Fall 2017 | CS 2030 | UWYO

Growing a Forest of Data

The Creation of Binary and Merkle Trees

McKade Umbenhower & Robert Randolph mumbenho@uwyo.edu & rrandol3@uwyo.edu

Motivation/Abstract

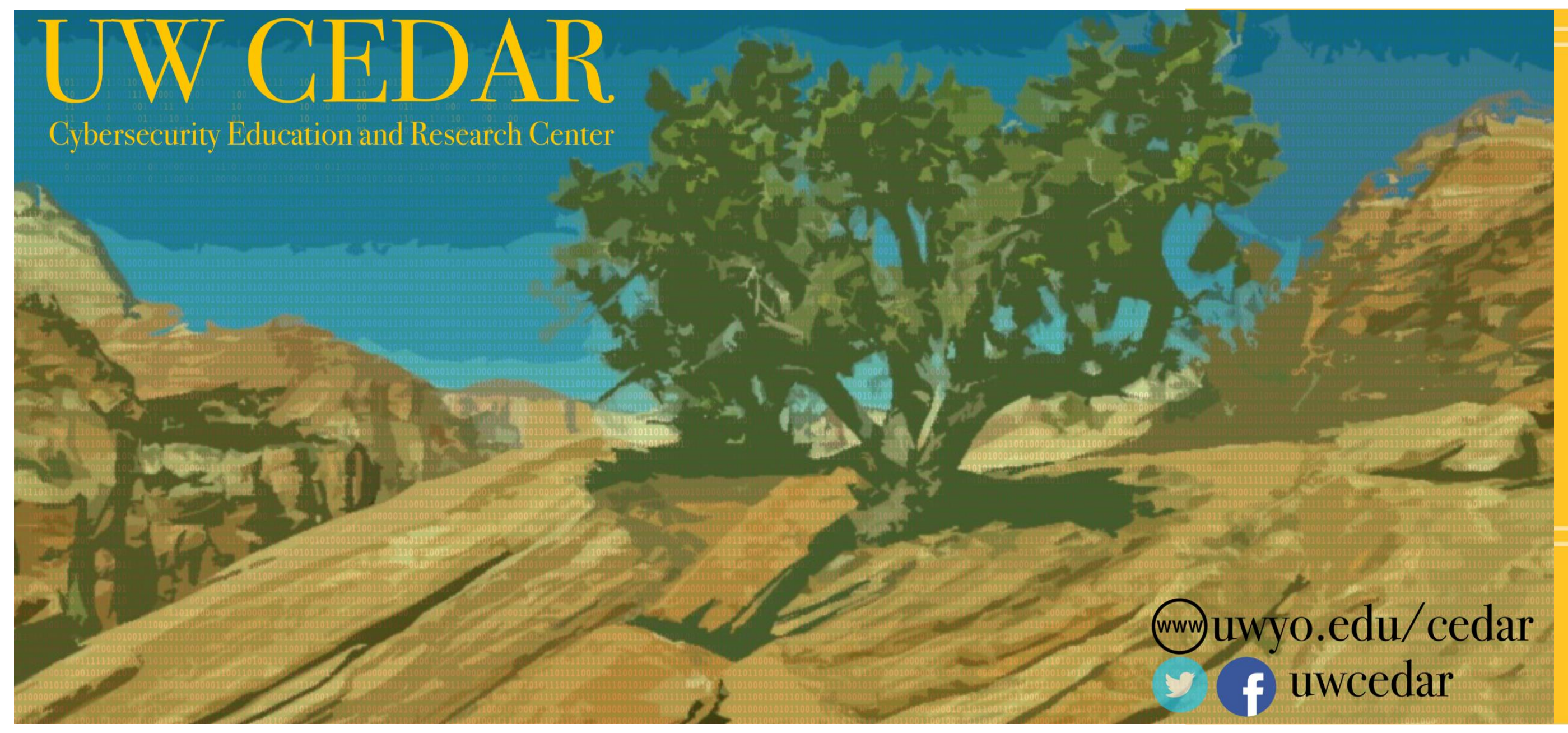
Merkle trees are invaluable data structures in computer science. With these trees, any collection of data can be verified by comparing it to an existing set of data with ease. Therefore, any deliberately changed or corrupted data can be detected and corrected. Having the experience of creating a Merkle tree is also important. While developing such a structure, the developers would become more familiar with the inner working of it, and its strengths and weaknesses would become more apparent. This project also exercises the ability to create an effective hashing function, which is another important component in the computer science field. Hashes provide many uses, and the having the experience of building and understanding such a function is indispensable.

Background

The Merkle tree is a type of data structure that uses a binary tree as a base, and hashes for every non-leaf node. The binary tree is a non-linier data structure that consists of nodes starting at a root node where each node has a parent and two children. The hash is a numeric value based off of data manipulation of a data entry that is being stored. The leaves of the Merkle tree are data nodes that hold pertinent information, and the non-leaves of the Merkle tree are hash nodes that hold a hash based off of the hash or data of it's child nodes. Due to hash nodes being dependent on its children, the hash is almost guarantied to be different from any other hash in the Merkle tree. Because of this property, Merkle trees are often used for quick verification and validity of data contained inside of large data-structures, as well as for peer-to-peer interaction between other computer systems. In the case of mismatching Merkle trees, the offending data node(s) can be easily found as the hashes of both trees, starting at the root node, have different values down through the subtrees that have the data node as a child or grandchild.

Project Summary & Major Tasks

To overcome the final goal of implementing Merkle trees, we needed a foundation to build upon, like a binary tree. We built this structure out of nodes that contained all of the vote data we would need to eventually store, alongside pointers to a left and right node. It was also important that leaf nodes could be differentiated between non leaf nodes, so we added functionality for this as well. To complete the binary tree we added the ability to do basic operations, like insert nodes into the tree, find nodes in the tree, and print the tree to the console. Upon completion of the binary tree, we were able to tackle the Merkle tree itself. A Merkle tree keeps all data in the leaf nodes, and all internal nodes are compromised of the hash of the hashes of its children added together. With this information, we realized that while the Merkle tree would have a similar structure to the binary tree, it needed a few of its own functions to operate correctly, so we made the Merkle tree inherit from the binary tree. The main difference between the two were the insertion methods. Our binary tree inserted in level-order, but our Merkle tree needed to insert time-sorted leaf nodes with parents of the correct hash values. To do this, we created a linked-list of leaf nodes that were cycled through until the insertion point was found. The new node was then partnered with an existing node, and a mutual parent was made between them. The parents were then set using hashes computed with our three different hash functions. Other functionality, like checking if and where two trees were different was then added, finishing the core implementation of the Merkle tree.



Limitations & Challenges

We succeeded in developing a fully functional Merkle tree, however a few challenges arose that needed to be solved in order for it to work at all. The first major problem we came across was how the Merkle tree would use the binary tree as a base, but still have it's own functionality separate from the binary tree. We decided on a hierarchy where the Merkle tree inherited from the binary tree. Another problem was figuring out a way to print out nodes that differed between two separate trees. Instead of using an operator to return a tree, we created a public function. This function, when passed a separate Merkle tree, would print the differing subtrees as well as the offending nodes.

3 Hashes

Arash Partow's DJB hash: The function takes in a string and returns the hash of the string. This hash integer starts out as some prime number. Then, for every character in the string, the hash will add to itself another copy of itself bit-shifted 5 bits to the left and the current character of the string. The result after looping through each character of the input string then gets converted to a hexadecimal string and returned.

Arash Partow's RS hash: This function uses two prime numbers, 'a' and 'b', alongside the hash integer that starts at 0. For every character of the string, the hash will equal itself multiplied by the value in 'a' (which initially is a prime number) plus the value of the current character. Then the value in 'a' gets multiplied by the prime number 'b', and the loop repeats for every character of the initial string. The hash value will then get converted to a hexadecimal string, and the result from that gets returned.

Arash Partow's DEX hash: It first initiates the hash value to the length of the string that was passed. Next, for every character of the string. It will take the current hash value bit-shifted left 5 times and raise it to the power of the hash value bit-shifted right 27 times. It then raises the previous result to the power of the numerical character value of the string in question. Finally it assigns the final result to the hash value. This repeats until every character of the string has been iterated through.

Results

Merkle Tree Operation Data
('m' is the number of leaves, 'n' is the number of total nodes)

Function	Number of Operations	For tree with 83 nodes.
Insert	O(m)	41
Find	O(m)	40
FindHash	O(N)	42
LocateData	O(N)	85
LocateHash	O(N)	42
Print_dif	O(N)	82

So what good is it?

Uses of the Merkle Tree

The Merkle tree we designed is able to effectively store time stamped votes retrieved from files in a way that makes comparison between two files trivial. After a file is inserted into a tree, one comparison between two strings is all that is needed to guarantee that the two tree structures are the same or different. This data structure also enables us to efficiently figure out which parts of two trees differ and the offending nodes, without necessarily checking every node of each tree.

Conclusions & Future Work

After completing the Merkle tree project, we have learned how to take an idea and implement it through rigorous planning alongside trial and error. We also have a better understanding of the benefits and limitations of the Merkle tree structure as a whole.

Looking toward the future, there are items that could be added to the tree to increase its usability. Actually displaying the structure of the tree instead of outputting a list of nodes would be a useful visualization tool. Also, some of our functions could be better optimized to increase the tree's performance.

UW

UNIVERSITY OF WYOMING

Acknowledgements:
Arash Partow's General Purpose Hash Function Algorithms

This poster made possible by:
UWYO College of Engineering & Applied Science, UWYO Computer Science Department, and UWYO CEDAR.