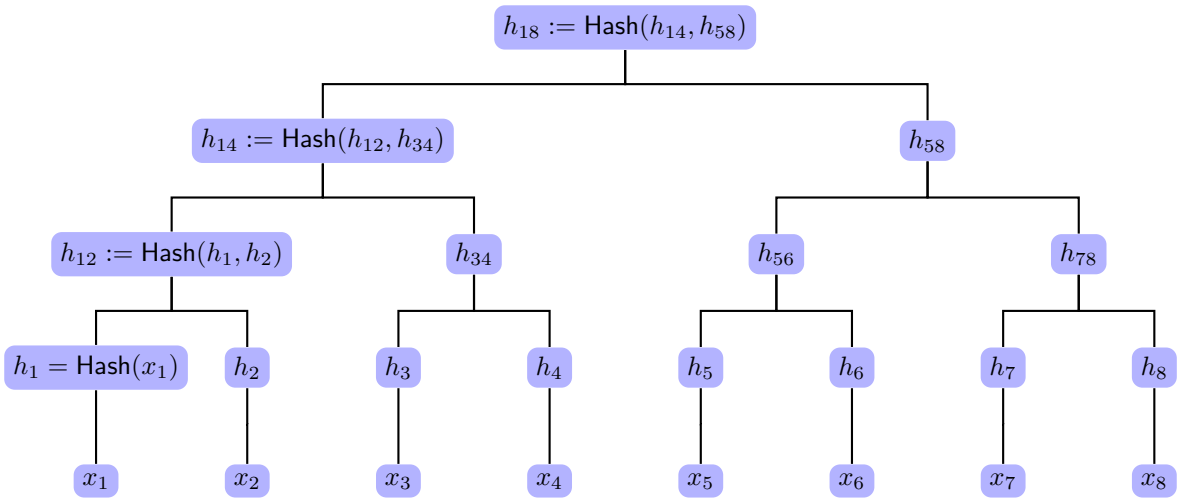


One possible solution is using SGX enclave. However, if there is a giant dataset, it is impossible to store all the data in enclave. The data must be encrypted and stored in the disk which is not part of SGX and untrusted. In this case, there is the same proble again where enclave is just the client for the disk server. So, applying SGX itself does not really solve this problem.

This class we first describe a practical scheme called the Merkle Hash Tree, which was first invented by Ralph Merkle [2], to provide data authentication and integrity in the ODB model, and finally present VerSum which support outsourcing expensive computations over large and frequently changing data structures.

2 Basic concepts of Merkle Hash Tree

Suppose that the entire dataset consist of n sorted values, $x_1, x_2, ..., x_n$. Assume n is a power of two, otherwise the dataset can be padded up to the nearest power of two incurring only a constant factor blowup in costs.



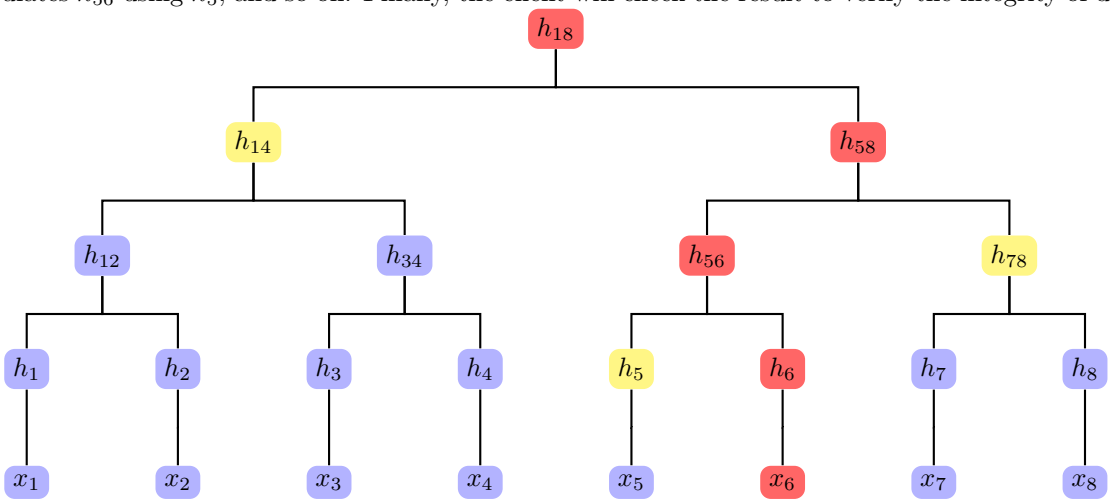
Each leaves of the Merkle Hash Tree will correspond to an original value. And each value will be hashed individually:

$$\text{for } i \in [n], h_i := Hash(x_i)$$

For each level in the Merkle Hash Tree, the hash value of each node can be computed as the hash of corresponding two children. Finally, clients get the root digest and servers will store the entire Merkle Hash Tree including original value and hashes of all internal nodes in the tree.

When the client want to read a single value from the server, the server will return the result and the corresponding co-path including all necessary nodes for calculating the root of the Merkle Hash Tree which enables the client to verify its signature.

For example, the client want to read x_6. The co-path is shown in following figure. The set of red nodes shows the path from x_6 to the root, and the set of yellow nodes shows the corresponding co-path. It is easy to see how the client calculates the root h_{18}. The client firstly get the hash of the result which is h_6, then calculates h_{56} using h_5, and so on. Finally, the client will check the result to verify the integrity of database.



Modification to the Merkle Hash Tree is similar to the read operation. The server returns to the client the co-path of the leaf node where a new value need to be updated. The client then recompute nodes from this leaf node up to the root, and generate a new root node with a new signature.

Note that the client does not need to store or maintain the whole Merkle Hash Tree, and it only need to store the hash value of the root node which can be viewed as the concise description of the whole dataset. And during the construction phase, the client may not have enough space for the whole dataset. In this

case, the client can build Merkle Hash Tree incrementally or ask someone else to build and upload Merkle Hash Tree.

Also, collision resistant hash function(CRH) is needed during the calculation process, because we want to protect against adversarially introduced errors. The formal defiinition is as follow:

Definition 1. *A set of functions* $\{h_k : \{0,1\}^{n(k)} \to \{0,1\}^{m(k)}\}_{k \in K}$ *generated by probabilistic polynomial time algorithm Gen where* $\forall i \in K, n(k) < m(k)$ *is a family of collision-resistant hash functions if:*

- $\exists$ *probabilistic polynomial time algorithm* E*, s.t.* $\forall k \in K, \forall x \in \{0,1\}^{n(k)}$, $E(k,x) = h_k(x)$
- *for all non-uniform probabilistic polynomial time algorithm* A*, there exists a negligible function* μ *s.t.*

$$\forall n \in \mathbb{N}, Pr\left(h_k(x) = h_k(x') \,\middle|\, k \leftarrow Gen(1^n); x, x' \leftarrow A(1^n, k), x \neq x'\right) < \mu(n)$$

Collision-resistant hash function can be constructed by cryptographic assumptions, such as Discrete Logarithm. SHA-256, SHA-384 and SHA-512 [3] are widely used in practice, while there is no provable security.

3 Static data structure with single client

In this setting, there is no update to the data structure and only one client will query the server. Suppose we want to support a set that

$$S = \{s_1, s_2, ...s_n\}$$

Query 1. *Given by* c_0 *and* c_1*, find* x *such that* $c_0 \leq x \leq c_1$*.*

Assume the set and leaves are sorted so that the leave nodes of result will be consecutive. The server can return all the leave nodes in the range(note that two nodes after and before this range should also be included to avoid server drop nodes in the edge) along with co-path of edge nodes to prove the integrity. Suppose the result set is k, then the communication complexity will be

$$O(k + log(n))$$

Query 2. *Suppose there is a mapping* $f : k_1 \mapsto v_1, k_2 \mapsto v_2, ..., k_n \mapsto v_n$ *for key value store, and given by a key* x*, the server need to find corresponding value* $f(x)$*.*

Actually this is same as the last case. The leave nodes contain the sorted key value and build a Merkle Hash Tree on them.

Query 3. *Suppose there is a table containing name, gender and birthday in the database, the client wants to do select query such as selecting all rows where birthday is inside a given range.*

name	gender	birthday

One solution is to make multiple copies of database sorted by each different attributes. However, it takes to much storage space and is time-consuming to synchronize all the copies if this is in the dynamic setting. To avoid tha space explosion, the server can build index on all the attributes and each index is mapped to a pointer of the corresponding row(note that the pointer is placed with the row hash). Finally, the server just need to build Merkle Hash Tree over different columns according to the index.

As for select query for union, intersection and joint, these kinds of query can be simply achieved by seperate steps and queries for different conditions.

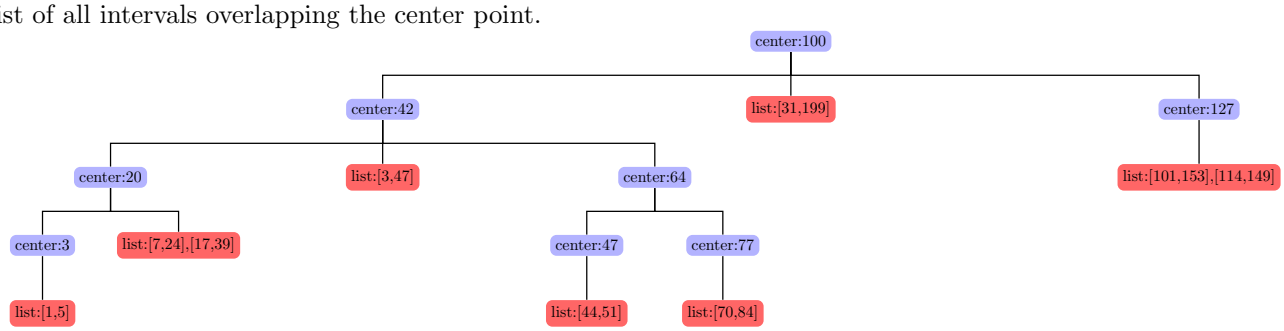
Query 4. *Suppose there is a set of many intervals:*

$$S = \{[l_1, r_1], [l_2, r_2], ...[l_n, r_n]\}$$

Given by x, the client want to find all interval $[l_i, r_i] \in S$ *such that* $x \in [l_i, r_i]$.

The simple solution is that the server can build Merkle Hash Tree over two different indexes of two ends, and do the intersection for $l_i \leq x$ and $x \leq r_i$. Suppose there are half intervals in the left side of x and half intervals in the right side of x, even if there is only one interval satisfying the condition, the running time is still $O(n)$ which is bad in practical.

The better solution is to use interval tree. The structure is shown in following figure(there are two akternative designs for an interval tree and the example shown here is the centered interval tree and another one is augmented tree [4]). Each node contains a center point, a pointer to the left, a pointer to right, and a list of all intervals overlapping the center point.



The construction steps is shown as follows:

Construction(ListOfIntervals)

LeftEnd ← The lowest value of left end in ListOfIntervals
RightEnd ← The highest value of right end in ListOfIntervals
CenterPoint ← (LeftEnd + RightEnd)/2
ListOfNode ← all intervals in ListOfIntervals cover CenterPoint
Construction(all intervals in ListOfIntervals completely to the left of CenterPoint)
Construction(all intervals in ListOfIntervals completely to the right of CenterPoint)

The server will build a Merkle Hash Tree and put hash value associated with each node which is the hash of left child, right child and the list. Also, the list itself is stored in the one of the Merkle Hash Trees. The overall running time will be about $O(k + log\ n)$.

This example shows how to build authenticated data structure. We first build a pointer based unauthenticated data structure and then turn it into authenticated data structure. If the running time for unauthenticated data structure is $T(n)$, the size of response from the server in authenticated case will be $O(T(n))$.

Sometimes we can not directly use this construction over some kinds of structure such as an array or a hash table. But there are still several ways to simulate the structure using pointer based structure. We can build index related to the pointer just like key store problem or simply use self-balancing binary search tree [5] instead of an array.

4 Static data structure with multiple clients

Since the structure is static, each client cannot modify the data and just do the same query indepently which can be reduced to the single client setting.

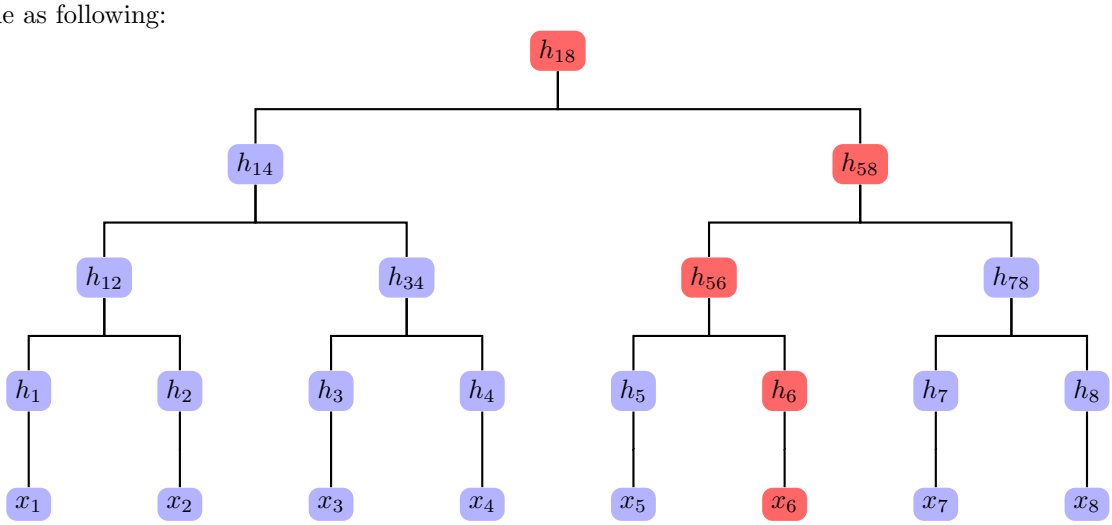
One example is about accountability. Let's imagine there is a lottery system. The system need to prove the result is truely random. The system need to publish the list of all participants with specified order and pick a random number to indicate the winner. Note that the list must be published before reveal the random number, otherwise the adversary may choose the order they want. To achieve randomness, the system may build a Merkle Hash Tree on the list of all participants' names and use the hash of the root produce the random number.

But there is still a problem(thanks to the question posted on Piazza). The organizer of the lottery system is able to try different names in the last few minutes, and is able to get the desired hash in advance. One possible solution is to ask participants sending random number after the organizer has committed to the list of participants. However, this solution might become complicated and need to be carefully analyzed in multi-party setting. For example, it is hard to force every participants sending the random number and it is possible for a group of participants to controll every bits of hash result. In practical, maybe we can take use of the hash of blockchain in Bitcoin since Bitcoin is a large scale decentralized system which is hard to be controled by a party without enough computation power [6].

5 Dynamic data structure with single client

Let's take an example of a set $S = \{x_1, x_2, ..., x_n\}$, and build a Merkle Hash Tree of the set. In dynamic setting, the data structure is supposed to support operations to modify the structure.

If the client want to change one of elements in the set, the server not only need to change the hash of the leaf node corresponding to this element, but also all the nodes on the path from this leaf node to the root node as following:



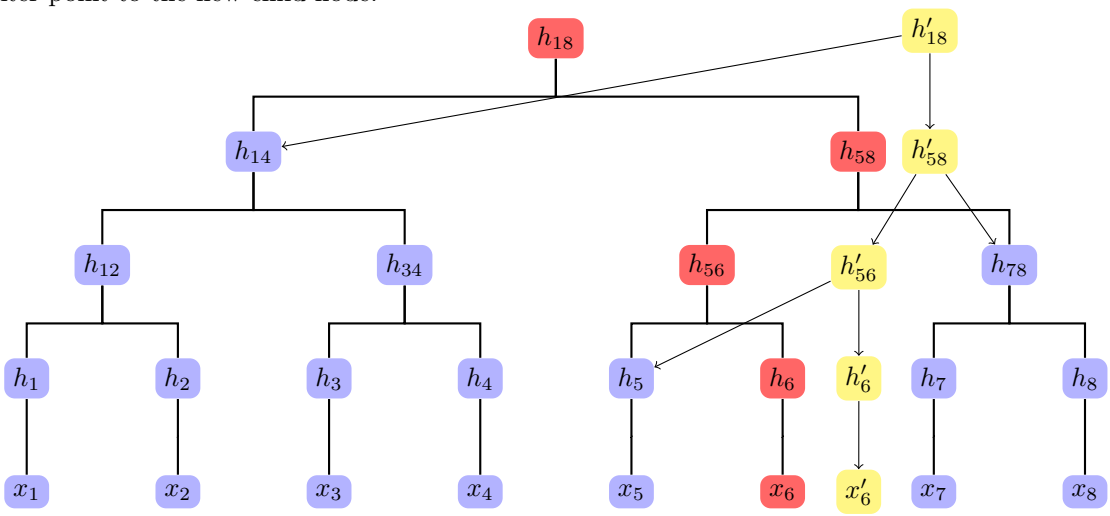
Note that we can compute new hash for each of red nodes through bottom-up approach and take use of old node hash so that we do not need to recompute hash for the whole tree:

1. First, replace x_6 with new elements x'_6, and compute new hash $h'_6 := Hash(x'_6)$.

2. For each internal node in red, notice that there are two kinds of chirdren nodes. One is exactly old hash and the corresponding subtree does not contain new element so that the old hash can be used in new version. The other one contains new element, however, through the bottom-up approach, we already have the latest hash of this subtree. Therefore, we can calculate the hash of this node directly using hash of two children nodes.

The running time of this algorithm will be $O(log\ n)$. In practical(also thanks to the question from Piazza), we want to publish the changed version with small space and also want to publish the state at each time point when building a log. Therefore, we introduce the use of functional persistent data structure. We

make a copy of each node on the path(for example the red nodes in the above), and make one of its children pointer point to the new child node.



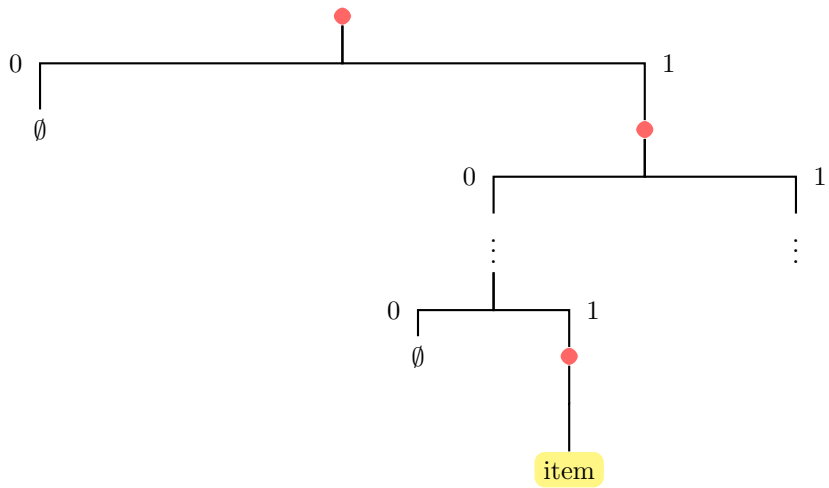
In fact, whatever data structure we want, we can build a functional persistent version of it which is log_n slower than the original version. That is because we can view the memory as an array, and then simulate the array with a self-balance binary search tree which can be persistent. So we can change corresponding static data structure into persistent structure with at most $log\ n$ factor blowup in costs for any dynamic, single client situations.

6 Dynamic data structure with multiple clients

In this setting, multiple clients can read and write the database. The challenge now is that someone else may append something while this client is not aware of it. The security property we want is that the client is able to check the membership in the log and verify the new data structure is consistent with the last time it has seen. There are a bunch of data structures to achieve this.

Imagine building a Merkle Hash Tree with space for total number of entries which the client might have, and then fill in entries the client had so far. When adding a new entry, the server need to update all the nodes along the path to the root. And it is easy to verify the new hash is consistent with the old hash in $log\ n$ time. Also $log\ n$ time to verify the item is present in the tree. But the thing clients can not do is to verify the item is missing without downloading the whole log, because we do not guarantee the entries is sorted in order in dynamic setting.

There is a different data structure which support proving non-membership, but unfortunately does not support consistency proof. In this structure, each item is treated as a big string(for example, 10...1), and put into a Trie [7] [8]. The item will be stored as following:



This will be a sparse Trie which contains lots of empty slots. When a subtree is empty, we just mark it as empty and do not compute anything below. To prove membership of item(10...1), server just need to show the corresponding path. As for non-membership, the server can just show the two neighbors which is also efficient. Note that in this structure, each item is represented as a unique string. But unfortunately, when someone else adds a new entry in it, there is no efficient way to prove the old root hash and new root hash are consistent. The client must download the old data structure.

So for certificate transparency [9], the basic idea is to build one of those data structures, and then relies on some third parties to periodically download the whole thing and verify consistency.

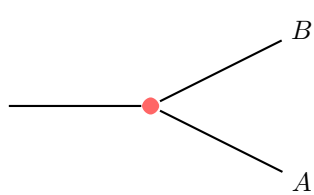
Due to the time limitation, we do not have much time to discuss further in this class, such as dynamic structure in file system. The next section will be the presentation given by Nick Spooner.

7 VerSum: Verifiable Computations over Large Public Logs

VerSum is a shceme for verifiable, efficiently, and updatable outsourcing of computation [10]. Verifable means the server can prove the computation is performed correctly. And also the computation might take a lots of time, while the verification is very efficent. Besides, updatable allows parts of input to be changed, and at the same time, the server does not need to recompute the whole authenticated data structure. Just as Merkle Hash Tree, when one of the leaf node is changed, only part of nodes need to be changed. Moreover, the verifiability of computation can be still preserved during this process in VerSum scheme.

Bascally, VerSum is a refereed delegation of computation(RDoC) system [11] which consist of lots of servers in this system. The guarantee is that at least one of these servers is not malicious.

We start from another RDoC system, known as Quin, for verifying the execution of Turing machines, applied to x86 binaries. Suppose there are two servers A and B where B is an honest server and A is malicious. They disagree on the output of computation, because one of them is cheated and the user does not who is malicious. At some time point, there is a diverge point between computation of server A and server B.



User can use binary search to find that diverge point and get the state of that point. Then it is easy for user to compute the next status and find out who is cheating using authenticated data structure.

However, there is a problem for Quin system. In Quin system, we can not reuse the computation because of the miss of some global states. Therefore, if one bit of the input is changed, the entire computation can be completely different from the beginning.

To solve this problem, VerSum takes use of functional programming language. Given by three functions:

$$\begin{aligned} F(x,y): &\quad \textsf{return} \quad G(x)+H(y) \\ G(x): &\quad \textsf{return} \quad H(x)+1 \\ H(x): &\quad \textsf{return} \quad x+2 \end{aligned}$$

By functional programming language, we can store and track the computation history as following:

Computation history of A	Computation history of B
Call $F(5,4)$	Call $F(5,4)$
Call $G(5)$	Call $G(5)$
Call $H(5)$	Call $H(5)$
Return 7	Return 7
Return 8	Return 8
Call $H(3)$	Call $H(4)$
Return 5	Return 6
Return 13	Return 14

Misbehavior happens in the middle where it is supposed to call function $H(4)$ while A calls function $H(3)$, and thus two servers output different result. We can use binary search to find the first deviation instruction and check which computation is correct. We can see that before this instruction, the prefix of two servers are the same. By functional programming language, we can avoid recomputing $G(5)$ or $H(5)$, and restart at $H(4)$ to continue computing.

For updatable, we might want to change some parts of input like changing $y = 4$ into $y = 3$. Then we need to redo the computation and recompute authenticated data structure. And we can see that all computation inside $G(5)$ does not need recomputation since x is not changed and it has nothing to do with the value input y. In fact, we only need to recompute the function relies on y.

We could benefit a lot from functional programming because we can track the computation history, however, it is not trivial to achieve these functional structures. For example, we might want to apply it on Merkle Hash Tree, and the problem is that when we reuse some computation history, we have to concatenate authenticated data structures efficiently. And the hash of the root in Merkle Hash Tree might depend on the order of bottom leaves so that we need to avoid the order of concatenation affect the result. And that is the reason why VerSum takes use of SeqHash, a hash-tree structure which supports fast positional indexing, fast concatenation, and is efficiently comparable.

Reference

[1] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Providing authentication and integrity in outsourced databases using merkle hash trees. *UCI-SCONCE Technical Report*, 2003.

[2] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[3] Fips pub 180-2, secure hash standard (shs), 2002. U.S.Department of Commerce/National Institute of Standards and Technology.

[4] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[5] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

[7] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[8] Peter Brass. *Advanced data structures*, volume 1. Cambridge University Press Cambridge, 2008.

[9] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. Technical report, 2013.

[10] Jelle van den Hooff, M Frans Kaashoek, and Nickolai Zeldovich. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1304–1316. ACM, 2014.

[11] Ran Canetti, Ben Riva, and Guy N Rothblum. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 445–454. ACM, 2011.