Overview

The project combines ideas from integrity verification, forensic analysis, and information hiding. Its goal is to embed keyed information in data structures, that, while achieving the desired functionality of the data structure, makes it possible for anyone who has the key to (i) detect whether there has been any modification of the information stored in the data structure (i.e., was there any tampering with the information?); and (ii) pinpoint where in the structure the tampering took place (for simplicity, we assume the adversary will make at most one unauthorized change).

Part 1: Integrity verification

For this part of the project the goal is to write software for creating data structures, with embedded information, that will subsequently be used to determine whether the original data was modified or not.

Part 1a: A balanced search structure

Given n sorted and distinct data items $d_1 < d_2 < \cdots < d_n$, the goal is to write software that creates a binary tree T that

1. has $O(\log n)$ depth because, for every node v of T, if the subtree of v contains ℓ items then the item at v has rank between $\ell/4$ and $3\ell/4$;
2. has the "search property" (i.e., the item at each node v of T is greater than any item in v's left subtree and smaller than any item in v's right subtree);
3. encodes, in the manner we describe next, a hash-based message authentication code (HMAC) of the concatenation of all the d_is (in sorted order).

The encoding is based on the choices available for which item to put at a node: For a node v whose subtree contains ℓ items, there are $\ell/2$ candidate items for the root, which provides a way of encoding $q = \log(\ell/2)$ bits in that choice: The choice of the leftmost candidate (of rank $\ell/4$) corresponds to integer 0 (hence a bitstring of q zeroes), whereas the choice of the rightmost candidate (of rank $3\ell/4$) corresponds to integer $\ell/2$ (hence a bitstring of q ones). If we denote by s_v the bitstring encoded at node v (in the above manner), and $v_1, \ldots, v_n$ the breadth-first search order listing of the nodes of T, then the bitstring for the whole tree (call it θ_T) consists of the concatenation of the bitstrings of the v_i's, that is:

$$\theta_T = s_{v_1} || s_{v_2} || s_{v_3} || \cdots || s_{v_n}$$

In the test data that we will provide, n will be large enough for θ_T to be much longer than the length of the HMAC: In that case store as many copies of the HMAC as will fit in θ_T.

You have to write software that, given the n sorted items and the HMAC key K, produces the tree T with the desired properties. You have to also write software that, given a properly created T, detects whether or not any modification has occured.

Part 1b: A Huffman coding tree

Given a message M (as an array of bytes) and an alphabet Σ of n symbols along with their respective probabilities, the goal is to write software that creates a Huffman tree T that

1. corresponds to a prefix-free binary code with minimum expected codeword length;
2. encodes, in the manner we describe next, a hash-based message authentication code (HMAC) of the message being compressed.

The encoding is based on the choices available for how to order the children of each node: For a node with ℓ children, there are $\ell!$ candidate orderings of the children, which provides a way of encoding $log(\ell!)$ bits in that choice. For a binary tree ($\ell = 2$), there are 2 orderings and 1 bit of capacity. Choosing the least weighty child as the left child corresponds to the bit 0, whereas choosing the least weighty child as the right child corresponds to the bit 1. If we denote by s_v the bit encoded at node v (in the above manner), and $v_1, \ldots, v_n$ the breadth-first search order listing of the nodes of T, then the bitstring for the whole tree (call it θ_T) consists of the concatenation of the bits of the v_i's, that is:

$$\theta_T = s_{v_1}||s_{v_2}||s_{v_3}||\cdots||s_{v_n}$$

In the test data that we will provide, n will be large enough for θ_T to have length equal to the length of the HMAC. There is room to store just a single copy of the HMAC.

You have to write software that, given the message M and the HMAC key K, produces the tree T with the desired properties and compresses M using T. You have to also write software that, given a properly created T and a compressed message C, detects whether or not any modification has occured.

Part 2: Pinpointing the corrupted item

This part will use the binary search tree from part 1a. Because for large n the θ_T is so much longer than the number of bits in an HMAC, this part consists of storing $1 + \log n$ HMACs in the θ_T, using the scheme presented at

http://www.cs.purdue.edu/homes/mja/hwks/cgtforproject.pdf

You have to write software that, given the n sorted items and the HMAC key K, produces the tree T with the $1 + \log n$ HMACs stored in its θ_T (as many times as will fit). You have to also write software that, given a properly created T, pinpoints which item (if any) has been modified.

Report

You also have to write a report about the project and your solution. The report should be no more than 15 pages and should include an introduction, the problem statement, a discussion of your solution, and a conclusion.

Notes

1. Students should work in groups of 3. Group membership should be reported to the TA by January 25. Groups of size less than 3 must be approved by the TA or Professor.

2. Source code must be written in Java. Students may use whichever IDE they prefer. The TA recommends Eclipse.

3. The report and source code are due on April 12. Late submissions will not be accepted.

   - Submit by email to pritchey@cs.purdue.edu
   - Subject of email: "CS 426 Project - Group X", where $1 \leq X \leq 17$ is your group number.
   - Attach a single compressed file (.zip or .gz) that contains your submission.
   - Include a README file with instructions for how to run your code.
   - Submit ONLY .java source files, .pdf report, and README file.
   - Your code **must** run on the lab computers.
   - One (1) submission per group. Please only submit once. And, whatever you submit should be on behalf of your entire group.

4. Project demonstrations will be held April 15 – 26. Each group should set up an appointment with the TA before April 15th (i.e. before April 15th, have an agreement between group members and TA about when the group members will demonstrate the project to the TA). By default, project demonstrations will be during PSOs on 4/15, 4/19, 4/22, and 4/26.

5. Check the TA website and attend PSOs to get updates to the project requirements. Notice will be given on the TA website of particularly useful PSOs.

6. The TA website will provide detailed specifications regarding to the formatting of input and output.

7. Use Java's Hmac256 library to compute HMAC values.

8. The tree encoding scheme to use is posted on the TA webpage.

9. For part 1b, please also output the decompressed message to a file.