Rahul Mohandas
Dylan Fistrovic
Nicholas Bachman
CS426: Project 0

Trees...and Their "Chamber of Secrets"

Introduction:

This project encompasses aspects of information hiding, integrity verification, and forensic analysis on balanced search tree and Huffman tree data structures. Information hiding is a simple idea in which the literal structure of the mentioned trees is used to hold a secret "mark" that can be recovered and used to analyze whether or not the information hidden in the tree was modified by an "attacker". The report will explain the problem of hiding information into both tree structures using a simple key. The report will also explain how to verify the integrity of the information hidden in the structure of the tree using the key given to initially embed information in the tree. Finally, using a combinatorial group testing scheme, the project will explain how to discover the indexed position of the modified element of the tree if it was corrupted.

Problem:

The goal is to embed keyed information into each type of structure such that any persons possessing the key can validate the integrity of the embedded information in the balanced search tree and Huffman tree structures. As a secondary goal, persons possessing the key will also be able to identify the data element that was modified in a balanced search tree using a combinatorial group testing algorithm..

Solutions:

Balanced Search Tree Solution

Marking the Tree

The required information to mark the tree:

- The key to mark the tree
- The data elements given in sorted order

Step 1: Determine the mark

Option 1: Without combinatorial group testing

- Using the key and message we calculate the HMAC using 'SHA256' as our hash function
- The message is the data elements concatenated together

Option 2: With a combinatorial grouping mark

- The length of the mark is $1 + log(n)$ HMAC constructions.
- The first HMAC is constructed the same as Option 1.
- The remainder of the HMAC messages are constructed using this scheme $for\, j = 1, 2, ..., log(n)\ and\ i = 1, 2, ..., n,\ HMAC(K, D_{i,j=1})||...||HMAC(K, D_{i,j=n})$ $where\ D_i = D_k||..||D_n$ s.t. index of D possesses a '1" in the jth least significant bit

Step 2: Construct the tree

- Starting at the root node we calculate the capacity and lower-bound at that node
- Capacity is $floor(log_2(ceiling(\frac{n}{2})))$ and lower-bound is $ceiling(\frac{n}{4})$ where n is the amount of elements at that node
- If we find the capacity to be "0" we set our offset to "0". If not, our offset is capacity amount of bits from the mark in integer form.
- The portion of the mark used for the split is then placed in that node.
- The data element in which we split is indexed at $lowerbound + offset$
- The left child will contain all elements with values less than our split value
- The right child will contain all elements with values greater than or equal to the split value
- We continue running this scheme on the left child then the right child in breadth-first search order.

<u>Validating the Tree</u>

The required information to validate the integrity of the tree:

- The structure of the tree given in breadth-first search order.
- The data elements in the leaves of the tree given in breadth-first search order.
- The key of the structure.

Step 1: Construct the skeleton of the tree

- Using the given order of the nodes, a tree was constructed in the scheme of a "1" representing an internal node and a "0" representing a leaf node in breadth-first order.
- The given data was also placed in order into the leaf nodes when they were reached using the aforementioned scheme.

Step 2: Determine the number of leaves in each given node's left and right subtrees

- A count is run recursively down the tree determining the total of left and right subtrees each node possesses.
- When a leaf node is reached, that node returns a value of "1".

Step 3: Extract the mark

- Starting with the root of the tree, a breadth-first search is run calculating the index of the initial split value for embedding the mark when the tree was first constructed.
- The index is calculated by comparing the number of leaves in each left subtree in relation to a lower-bound constraint.
- The lower-bound constraint is determined by the number of data values (divided by four) that would reach that particular node upon first construction and embedding of the tree.
- The mark is then extracted from the index minus the lower-bound constraint represented as a bit-string and padded to a length that can represent all the distinct data elements in the tree.
- The mark is concatenated with the other node's marks in breadth-first order.

Step 4: Detecting modification of embedded mark

- The mark extracted from the tree is then compared to a newly created HMAC made from the given key and data.
- The new HMAC is either calculated as a single iteration with all of the data items or in a combinatorial grouping scheme that results in several HMACs that can be used to pinpoint the corrupted item.

Step 5: Determining which indexed data element has been modified

- Given that the tree was marked in a "combinatorial grouping" fashion of $1 + log(n)$ HMACs, the newly created mark (mark') is compared to the mark extracted from the tree (mark) starting from the $1 + log(n)$ HMAC message position of mark' to the same position in the mark.
- If the two marks match at that segment, a "0" is placed in the most significant bit of the binary representation of the corrupted index we are trying to construct. Otherwise, if the marks do not match at that position, a "1" is placed instead.
- The process repeats backwards for the remaining $log(n)$ HMACs constructing the binary representation of the corrupted index from most to least significant bit.
- The binary representation is then produced and converted to an integer representing the corrupted data element's index in the initial ordered set of data.
- Note: The indexing of the data elements proceeds from 1 → n.

Huffman Tree Solution

<u>Marking the Tree</u>

The required information to mark the tree:

- A given key.

- A given message.

Step 1: Determine the mark

- Using the key and message, calculate the HMAC using ‘SHA256’ as our hash function
- The first 255 bits of the 256 bit hash are used as the mark.

Step 2: Calculate Frequencies of the message symbols

- Where message symbols are the byte values of 0→255.

Step 3: Build the Tree

- Using the list of symbols and their corresponding frequencies. Sort the list in ascending order.
- Add the nodes with the two lowest frequencies together and concatenate the left and right symbols of the node together to create a new parent node for those items.
- Remove those two items from the frequency list after setting them as the left and right children of the newly created parent node..
- Add the new item to the frequency list.
- Repeat steps above until one item remains in the list that represents the root node.
- Note: When sorting the list, a ‘custom comparator’ must be used. The custom comparator will compare nodes based on these three properties in order: the frequency of the two nodes, the length of the super symbol of the two nodes, and finally the first character of the symbol.

Step 4: Embed the mark into the tree..

- Traverse the tree in the breadth-first search fashion starting with the root node.
- Compare the given node’s children using the custom comparator described above.
- Using the mark in order from first to second-to-last bit.
    - if the bit is a 1 place the least weighty child (as indicated by the custom comparator) on the right
    - if the bit is a 0 place the least weighty child on the left
- Continue until all children have been compared.

Step 5: Compress the message

- Determine the compressed value of each leaf node by traversing the tree in a depth-first search order to that node and storing the path.
- The path is built by a traversal to a left subtree representing a “0” where a traversal to a right subtree represents a “1”.
- Using the compressed value for each symbol, traverse the original message and replace the symbol with the corresponding compressed value.

Validating the Tree

Required information to validate the tree:

- A given key.
- A given tree structure with corresponding message symbols and compressed message

Step 1: Construct the skeleton of the tree

- Using the given order of the nodes, a tree is constructed in the scheme of a “1” representing an internal node and a “0” representing a leaf node in breadth-first order.
- The given message symbols are also placed in order into the leaf nodes when they were reached using the aforementioned scheme.

Step 2: Decompress the message

- Using the given compressed message, traverse the tree starting from the root.
    - If the bit in the compressed message is a “0”, take the left branch
    - If the bit in the compressed message is a “1”, take the right branch
- Continue using the next bit in the decompressed message until a leaf is reached.
- When a leaf is reached, save the symbol of the corresponding leaf into a list, and continue from the root until there are no more bits in the compressed message.

Step 3: Calculate the frequencies of the message symbols using the decompressed message, and save these values in the leaf nodes.

Step 4: Calculate values of inner nodes of tree.

- Traverse the tree in a breadth first search fashion. (Ignoring leaves in exclusion list)
- Combine the last two leaves frequencies, and symbols (left symbol + right symbol), and save these values in the parent node, and place the two leaves in the exclusion list.
- Repeat until only one node is found (the root)

. **Step 5:** Extract the mark

- Traverse the root in a breadth first search fashion.
- Compare the children of the node using the custom comparator.
    - If the left child is smaller than the right child
        - The next bit in the mark is 0
    - If the right child is smaller than the left child
        - The next bit in the mark is 1

Step 6: Compare the mark determined in step 5, with the first 255 bits of the HMAC. The HMAC is determined using decompressed message, and they key from the keyfile.

Conclusion:

Information hiding in the structures of these trees leads to a simplistic way of validating said information easily determined by the literal structure of the given tree. For both balanced search trees and Huffman trees, a simple modification is easily detectable. However, pinpointing the exact position of corruption was only made possible in a balanced search tree using the given “combinatorial group testing” scheme. Also, for simplicity, the scheme assumes that only one element is ever modified at a given time which doesn’t translate well to ‘real-world use’. Overall, it’s an interesting and simplistic set of ideas for information hiding, integrity verification, and forensic analysis on these particular types of trees.

Division of Labor:

Nicholas Bachman:

- Responsible for validating Balanced Search Tree structure.
- Responsible for creation of “combinatorial group testing” scheme and its use to mark, validate, and pinpoint corruption.
- Responsible for introduction, problem statement, corresponding validation/pinpointing-corruption solutions, and conclusion of paper.

Dylan Fistrovic:

- Responsible for Huffman Tree Solution
    - Mark the tree
    - Validate unmodified tree
    - Detect modified tree
    - Compress data
    - Decompress data
- Debugged CGT issues

Rahul Mohandas:

- Responsible for Balanced Search Tree Solution
    - Building the tree
    - Marking the tree
    - Wrote HMAC interface
    - User interface to run Huffman and Balanced Search Tree