Trial exercise for Guardtime: Merkle tree signing of log files

Madis Ollikainen

July 11, 2016

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

1 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 a 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.

2 Algorithm description

2.1 Merkle tree structure

One of the key points in the task was the selection of a 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 a tree can be constructed in an *on-line* manner, without previously knowing the number of leaves. The canonical Merkle tree used can be 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 merged into one.

2.2 Merkle tree root calculation

As noted in Buldas et al. (2014), for calculating the root of the canonical binary tree, only the roots of the complete trees in the forest (see section 2.1) have to be kept in memory. During the forest creation, some additional information is needed to make sure, that after each leaf is added, the complete trees in the forest are updated correctly. Fortunately, this extra information can be neatly encoded into the layout of the array-like data structure storing the roots of the forest.

Let's consider the forest of complete trees. Note, that every time a new leaf is added it will either become the root of the smallest tree in the forest (tree of height 0) or if there already was a tree of height 0 in the forest, it will be merged with this tree (it must have been the previously added leaf). In the case of merger, if there was no other tree with height 1, the new tree will now be the smallest tree, but if there already was a tree of height 1, these two trees will be merged. Similar logic will continue recursively. We can see, that for each possible height h there can never be more than one tree of such height in the forest.

Thus, the necessary information for correctly updating the forest after addition of a new leaf is reflected in knowing for which height there is a tree in the forest and for which there isn't. When storing the roots of the complete tree in an array-like data structure, the height of a tree can be encoded by the position of the root in the array. In such a case the size of the array would be set by the largest tree. The absence of a tree of height $h < H$, where H is the height of the largest tree, can be marked by a suitable specifier being placed at position h. In my code, where I'm using C++ standard library strings for holding the hashes, this specifier is an empty string. In some other setting is could be something else, a null-pointer for example.

In my code the array-like data structure is C++ standard library vector of strings, which was chosen for ease of implementation. During the forest creation, my code is reading the log file line-by-line. It hashes each of the lines and adds it into the forest as a new leaf. The forest is updated via looping over the vector holding the roots. For each step in the loop, the following rules are implemented:

(a) The loop is exited when the first empty string is found. Finding of an empty string meant that there was no tree of that high in the forest before. Thus the new tree will have this height and its root value is set to this position.

(b) For every entry, which has a non-empty string, the current agglomerated root value is hashed together with that non-empty root value. This meant that there already was a tree of this height in the forest and these two trees could be merged into a new larger tree. The non-empty string value is changed to be empty and the loop goes on.

(c) If the end of the vector is reached without finding any empty string, then that meant that there already was a tree of every height from 0 to H. Thus the new tree will now be of height

$H + 1$ and the agglomerated root will be pushed to the back of the vector, increasing the size of the vector by one. Now we have a vector where for only the last value there is a non-empty string, for all previous values there will be an empty string.

After the end of the log file is reached, the forest of complete trees has been assembled and its roots vector can be used to merge the forest into a single tree. This merger is done by yet again looping over the roots vector. Note that the roots in the beginning of the vector correspond to the small trees and the roots in the end correspond to the large one. Thus the merger is indeed done from the smallest to the largest (right-to-left). This time the loop is rather straightforward. The variable holding the final root value is initialised by the first non-empty value in the vector. Every empty string is just ignored. Every non-empty string is hashed together with the current value of the "global root". In the end we get the root value of the whole tree. I'll also note that in my code the hashing of two nodes to make a new one is always done so that the node on the left enters the function as the first argument and the node on the right enters it as the second argument.

2.3 Hash chain extraction

Given a specific line of the log file and the log file itself, the full hash chain from leaf to root can be extracted. Of course, this can only be done, if the line given as an input actually does exist in the given log file. The hash chain consists of all the nodes on the path from the leaf to the root and of all the children nodes of these nodes. It is important that the user can take the chain and verify all of the hashing steps and results on the way from the leaf to the root. This is achieved by storing the chain as a sequence of pairs. The first value in a pair gives the position of the node in the hashing function for calculating its parent node. The second value is the node value itself. There is always an odd number of pairs in the chain. The pair on odd numbered positions in the sequence correspond to the node of the main path and the pairs on the even numbered positions correspond to the sibling nodes. Thus for calculating the value of the node on position $i + 2$ the nodes on positions i and $i + 1$ have to be hashed together in the order specified by the first values in the pairs.

For extracting such a hash chain, my code uses essentially the same algorithm as for calculating the root of the Merkle tree. The algorithm is slightly modified so that one could keep track of the values which should be added into the hash chain. This is done by always keeping track of the next value to be added to the chain. Let's call this value *target*. Its initial value is the hash of the log file line for which the chain is being extracted. As stated, the algorithm is essentially the same as the one described in section 2.2. But every time two nodes are hashed together to form their parent node, the code checks if one of these two nodes matches the *target*. If they don't, then nothing happens, but if one of them does, then both of these nodes are added to the hash chain, such that the node which matched the *target* is added first. The *target* is then set to equal the parent node of these two nodes, as it is clearly the next node added to the chain.

3 File structure and some implementation details

src/hasher.cpp

Implements the `main` function for the command line tools `hasher` and `test_hasher`. Both of them have essentially the same code, but for the hash function used: `hasher` uses SHA256 while `test_hasher` uses a trivial identity function. Preprocessor logic statements are used for switching the hash functions on the two tools. Usage of the trivial identity function as a hash function is motivated by the resulting readability of outputs. In combination with a very simple log file, it allows for manual checking of the output files correctness. Both of the tools have a simple command line user interface, the details of which can be gained from the help massage generated by the flags `-h` or `--help`.

The tools are capable of signing a text file via signing the root of the Merkle tree created from the file. The signature is written into a file `<log_file_name>.signature`. Secondly the tools are also capable of extracting hash chains for given lines. The lines, for which the hash chains are to be generated, are given to the tools via an extra input file. These chains are written into files `<log_file_name>.hash_chain_x`, where `x` indicates the line number from the extra input file. It is also possible to store the values of the Merkle tree leaves. If the corresponding flag is given, then the leaves are written into `<log_file_name>.leaves`.

include/merkelHasher.hpp

Implements a template class `MerkelHasher`. It carries the core of my code, having member functions for both the Merkle root calculation (`getRoot`) and the hash chain extraction (`getHashChain`). Both of the functions also allow for saving the leaves into a private field, which can be accessed via a getter. The class is templated on the hash function used as well as the "hash merging" function used. The "hash merging" function is used when two nodes are hashed together to form a parent node.

include/myHashInterface.hpp

Holds hash related code snippets:

(a) A trivial hash function `identity_hash`, which just returns the input.

(b) A wrapper around `OpenSSL` SHA256 hash function implementation, making it easier to use.

(c) The hash merging function used in my code (`myHashMerge`), which just adds (concatenates) the two input strings together.

(d) The `hash_chain_t` type definition.

include/mySignatureInterface.hpp

Implements a trivial empty signature function, which just returns the input. I have also tried to add the GUardtime SDK signing functionality into the comments. But I'm not sure I have included all the necessary files nor if I've called the right functions.

include/readcmd.hpp

Implements two helper functions for parsing command line arguments.

doc/

Directory with the files `doc.tex`, `doc.pdf` and `Guardtime.bib`.

example_logs/

Directory with two example log files. One is a real log file (`access_log_example`). The other one (`numbers.log`) has ten lines: numbers from 0 to 9. This can be used with `test_hasher` for checking the code.

run_test.sh

A `bash` script for running `test_hasher` with `numbers.log`. It stores the signature, leaves and the hash chains for all the lines. The script moves the output files into a separate directory and prints them to the command line.

Makefile

The `Makefile` for making the compilation easier.

4 Usage

4.0.1 Compilation

Before compiling one should refer to the `Makefile` and check that the compiler used (currently GCC) is compatible with their system. The code is dependent on `OpenSLL` library (https://www.openssl.org/). If its installation directory is not in the path, then one should change line 27:

```
OpenSSL = -lssl -lcrypto  →  OpenSSL = -lssl -lcrypto -L<path_to_OpenSSL>
```

For compilation just call `make` on the command line. The resulting executables will be placed into directory `built/`. At the moment the code has only been tested on Linux. It should work without issues on a Mac. I'm not sure about Windows machines.

4.0.2 Running

For both of the executables `build/hasher` and `build/test_hasher` the only compulsory argument is the log file name, which can be passed as `-i <log_file_name>`. The executables produce help messages in the expected way (`-h` or `--help`). If they are called with just the log file argument, then only root signing is conducted. For hash chain extraction use `--chain <file_with_requested_lines>` and for storing the leaves, just add `--leaves`.

For ease of testing the code output, I've include a script `run_test.sh`. It calls the `test_hasher` with `numbers.log`, storing the signature, leaves and the hash chains for all the lines. The script moves the output files into a separate directory and prints them to the command line.

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.