Technical Disclosure Commons

Defensive Publications Series

January 2021

Redactable Hashing for Time Verification

Jan Schejbal

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Schejbal, Jan, "Redactable Hashing for Time Verification", Technical Disclosure Commons, (January 24, 2021)
https://www.tdcommons.org/dpubs_series/3999

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

REDACTABLE HASHING FOR TIME VERIFICATION

Cryptographic hashes (outputs of cryptographically secure hash functions) are used to uniquely identify secret information without making such information public. A cryptographic hash function is a deterministic computation that produces the same outputs for the same inputs while ensuring that even a small change of the input (e.g., a change of only one or several bits) results in an unpredictable output that has little resemblance to the original (unmodified) input. Furthermore, it is not possible to efficiently find an input matching a certain hash value or two different inputs having the same hash value; an attacker that attempts to do so would need to test numerous values until one matches, which is computationally infeasible for a sufficiently large space of possible values. The quasi-randomness property of hashes makes them useful for mapping protected information while ensuring that the information is not easily discoverable by a malicious attacker or an unauthorized viewer. Hashes can be used to derive cryptographic keys from other (strong or even weak) cryptographic material. Hashes can be stored in lieu of passwords to verify identities of authenticating users, as well as in many other applications. Among such applications is the common practice of using hashes to prove the time a particular information was originated without revealing the information until some time in the future. For example, a security researcher who has discovered a vulnerability in a software or a database may want to establish the time when the vulnerability was found, without publicly disclosing its nature, to give the affected vendors and developers time to fix the problem. In such instances, the researcher can write up a description of the discovered problem, input the description into a hash function and publish the output hash H right away. The actual description can be published at some later time, after the vulnerability has been addressed. A verifier can input the digital file containing the description into the hash function and reproduce the hash value H. Because it is practically impossible to generate a different document matching a given hash (to find a second input value that reproduces the hash), such verification proves that the revealed document existed at the time the original hash H was published. Coupled with the fact that the original hash was published by a certain author at a certain time, this can help establish both the origin (authorship) of the description as well as the time when such description was mapped into the hash H. Such a procedure works as long the original description file has not been modified in the meantime.

In the process described above, any changes to the original description file, no matter how small, would result in a verification failure (hash mismatch). Yet changes often need to be made to redact out some sensitive information or any other information that the author of the description may wish to keep out of the public view. Sometimes, such information cannot be redacted during the time the hash H is generated because of a time pressure. In some instances, the changes cannot be foreseen until a later time.

The present disclosure addresses these shortcomings of the existing technology and describes methods of verifying the origin, authorship, and the time of creation of editable documents by splitting the documents into small redactable data units and using separate hashes created for each data units to produce the ultimate output hash H. The methods described herein allow for redacting out any number of the data units while still ensuring that the output hash H can be reproduced.

In one example, the document can be parsed into N data units D[1]…D[N]. A data unit is a minimum unit that can be independently redacted out. Depending on the implementation, a data unit can represent a paragraph, a line of text, a single symbol (a letter or a numeral), or a predetermined number of symbols (and/or spaces), such as 2, 5, 10 or any other number of symbols. For example, a data unit value can be an ASCII set (byte) that encodes letters, numbers, special characters, etc., or can be any other type of text encoding. In graphical documents, a data unit can represent a pixel or a group of pixels. The data unit value D can represent the brightness of the pixel/group of pixels in each of the colors that are used (e.g., B&W, RGB, CMYK, etc.).

For each data unit D, a blinding (masking) value can be generated. The blinding value B can be generated randomly and can be a large number having a significant entropy (the number of possible numbers of the same size), e.g., a 128-bit number, a 256-bit number, and so on. Each data unit value D[k] can be input together with its blinding value B[k] into a hash function Hash() to produce a hash unit value h[k]=Hash(B[k],D[k]). The function Hash() can be SHA-256, SHA-512, or any other secure cryptographic hash function. The function Hash() can be a hash-based message authentication code (HMAC). Other constructs may also be suitable, e.g. a hash-based key derivation function (HKDF), or any other key derivation function (KDF). Single- or multiple-stage hash functions can be used. The blinding value B[k] can be concatenated (appended) to the data unit value D[k] and the concatenated value used as a single input. Alternatively, B[k] and D[k] can be used as separate inputs, e.g., D[k] can be used as the 'salt' (or 'info') of HMAC/HKDF and B[k] can be used as the 'key' (or vice versa).

Once the hash values h[1]…h[N] are produced for all N data units, the hash values can be concatenated (e.g., appended in a sequential manner or combined in any other deterministic way) and input into another instance of a hash function (which can be the same function Hash() or a different function). The final hash *H*=Hash(h[1], .. h[N]) can be published at (or close to) the time of the document creation. At the time of verification, the author or the owner of the original document can upload for verification the following information: (1) the data units D[k] and the corresponding blinding values B[k] for each revealed (unredacted) data unit, and (2) the hash unit values h[r] for each data unit D[r] that has been redacted out. The verifier can then compute the data unit hashes h[k]=Hash(B[k],D[k]) for each data unit that is revealed, and input the data unit hashes h[k] together with the hashes h[r] of the redacted units into the final hash function to obtain the output hash *H* and compare it with the previously published value.

The advantage of using blinding values B[k] is that such use prevents the verifier (or any other person) from uncovering a redacted out unit D[r] from its known hash h[r], e.g. by trying out all possible values of D[k]. For example, a byte-sized unit D[k] can only have 256 different values, an easy target for a brute-force attack.

The above-described approach requires a significant amount of storage space to communicate such blinding values and unit hashes as may be needed for the final hash verification. For example, a Letter-sized document (8.5 by 11 inches) has more than 8.4 megapixels, at 300 dpi resolution. With each pixel represented by a 128-bit number (to store the respective blinding value or unit hash value), the total of 8.4 M x 16 bytes = 134 MB would need to be stored by the author (owner) and later transmitted for the verification.

The above process can be optimized by using a tree-based (e.g., using Merkle trees) hash derivation, illustrated in **FIGs 1** and **2**. As depicted in **FIG. 1**, blinding values can be derived from a single master value B, shown at the root level (the top) of the derivation tree, with the left branch branching off from each node indicated with bit ‘0’ and the right branch indicated with bit ‘1.’At the first level, two blinding values B[0] and B[1] are generated based on the master value B, e.g., using a KDF, HKDF, HMAC, or any other function applicable for derivation of a cryptographic key. In one example,

B[0]=KDF(metadata[0],B),
B[1]=KDF(metadata[1],B),

where ‘metadata[0]’ and ‘metadata[1]’ are some deterministic values (e.g., ‘info’ values into the KDF being used) identifying the node of the tree to which the blinding number being generated pertains, and used to ensure that the generated values B[0] and B[1] are distinct. Similarly, four blinding values are generated at the second level,

B[00]=KDF(metadata[00],B[0]),
B[01]=KDF(metadata[01],B[0]),
B[10]=KDF(metadata[10],B[1]),
B[11]=KDF(metadata[11],B[1]),

and so on, until the last (bottom) level is processed. The tree can have M levels (below the root level) in total. In order to provide blinding values for each of the N data units, the number of levels selected should be such that $M \geq \log_k N$, where k=2 for the tree shown in **FIG. 1** and k=4 for a quad tree.

Different kinds of trees can be used depending on the type of the document. For example, for images, a quad tree (a tree where each node has 4 child nodes instead of 2) can be practical. Note that the shape of the tree used for deriving the blinding values does not necessarily have to match the shape of the tree used to calculate the hash.

With the blinding levels at each of the M levels determined, the hashing process can be performed as shown in **FIG. 2**. Whereas generation of the blinding values is performed starting from the root level (as indicated schematically with the arrow in **FIG. 1**), the hashing process starts at the last (bottom) M-th level (the M=3 example is shown for brevity although much larger values M are likely to be used with large documents). The document is first split into data units D[k]. If the number of units N is smaller than 2^M, the “surplus” $2^M - N$ data units can be padding units (e.g., the units can be filled with empty/zero values, or the hashes of the units or subtrees can be replaced with e.g. empty/zero values). Each data D[k] unit can be input into a hash function together with the respective blinding value B[k] to obtain the unit hash value H[k]. For example, at the bottom (third, as in the example depicted) level,

H[000]=Hash(Metadata[000], D[000], B[000]),
H[001]=Hash(Metadata[001], D[001], B[001]),
…
H[111]=Hash(Metadata[111], D[111], B[111]),

where a deterministic value 'Metadata[k]' can be the same or different from the value 'metadata[k]' used in the blinding value generation operation of **FIG. 1**.

At the next (second, as depicted) level, the unit hash values are processed to obtain subsequent hash values. For example,

H[00]=Hash(Metadata[00], H[000], H[001]),
…
H[11]=Hash(Metadata[11], H[110], H[111]).

At the next (first) level, the process is continued, e.g.,

H[0]=Hash(Metadata[0], H[00], H[01]),
H[1]=Hash(Metadata[1], H[10], H[11]),

until the output hash value is generated,

H=Hash(Metadata[Output], H[0], H[1]),

The hash function Hash() can usually be a simple cryptographic hash (e.g., SHA-256 or SHA-512), although other constructs like HMAC or a KDF may also be suitable.

FIG. 3A demonstrates how the verification process can occur. In the example shown, the author (owner) of the original document may want to redact out the data units D[111], D[110], and D[011] while publishing the rest of the data units. The following rules describe one possible implementation of the verification process and identify the information that is to be revealed (a white rectangle in **FIG. 3A** indicates that the value inside it is revealed and a grey rectangle indicates that the value inside is not revealed):

1. If no data unit is revealed below a certain node [k] of the tree ("redacted" node), reveal the hash value for that node. For example, as no data units (neither D[110] nor D[111]) are revealed below the [11] node, only the hash value H[11] is revealed.
2. If all data units are revealed below a certain node of the tree ("public" node), reveal the blinding value for that node. For example, as all data units (both D[000] and D[001]) are revealed below the [00] node, the blinding value B[00] is revealed. Using the revealed blinding value, the verifier will be able to determine (via the procedure depicted in **FIG. 1**) all downstream blinding values needed to obtain the respective hashes. For example, the blinding values B[000] and B[001] necessary to generate hashes H[000] and H[001] can be generated using the revealed blinding value B[00].
3. If some data units below a certain node are revealed and some are redacted ("mixed" node), go one or more steps down the tree (e.g., from level K to level $K + 1$) until either scenario 1, or scenario 2 is encountered (in which case follow the above instructions), or a node of the $M - 1$ level (one level above the bottom level) with one revealed and one redacted data unit is encountered. For the revealed data unit, reveal the corresponding blinding value B[k]; for the redacted data unit, reveal the unit hash value H[k]. For example, as the node [0] of the first level is a mixed node, by going down the tree to the second level, we encounter the public node [00] (scenario 2) and the mixed node [01]. The left branch of the node [01] includes the revealed data unit D[010], and the right branch of the same node includes the redacted unit D[011]. Accordingly, the blinding value B[010] is revealed for the left branch and the hash unit value H[011] is revealed for the right branch.

In the hash-tree example of **FIG. 3A**, five blinding or hash values (B[00], B[10], B[010], H[11], and H[011]) need to be transmitted (and either stored or generated anew prior to transmission) to the verifier. For 128-bit hashing/blinding, this amounts to 5*128 bits = 80 bytes of information. On the other hand, in the linear approach all 8 blinding or hash values have to be transmitted, amounting to the total of 8*128 bits = 128 bytes of information. Accordingly, the 37.5% reduction in the storage space is achieved in this example. With an increasing number of data units, the reduction in storage and transmission will likely be much larger in practice (e.g. an unredacted, aligned square of 256x256 pixels could require only one blinding value instead of 65536 values in a quad-tree-based version of this algorithm optimized for images).

FIG. 3A illustrates the algorithm that minimizes the amount of data to be stored and transmitted to the verifier. In this algorithm, the need to know the blinding values of the bottom level (e.g., B[000], B[001]) is addressed by providing an intermediate blinding value (e.g., B[00]) that allows to generate the needed blinding values by shifting the task of generating such values to the verifier. In some instances, as illustrated in **FIG. 3B**, it can be advantageous to decrease the amount of computation performed by the verifier at the expense of storing and transmitting more data. Shown in **FIG. 3B** is an alternative implementation of the verification process where the explicit blinding values of the bottom level are provided to the verifier. The resources used in the implementations shown in **FIG. 3A** and **FIG. 3B** can be compared as follows. Assuming that there are M levels in the hash tree and that the public node P (e.g., node [00] with all downstream data units revealed) is at the level K. In the implementation shown in **FIG. 3A**, revealing the single blinding value associated with the node P (e.g., value B[00]) allows the verifier to compute all 2^{M-K} blinding values of the lowest level using $2 * (2^{M-K} - 1)$ key generating operations (e.g., instances of KDF). Instead, in the implementation shown in

FIG. 3B, the author needs to store (and the verifier needs to receive) $2 * (M - m)$ blinding values of the bottom level (e.g., blinding values B[000], B[001]). Depending on whether decreasing the storage volume or decreasing the verifier's computational load is prioritized, one or the other implementation can be preferred.

Although in some implementations, the data units D[k] may have equal size (length, pixel area), in some implementations, data units can have arbitrary sizes. For example, in an image, certain redacted areas can have unequal sizes. The text of an image of the document may often need to be redacted on a word-by-word basis or even letter-by-letter basis whereas logos, signatures, stamps may be redacted in larger portions, e.g., an entire picture of a logo may be redacted at once. In such implementations, the size of the data units can vary throughout the document. In some implementations, multiple documents (including dissimilar documents, e.g., a bitmap image, a vectorized graphics file, an ASCII text, etc.) can be combined using a single hash tree, as described above. An additional advantage that comes with the use of trees, is that the computations can be parallelized for processing speed.

The described methods can be used in any applications and/or situations where there is an advantage in delaying a public disclosure of sensitive information while securing a proof of discovery or conception. The information can be contained in a research paper, a memo, a legal document, a description of an invention, forecast, mathematical formula, flaw, vulnerability, image, video, entry in a database, cryptocurrency transaction, or any other document or entity whose existence at an earlier time may need to be securely confirmed at a later time.

When hashing of images is performed, a special care must be exercised to ensure that the pixel values being hashed stay the same when (1) the image originator (owner) generates the initial hash, (2) the originator produces the redacted image, (3) the verifier performs the verification process, and (4) the image is publicly displayed. Depending on the format used to store the image, such factors as variations in color profiles, a mismatch between compressed image versions being used, and the like, can cause problems. Specifically, discrepancies in operations (1)-(3) can cause the verification ending in a failure whereas discrepancies in operation (4) can change the representation of the image significantly enough so that a content is displayed which is different from the content for which the proof was performed. In some implementations, ensuring that no color profiles are used may likely be the safest option.

Additionally, it may be advantageous to ensure that redacted areas can be clearly distinguished from other document content when the result is displayed publicly. For example, if redactions are displayed in black, a malicious prover could potentially create a proof that looks as if it has additional text, e.g., by selecting a blank area and removing individual pixels so that letters (or other alphanumeric symbols) are formed. The likelihood of such an occurrence can be lowered (or eliminated) by the use of black-and-white images with colored redactions or by the use of animated (color fade, moving checkerboard, etc.) patterns indicating redacted areas. For the same reason, transparency may best be disallowed. Alternatively, transparency can be used to indicate redaction, if the original color in the RGBA image is carefully replaced to ensure that the RGB value of a transparent pixel does not reveal the original data.

ABSTRACT

Methods, based on hash trees, of proving a time of origination and a source of digital redactable documents is described. A document is represented via a number of redactable data units masked using a list or tree of blinding values and hashed using a list or tree of hashes, allowing for a selective subsequent revealing of the content of the document, any part of which can be redacted out. Depending on whether a data unit (or multiple data units corresponding to the same branch of the tree) is to be revealed or redacted, a respective blinding value or a respective hash value is provided. The described methods allow the verifier to confirm the time of origination and the source of the document by recovering the final hash notwithstanding any number of the data units being redacted. The described methods optimize the amount of data that has to be stored by the document originator and transmitted to the verifier for hash verification.

Keywords: hash, hash publishing, hash tree, Merkle tree, hash function, blinding, commitment, masking, key derivation function, redactable documents, security research, proof of vulnerability, proof of discovery

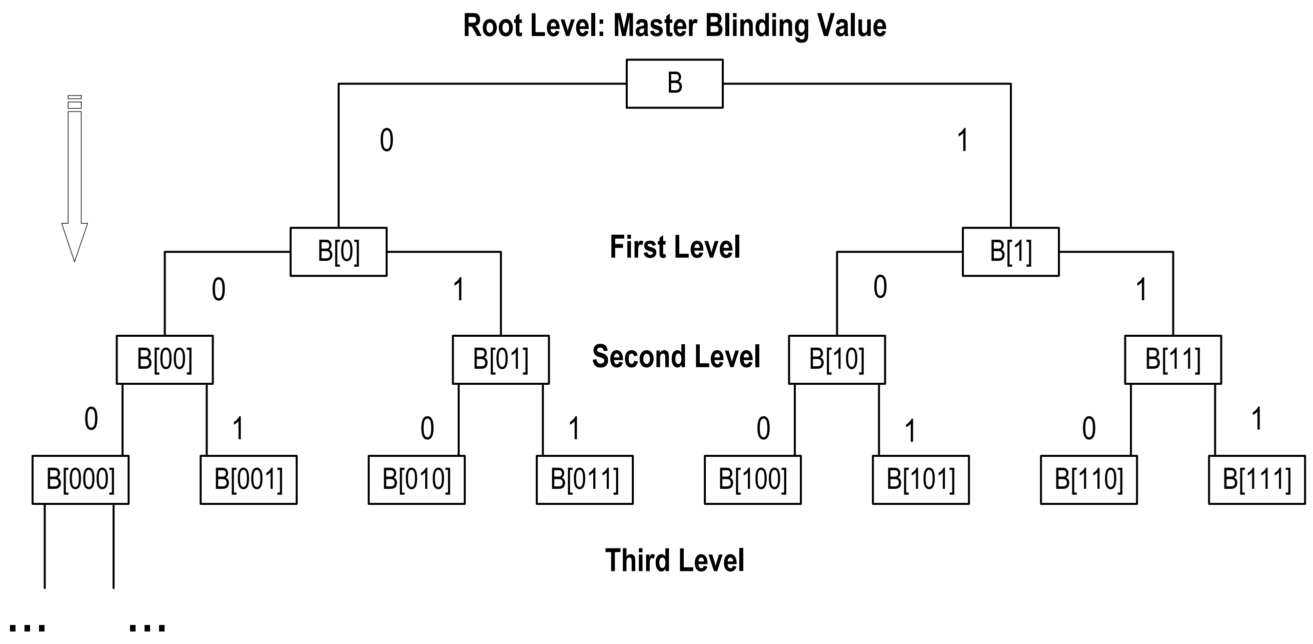
Root Level: Master Blinding Value
B
0
1
B[0]
First Level
B[1]
0
1
0
1
B[00]
B[01]
Second Level
B[10]
B[11]
0
1
0
1
0
1
0
1
B[000]
B[001]
B[010]
B[011]
B[100]
B[101]
B[110]
B[111]
Third Level
...
...

FIG. 1

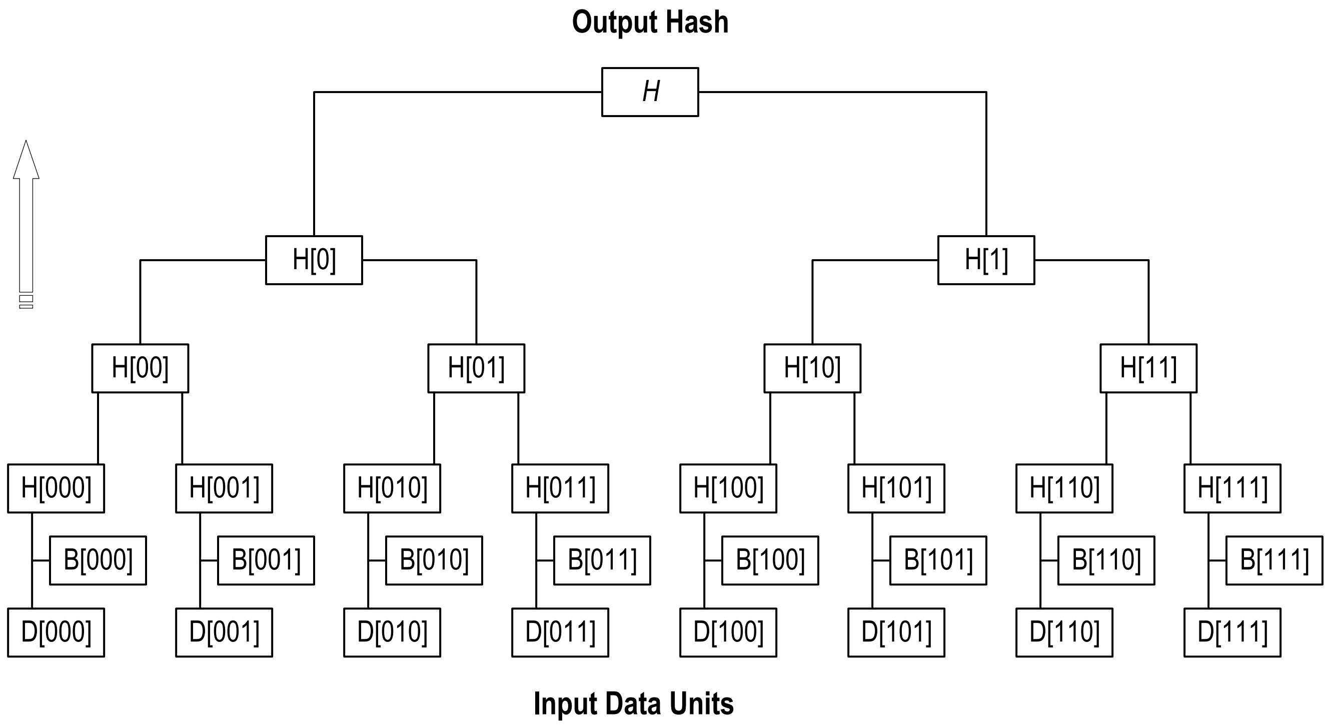
Output Hash
H
H[0]
H[1]
H[00]
H[01]
H[10]
H[11]
H[000]
H[001]
H[010]
H[011]
H[100]
H[101]
H[110]
H[111]
B[000]
B[001]
B[010]
B[011]
B[100]
B[101]
B[110]
B[111]
D[000]
D[001]
D[010]
D[011]
D[100]
D[101]
D[110]
D[111]
Input Data Units

FIG. 2

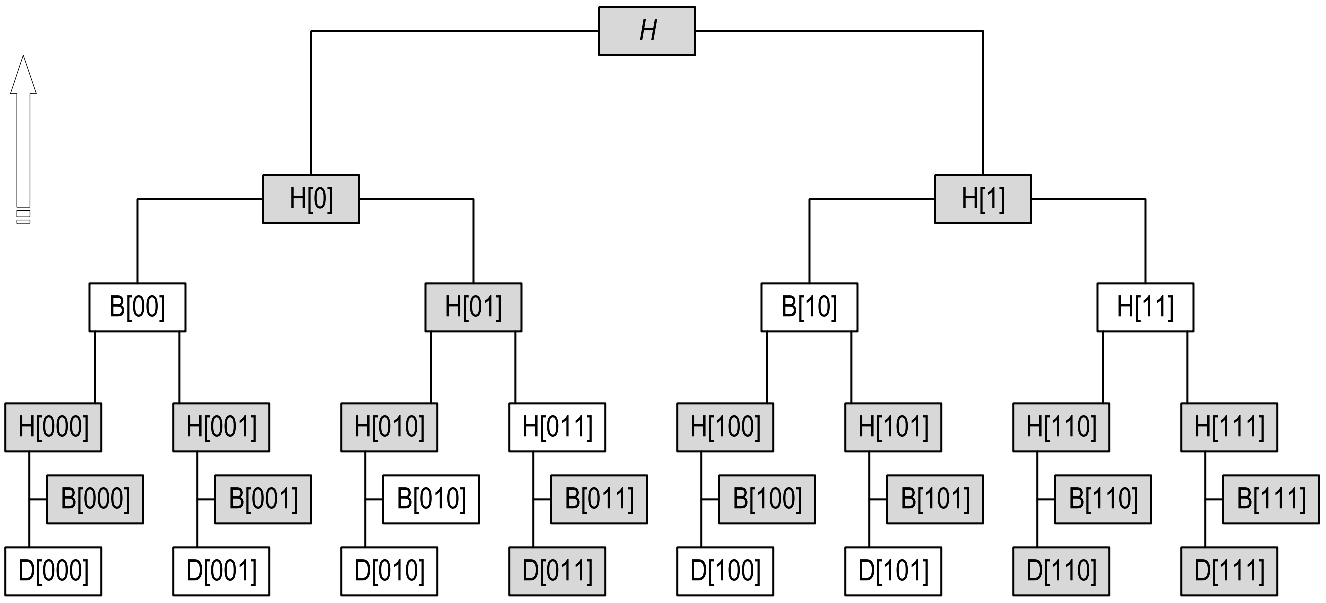
H
H[0]
H[1]
B[00]
H[01]
B[10]
H[11]
H[000]
H[001]
H[010]
H[011]
H[100]
H[101]
H[110]
H[111]
B[000]
B[001]
B[010]
B[011]
B[100]
B[101]
B[110]
B[111]
D[000]
D[001]
D[010]
D[011]
D[100]
D[101]
D[110]
D[111]

FIG. 3A

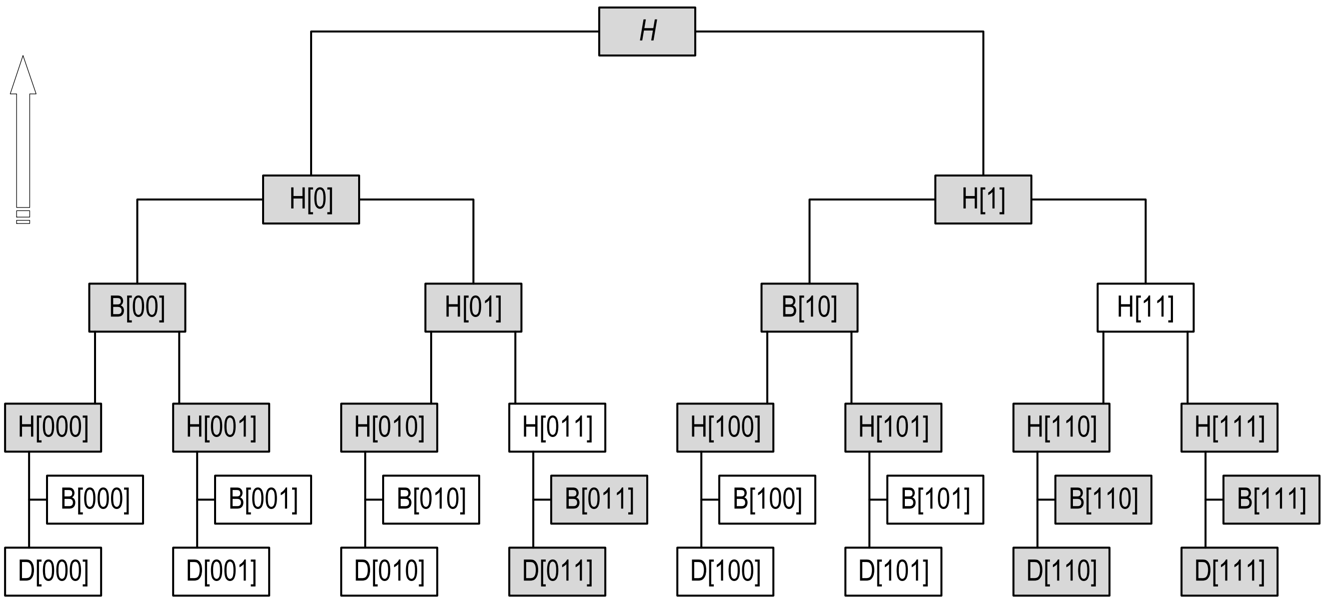
H
H[0]
H[1]
B[00]
H[01]
B[10]
H[11]
H[000]
H[001]
H[010]
H[011]
H[100]
H[101]
H[110]
H[111]
B[000]
B[001]
B[010]
B[011]
B[100]
B[101]
B[110]
B[111]
D[000]
D[001]
D[010]
D[011]
D[100]
D[101]
D[110]
D[111]

FIG. 3B