Trial exercise for Guardtime: Merkle tree signing of log files

Madis Ollikainen

July 10, 2016

In the following I'll shortly describe my code produced for the trial exercise from Guardtime.

Task overview

A binary hash tree or a Merkle tree is a binary tree, where all the nodes are constructed via hashing their child nodes

$$\text{parent node} \leftarrow hash(\text{child node 1}, \text{child node 2}). \tag{1}$$

For log file signing one can construct a Merkle tree, whose leaves are the hashes of the lines of the log file. This allows for verification procedures for both the file as an whole as well as individual lines in the file. For individual lines, a hash chain starting from the leaf and ending at the root must be extracted. My trial exercise was to construct a toy tool for log files signing, which would enable:

1. Signing of an arbitrary text file via signing the root of a binary hash tree (Merkle tree), whose leaves correspond to the hashes of the lines in the file.
2. Extraction of hash chains (from leaf to the root) for arbitrary lines in the text file.

It was noted, that the signing processes itself can be implemented as just an empty function (a commented call to Guardtime SDK could also be added). The main part of the task was to implement the hash tree construction and hash chain extraction. Especially taking into account that the number of lines in a log file doesn't have to be a power of two.

Algorithm & code description

One of the key points in the task was the selection of an suitable tree structure for the Merkle tree. The tree should be easily constructible for any number of leaves without sacrificing too much of computational efficiency. I chose to use the *canonical binary tree* introduced in Buldas et al. (2014). Such an tree can be constructed in an *on-line* manner, without previously knowing the number of leaves. The canonical Merkle tree used can neatly defined by the construction procedure:

1. The leaves (hashes of the log file entries) are added from left-to-right.
2. Moving from left-to-right the leaves will be gathered into a forest of complete trees.
3. All of the complete trees will be as large as possible with the currently available leaves. Due to the above mentioned process it is clear that larger trees will be on the left and smaller on the right.
4. When parsing a file (or any other input entity) is complete and no more leaves (entries) are added to the tree, the resulting forest of complete trees can be merged into a single *canonical*

tree. This merger is done by merging the root nodes of trees in the forest from right-to-left. Thus first the two smallest trees are merged to form a larger tree, which is then in turn merged with the third smallest tree. This procedure is repeated until all of the trees have been merge into one.

References

Ahto Buldas, Ahto Truu, Risto Laanoja, and Rainer Gerhards. Efficient Record-Level Keyless Signatures. *Lect. Notes Comput. Sci.*, 8788:149–164, 2014. doi: 10.1007/978-3-319-11599-3.