11-17-2020

Merkle Hash Grids Instead of Merkle Trees

Jean-Francois Paris

Thomas Schwarz

Merkle Hash Grids Instead of Merkle Trees

Jean-Francois Paris
University of Houston, Houston, TX
Thomas Schwarz
Marquette University, Milwaukee, WI

Abstract:

Merkle grids are a new data organization that replicates the functionality of Merkle trees while reducing their transmission and storage costs by up to 50 percent. All Merkle grids organize the objects whose conformity they monitor in a square array. They add row and column hashes to it such that (a) all row hashes contain the hash of the concatenation of the hashes of all the objects in their respective row and (b) all column hashes contain the hash of the concatenation of the hashes of all the objects in their respective column. In addition, a single signed master hash contains the hash of the concatenation of all row and column hashes. Extended Merkle grids add two auxiliary Merkle trees to speed up searches among both row hashes and column hashes. While both basic and extended Merkle grids perform authentication of all blocks better than Merkle trees, only extended Merkle grids can locate individual non-conforming objects or authenticate a single non-conforming object as fast as Merkle trees.

SECTION I. Introduction

Merkle trees, also known as hash trees, [10]–[11][12][13] are a fundamental building block in many sophisticated authentication schemes. In its basic functionality, a Merkle tree authenticates a set of blocks in a file that is transmitted through time in a storage system or through space in a messaging system. As Fig. 1 shows, a Merkle tree consists of as many leaves as there are blocks in the file we want to test and as many internal nodes as required by the degree of the tree. Each leaf contains the hash of one of the file blocks and each internal node contains the hash of the concatenated values of its children. As a result, any change in any file block will affect the value of the root of the tree. Thus signing that root suffices to authenticate the whole tree. Its strength relies on the strength of the hash function, i.e. the incapability to invert a hash with realistic resources and the strength of the signature.

Building a binary Merkle tree for a file that contains $N = 2^n$ blocks requires $N = 2^n$ hashes and one digital signature. In comparison, identifying a single non-conforming block is a relatively inexpensive operation as it only require $2\log_2 N = 2n$ comparisons.

When we want to verify the contents of a local copy of a file, we construct the Merkle tree of that copy and compare its root with the root of the Merkle tree of the remote replica. If they are identical, we know (with overwhelming probability) that the two copies are identical. If they are not, identifying each nonconforming block requires a traversal of the tree.

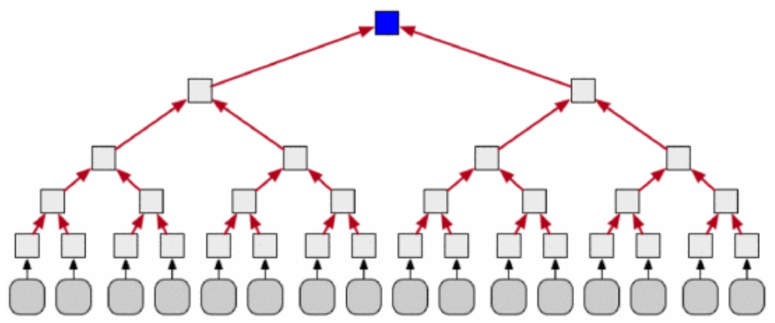
Fig. 1. A merkle tree. $N = 2^n$ blocks are arranged in a line. They are represented by the gray, rounded rectangles. We adorn each block with its hash (the light gray square). The arrows indicate taking the hash after concatenation. The root (dark blue) is a signed hash.

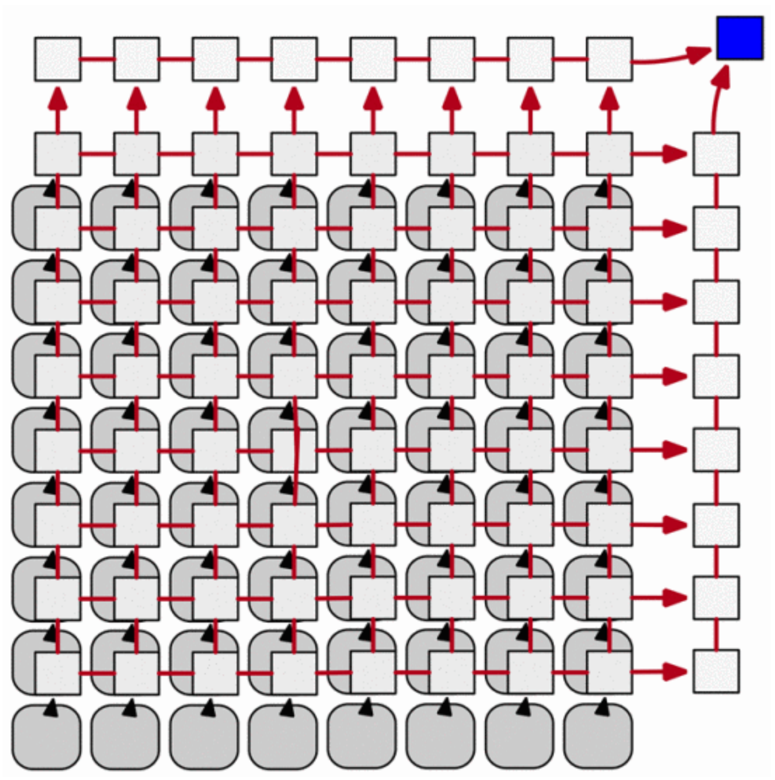
Fig. 2. The base version of our scheme. N = n2 objects are arranged in a square (represented by the gray, rounded rectangles). Light gray squares represent hashes. For each row and each column of hashes, we calculate the hash of the concatenation of the hashes in the line (the lighter gray squares on top and on the right). Finally, we calculate the hashes of each row and each column and sign it (the dark blue square in the upper right corner).

SECTION II. Merkle Grids

We introduce *Merkle grids*, a two-dimensional organization that has a smaller construction cost than the corresponding Merkle tree and only requires $2\sqrt{N}+1$ comparisons to identify all blocks that are likely to be non-conforming. As we will see later, we can even reduce the number of comparisons needed to locate individual non-conforming blocks to $2\log_2 \sqrt{N}+1$ by adding two auxiliary Merkle trees to our grid.

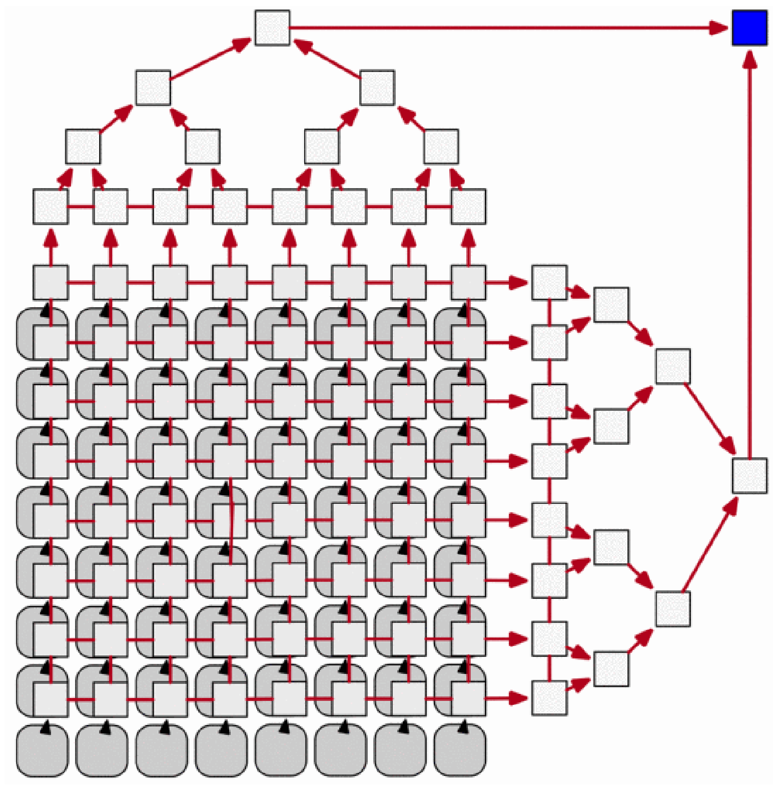
Fig. 3. The extended merkle grid with the two auxiliary merkle trees.

As Fig. 2 shows, Merkle grids organize the blocks of the file we want to verify in a square n*-by-*n array where $n = \sqrt{N}$ is the square root of the total number of blocks in the file. Rectangular arrays and arrays with incomplete rows or columns are possible but are somewhat less efficient. Each row of the array has a *row hash* containing a hash of the concatenated hashes of all blocks in that row. In the same way, each array column has a *column hash* containing a hash of the concatenated hashes of all blocks in that column. More formally, if b_{i_J} denotes the block at the intersection of the i-th row and the *j-th* column of the matrix, row hash r_i and column hash c_j are defined as

$$r_i = h(h(b_{i1}).h(b_{i2}).\dots.h(b_{in}))$$
$$c_j = h(h(b_{1j}).h(b_{2j}).\dots.h(b_{nj})),$$

where the dot. denotes concatenation. In addition, a *master hash* Ma contains a hash of the concatenated values of all row and column hashes. In other words.

$$M = h(h(r_1).\dots.h(r_n).h(c_1).\dots.h(c_n)).$$

Unlike the row and column hashes, this master hash is signed. We can thus see that building a Merkle grid requires $N + 2\sqrt{N} + 1$ hashes and one digital signature verification. This is significantly less than the number of hashes of a comparable Merkle tree and the space savings gets closer to 50 percent as the file size increases as

$$\lim_{N\to\infty} \frac{N + 2\sqrt{N} + 1}{2N - 1} = 0.5.$$

To verify the contents of a file, we first construct the Merkle grid of the local copy of the file and compare its master hash with the master hash of the remote copy. If they match, we know that the two copies are identical. Otherwise, we compare the row and column hashes of the local copy with the Merkel grid of the remote copy.

The operation will require 2 $\sqrt{N}$ additional comparisons and will detect all blocks that are likely to be nonconforming. The main advantage of our approach is that it detects all suspect blocks in a single sweep. Its main disadvantage lies in the additional diagnostic work it requires in the presence of several non-conforming blocks. If there are two non-conforming blocks, they are likely to be located in two different rows and columns. Since there are four potential nonconforming blocks in the intersections of the two rows and the two columns, we need to check these four blocks for nonconformity.

A. Extended Merkle Grids

While basic Merkle grids require up to 50 percent fewer hashes and occupy up to 50 percent less space than Merkle grids, their search performance is much less impressive as they require $2\sqrt{N}$ hash comparisons to locate a non-conforming block while Merkle trees only require $2\log_2 N$ comparisons.

A simple but effective way to reduce this number of hash comparisons is to add two auxiliary Merkle tree to the grid. As Fig. 3 shows, the first of the two trees is built upon the $\sqrt{N}$ row hashes while the second is built upon the $\sqrt{N}$ column hashes. As a result, both trees counts $2\sqrt{N}-1$ nodes and the Merkle grid now counts a total of $N+4\sqrt{N}-1$ hashes, that is N block hashes, $2\sqrt{N}-1$ hashes in each auxiliary Merkle tree, and a single signed hash of the concatenated roots of these two trees. Even so, the completed Merkle grid occupies up to 50 percent less space than a comparable Merkle tree as

$$\lim_{N\to\infty}\frac{N+4\sqrt{N}-1}{2N-1}=0.5.$$

The main advantage is that the number of comparisons required to identify a non-conforming block becomes $4\log_2 \sqrt{N}+1$, which simplifies into $2\log_2 N+1$, that is, one more comparison than with a Merkle tree.

SECTION III. Merkle Trees Revisited

A Merkle tree is a binary tree whose interior nodes are hashes and whose leaves are the blocks to be authenticated. The security of the Merkle tree depends directly on the security of the hashes, and in particular their collision resistance. The blocks could be messages, sensor data, blocks or objects in a file system, or any other type of large binary object. We give an example of a Merkle tree in Fig. 1. That Merkle tree has 16 objects (some of which could be null objects) represented by the rounded rectangles. The lighter gray squares represent hashes. The first level hashes are just the hashes of the objects. All other hashes are calculated recursively from the hashes of the two children of each node, namely the hash of the parent is the hash of the concatenation of the children. Formally, if p is the content of the parent and l and r are the contents of the children, the dot. denotes concatenation and $h\,()$ is our hash function, then

$$p=h(l.r).$$

The root of the tree is signed using a public signature. If a sender sends all objects, then the receiver would build the

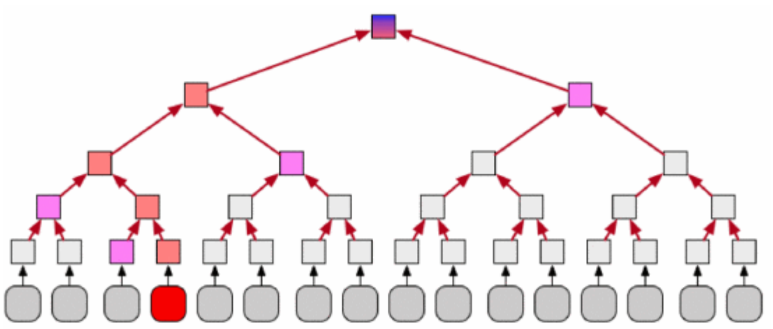
Fig. 4. Authentication of a single block in a merkle tree.

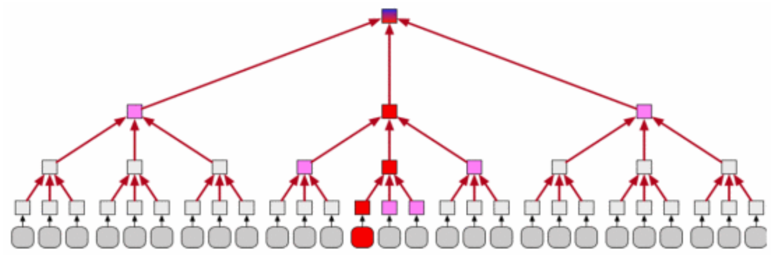
Fig. 5. A ternary merkle tree showing the authentication of a single object.

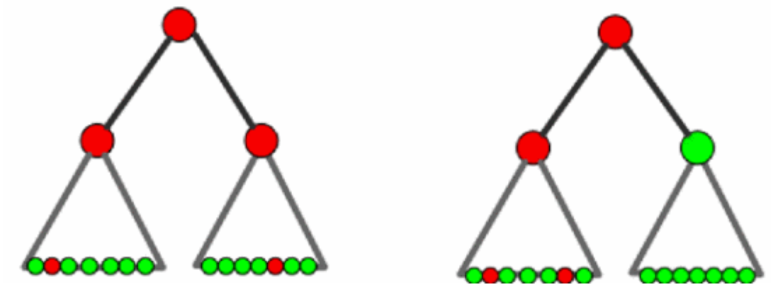
Fig. 6. The two cases for identifying two non-conforming blocks in a binary merkle tree.

Merkle tree from the received objects and then verify the authenticity of the root node contents using the signature. Of the operations involved, the creation and then the verification of a signature would be the most time-consuming steps.

In many circumstances such as in the sending of sensor data, all blocks are generated, sent, and authenticated. On some occasions though, only a single object needs to be verified. In this case, a Merkle tree can be authenticated by giving sibling hashes. Fig. 4 shows the case where a single block (marked in red) needs to be authenticated. For this to happen, we follow the path from the object to the root. We start by verifying the signature of the root hash. Then we verify the calculation of the root hash from its two constituents, namely the roots of the left and right subtree. Note that only one of these hashes lies on the path from the block to the root, the other one is its sibling. The hash of the root on the path (marked in light red in Fig. 4) is then verified recursively. At each level, we verify the hash on the path (red in Fig. 4) and its sibling (violet). With the exception of the root, we therefore access two hashes for each level in the Merkel tree. Constructing a Merkle tree for 2^n objects creates $2^n + 2^{n-1} + \cdots + 2 + 1 = 2^{n+1} - 1$ hashes which is very close to double.

A ternary Merkle tree, such as the one shown in Fig. 5, would be equally secure. Indeed, the number of hashes to be constructed for a tree with 3n blocks is

$$3^n + 3^{n-1} + \cdots + 3^2 + 3 + 1 = \frac{(3^{n+1} - 1)}{2}.$$

Compared to the binary version, the number of hashes per block has shrunk to approximately 3/2. In general, the number of hashes in a Merkle tree with fan-out k and height n, hence for kn blocks, is $(k^{n+1}-1)/(k-1)$. The number of hashes per block is this value divided by k^n, which has limit 1. The authentication of a single block uses now $kn+1$ hashes as at each level with the exception of the top level, we access k hashes.

Until now, we implicitly assumed that Merkle trees are just used for authenticating a file. In other words, the generic case to be optimized is that all blocks are true and have not been changed, be it by an attacker or by accident. We contend that for a large set of applications that use Merkle tree this is not the case. There are transmission errors and there are errors in storage devices that result in a device returning a different block than the one that was stored, for example, because of a write misdirect. These errors might be more frequent than attacks and failure of authentication because corruption then becomes the normal case. We do not deny that a malicious attack once detected has to be taken much more seriously. We deal with this scenario by talking about a sender, the entity that signed the hash, and the receiver, the entity that verifies the hash of the root node.

Assume that a single block out of the k^n blocks protected by a Merkle tree of height n and degree k is corrupted. Sender and receiver will then build different Merkle trees and the signed master hash sent by the sender does not agree with the locally constructed master hash of the receiver. Once the discrepancy between the sender's and the receiver's root hashes has been detected, the receiver asks for the k hashes at the level below the root. Since only a single block does not conform, only one of these k hashes differs between sender and receiver. The receiver now recursively proceeds by asking for the k child hashes of the non-conforming hash, and so on. The verification takes the same path as the authentication of a single block.

If two of the objects are corrupted, then we do not usually need to access twice the number of hashes. We first treat the case of a classic Merkle tree with fan-out 2 and height n, The exact number of accesses to hashes depends on luck, for example, if the first and the second block are corrupted, then we only need to access $2n$ hashes. We argue inductively. Let a_n denote the expected number of hashes and assume we have a Merkle tree of height n and that we know that the root hash does not match. In this case, we access the two child hashes. We know that at least one of the two child hashes is off, but we also need to consider the case where both are off. In the case illustrated at the left of Fig. 6, the two corrupted objects are the leaves of the left subtree rooted in the hash with the non-conforming hash. The probability of the two peccant block being located in two different halves is

$$p_{1,1} = \frac{(2^{n-1})^2}{\binom{2^n}{2}} = \frac{2^{n-1}}{2^n-1},$$

which is of course almost $1/_2$. In this case, we have read two hashes (the children of the root) and also need to find the two corrupted objects in a tree of height $n-1$. Thus, in total, we need $2+a_{n-1}$ accesses. Conversely, if we are in the case illustrated at the right of the figure, we need to find one nonconforming object in each of the two sub-trees of height $n-1$, so that together with the two child hashes, a total of $2+2.2\ (n-1)=4n-2$ hashes need to be accessed. If the height is one, then there are just two objects and we access two hashes. This gives us the recursion

$$a_n = 2 + \frac{2^{n-1}}{2^n-1}4(n-1) + \left(1 - \frac{2^{n-1}}{2^n-1}\right)a_{n-1}.$$

This recurrence is solved by

$$a_n = 2\left(\frac{1}{2^n-1}+2\right)n-4,$$

which can be approximated for large n by $4(n - 1)$. In the case of k-ary Merkle trees, the probability that the two nonconforming blocks are located in the same of the k sub-trees of the root is

$$^{(k^{n-1}-1)}/_{(k^n-1)}.$$

If the non-conforming blocks are located in two different subtrees, then we need to access $2k(n-1)$ hashes. If they are located in the same subtree, then we u recursion. This gives a recurrence $b_1 = k$ and

$$b_n = k + \frac{k^{n-1}-1}{k^n-1} b_{n-1} + \left(1 - \frac{k^{n-1}-1}{k^n-1}\right) k(n-1).$$

Its solution is

$$b_n = \frac{k(-k+n+4)-3}{k^n-1} + k\left(2n - \frac{k}{k-1}\right),$$

which is approximately $2k(n-1)$ for large n. Fig. 7 shows the similarity to a linear function very well.

We can extend this procedure to the case of three nonconforming objects, at least for binary Merkle trees. Thus, we assume the existence of 2^n objects but require $n \geq 2$ in order to have at least three objects. Again, we assume that we know that the root hash does not match. The probability that the three corrupt objects are in the same subtree with 2^{n-1} leaves is

$$p_{3,0} = \frac{\binom{2^{n-1}}{3}}{\binom{2^n}{3}} = \frac{1}{4} - \frac{3}{4(2^n-1)},$$

which is of course zero for $n = 2$, but then very quickly converges to ¼. The only other case is that of the three objects, two are in one sub-tree and the remaining one in the other, which happens with probability

$$p_{2,1} = 2\frac{2^{n-1}\binom{2^{n-1}}{2}}{\binom{2^{n-1}}{3}} = \frac{3 \cdot 2^{n-2}}{2^n - 1}$$

which starts out at 1 for $n = 2$ and then quickly sinks to ¾.

If we denote by τ_n the number of hashes accessed in the case of three corrupt objects in a Merkle tree of 2^n objects, we can now first develop a recurrence equation and then obtain its solution. Again, we assume that τ_n does not include the root hash, which is known to be non-conformant. We then access the roots of the two subtrees. According to the probabilities just calculated, either three of the non-conforming objects are in the same subtree or they are divided $2+1$ among the sub-trees. In the latter case, we have already determined the expected number of the hashes, in the former case, we proceed by recursion. This gives us

$$\tau_n = 2 + p_{3,0}\tau_{n-1} + p_{2,1}\big(a_{n-1} + 2(n-2)\big).$$

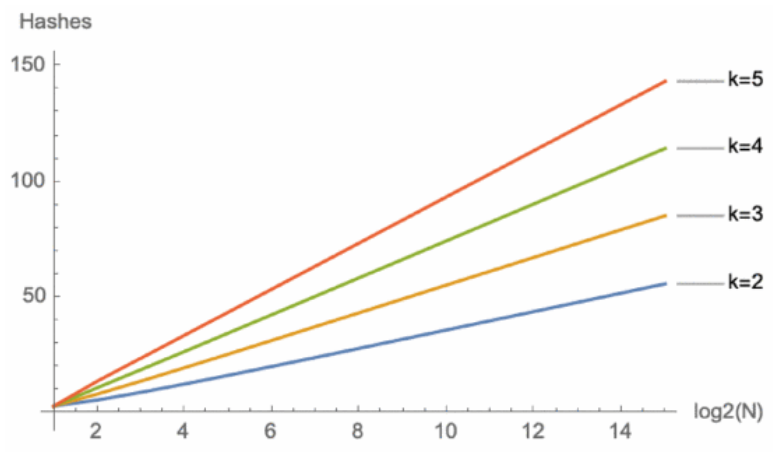

Fig. 7 The expected number of hash accesses in a merkle tree with N blocks if two blocks are non-conforming.

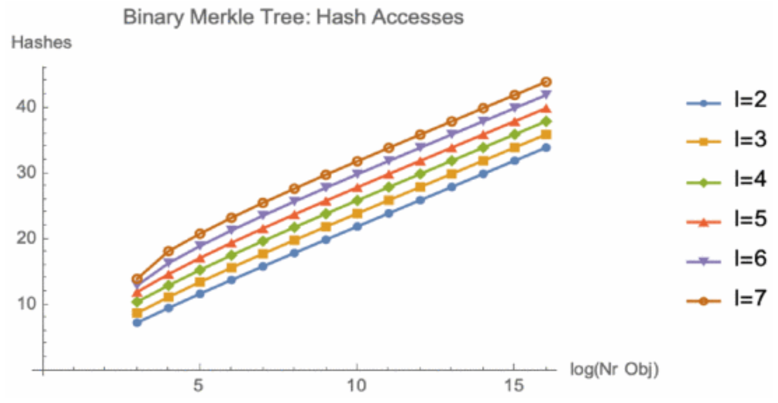

Fig. 8. Number of hashes accessed for a burst error of length l and a total of 2^n blocks in a binary merkle tree. The x-axis has n and the y-axis the number of hashes.

This complicated recurrence relation has a surprisingly simple solution, namely, τ_n equals

$$\frac{2(9\cdot 2^{2n}n - 9\cdot 2^{n+1}n + 6n + 9\cdot 2^{n+2} - 7\cdot 2^{2n+1} - 22)}{3(2^n-2)(2^n-1)}.$$

In an overabundance of caution, we used simulation to verify the result.

Finally, we treat the behavior of binary Merkle trees with 2^n leaves in case of a burst error of size l. While it is possible to give a recurrence relation for the number of hashes, this approach does not provide additional insight over a reasonable set of examples. We wrote a Python script that counts exactly the number of hashes accessed for various values of n and l and their position and then calculates the exact expectation under the assumption that al 1 $2^n - l$ possible positions of the burst are equally likely. While this is true only in certain scenarios, the results remain typical. The results are given in Fig. 8, which shows that the behavior is essentially linear. For small values of n and large values of l. the number of hashes approaches the total number of hashes. This explains why the curves overlap at their onset.

SECTION IV. Performance Evaluation

In this section, we evaluate the number of hashes required to identify non-conforming blocks in both basic and extended Merkle grids and compare these values with those obtained in the previous section for Merkle trees. Strictly speaking, we should make a distinction between taking the hash of objects and the hash of hashes as the latter are considerably smaller.

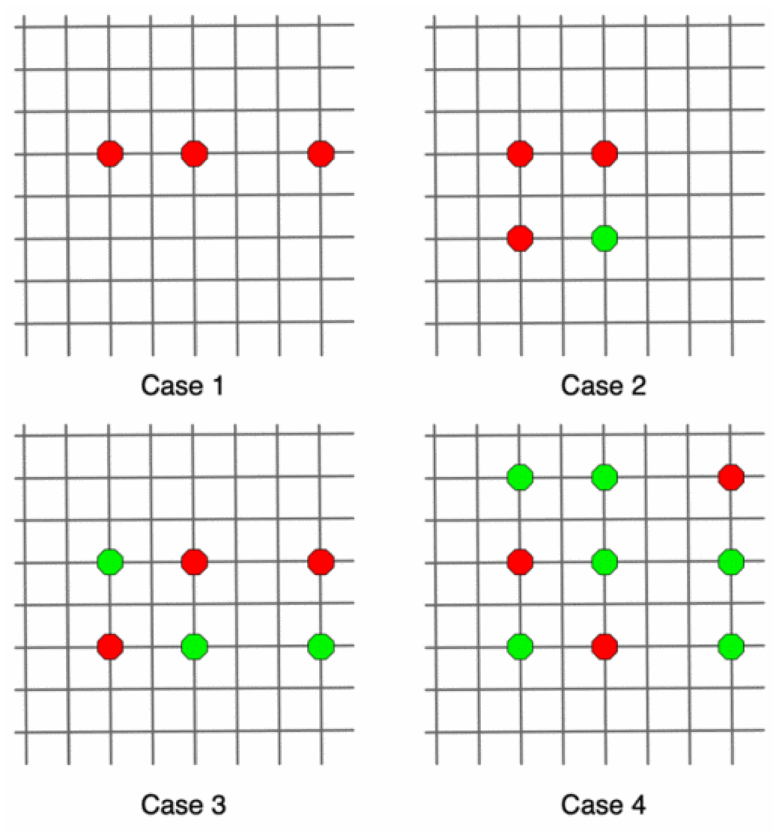

Fig. 9. Case distinctions for three non-compliant objects in a merkle grid.

A. Basic Merkle Grids

Let us consider first the case when we have a single nonconforming block to locate. In that case, the faulty block can be uniquely identified by its row and its column indices and the total number of hashes we need to compute is the total number of row and column hashes, that is, $2\sqrt{N}$ or $2n$.

- If the file contains two non-conforming blocks, we still need to compute the hashes of the $2n$ line hashes but they are not necessarily enough to identify the two faulty blocks. We have two subcases to consider:
- **Case 1**: The two non-conforming blocks share either a common row or a common column. In that case, we can uniquely identify the two blocks by their row and column indices.
- **Case 2**: The two non-conforming blocks do not share a common row nor a common column. In that case, the row and column indices identify four potential non-conforming blocks without telling us how many of these blocks are faulty. Identifying the two non-conforming blocks will thus require
- computing four extra hashes. There are $\binom{n^2}{2}$ distinct ways the two non-conforming blocks can be located on the grid and
- $2\binom{n}{3}^2$ distinct ways these blocks would not share a common row nor a common column.

Assuming that the two non-conforming blocks are uniformly distributed over the square array, the probability of having to compute four additional hashes is

$$\frac{2\binom{n}{2}}{\binom{n^2}{2}} = \frac{n^2(n-1)^2}{n^2(n^2-1)} = \frac{n-1}{n+1} = 1 - \frac{2}{n+1}$$

and the average number of hashes required to identify two nonconforming blocks is

$$2n + 4(1 - \frac{2}{n+1}) \approx 2n + 4 = 2\sqrt{N} + 4$$

If there are three non-conforming blocks, we have to take into consideration the four distinct failure patterns displayed in Fig. 9. We can distinguish them by noting how many rows and columns contain non-conforming blocks. The possibilities are Case 1: 1 × 3 and 3 × 1, Case 2: 2 × 2, Case 3: 2 × 3 and 3 × 2, and Case 4: 3 × 3.

- **Case 1**: There are twice $n\binom{n}{3}$ possibilities to select a single row and three different columns or a single column and three different rows. This selection not only determines the nonconforming blocks, but also their number. No additional hash computations are required as we can infer the identities of the three non-conforming blocks from their row and column indices.
- **Case 2**: There are $\binom{n}{3}^2$ possibilities to select two rows and two columns in the grid. There are then four possibilities to select the three non-conforming blocks at the four intersection
- points of these rows and columns for a total of $4\binom{n}{2}^2$ possibilities. We also need to ascertain the four object hashes to diagnose as we do not know *a priori* that we only have three non-conforming blocks.
- **Case 3**: There are $\binom{n}{2}\binom{n}{3}$ ways to select two rows and three columns for the non-conforming blocks and the same number to select three rows and two columns. In both cases, there are then six possibilities to select three non-conforming blocks such that each row / column selected has at least one non-conforming
- block. The total number of patterns is therefore $2 \cdot 6 \cdot \binom{n}{2}\binom{n}{3}$. All object hashes at the six intersection points need to be compared.
- **Case 4**: There are $\binom{n}{3}\binom{n}{3}$ ways to select three rows and three columns for the non-conforming blocks. There are 6 possibilities to place the three non-conforming blocks in the nine intersections such that all non-conforming blocks are in three different rows and three different columns. In this case, we need nine additional hashes in order to ascertain number and locations of non-conforming objects as the algorithm has no *a priori* knowledge of the number of non-conforming objects.

Taking everything together, the expected number of additional hashes is

$$\frac{8 + 9(n-2)n}{n^2 - 2},$$

which converges slowly to 9. The number of total hashes is

$$2n + \frac{8 + 9(n-2)n}{n^2 - 2}.$$

As Fig. 10 shows, the number of expected hashes for a Merkle Grid is better than that of an equivalent Merkle tree for small sizes, but since the former grow linearly with the number of objects, the logarithmic growth for Merkle trees soon wins out.

We also investigated the behavior of basic Merkle grids with 2^n leaves in case of a burst error of size l using the same technique as we did for Merkle trees. As Fig. 11 shows, the number of hashes to be accessed for simple Merkle grids in the presence of burst errors is again dominated by the need to access all row and column hashes. This eliminates the potential of having to access more than the needed block hashes if the burst stretches over two rows. As can be seen from the Figure, the curves lie virtually on top of each other for all bursts regardless of their length. This is shown more clearly in Fig. 12, which focuses on small grids.

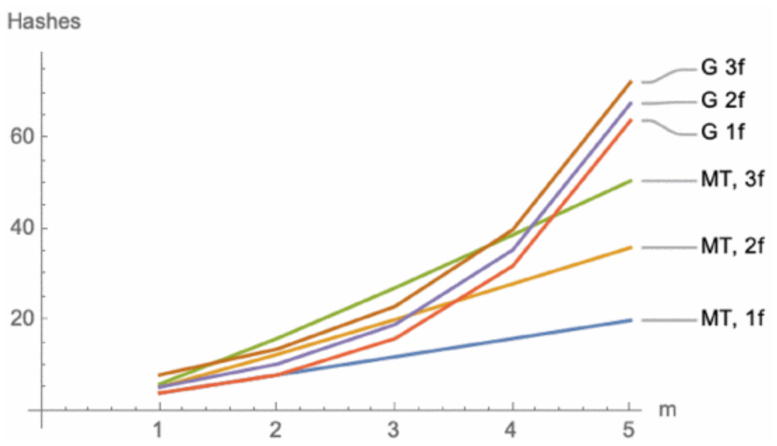

Fig. 10. Expected number of hash accesses needed to identify one, two or three non-conforming objects in a merkle tree or a basic merkle grid with 2^{2m} objects.

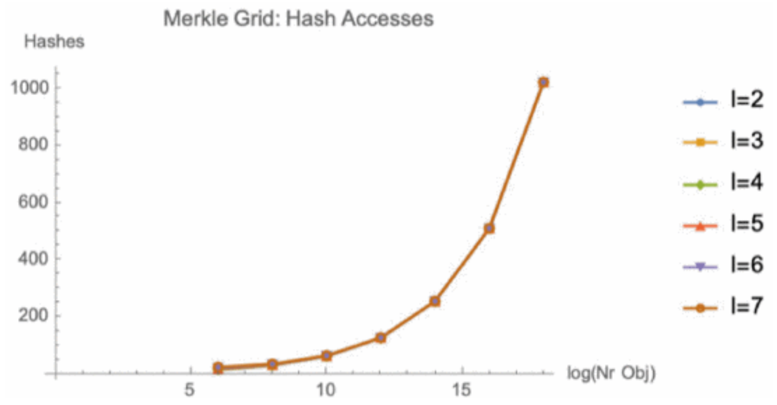

Fig. 11. Number of hashes to be accessed for simple merkle grids in the presence of burst errors of length l.

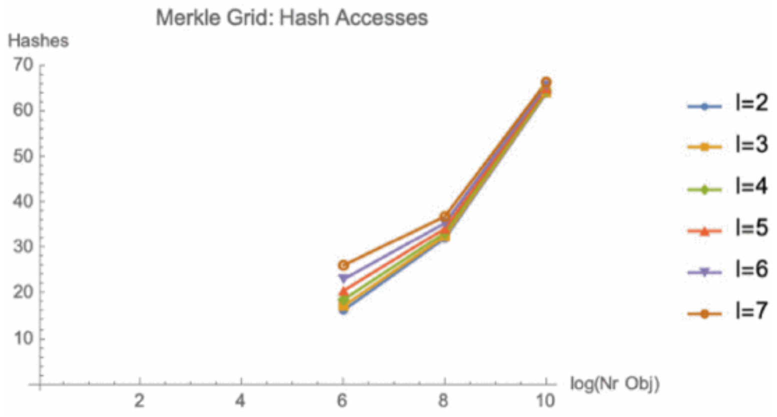

Fig. 12. Number of hashes to be accessed for simple merkle grids counting 64, 256 or 1,024 objects in the presence of burst errors of length l.,

B. Extended Merkle Grids

As we have seen, Merkle trees can authenticate a single block among N blocks by providing $\log_2(N)$ hashes and potentially verifying one signature. Merkle grids need the sender to send the $\sqrt{N}$ hashes of all column hashes and the $\sqrt{N}$ hashes of the blocks in the same column as the one to be authenticated,

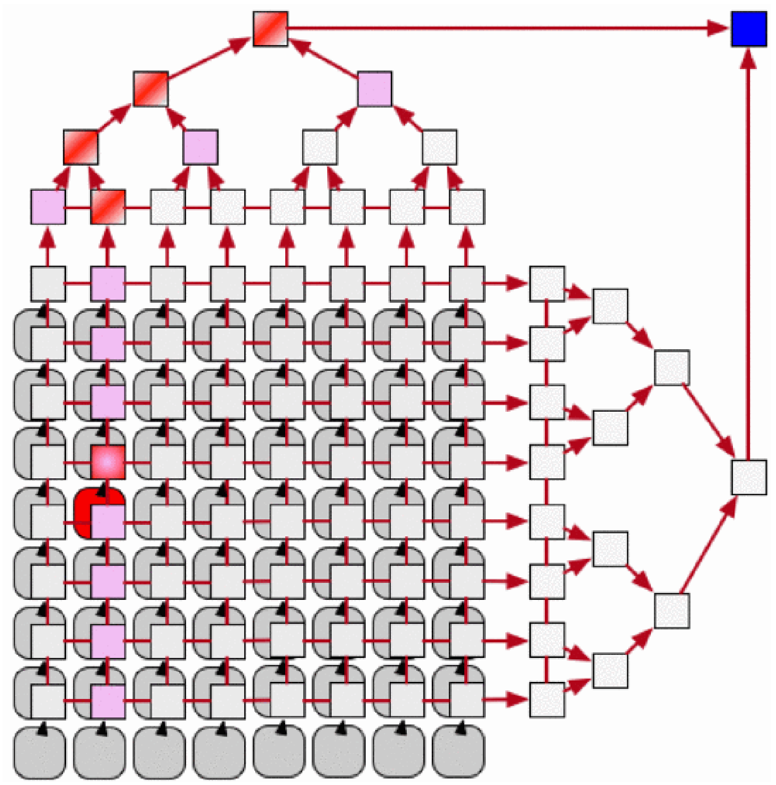
Fig. 13. Authenticating a single block in the extended merkle grid

We depict the situation for an extended Merkle grid in Fig. 13. We want to verify the block in red. Both sides calculate its hash. We then send the hashes of the blocks in the same column to the receiver. Alternatively, we could have chosen the row instead of the column. The receiver then calculates the column hash (indicated by the red-white double striping of the square representing the hash) from the hash of the block at its side and the witness hashes just sent. The sender sends a companion hash, i.e. the column hash of the column next to the column with the block to be authenticated, of the lowest leaf layer of the column Merkle tree (indicated in violet) as a witness. The receiver now calculates the hash of the concatenation of both and starts this way working up the tree. At each step, there is a just calculated hash and a companion hash sent by the sender. When the top of the Merkel tree is reached, the hash is compared to the corresponding part of the signed concatenation of two root hashes. This finishes authentication. We have n object hashes and log2(n) hashes of hashes to calculate. We also need $\log_2(n)$ witness hashes to be sent possibly in a single message from sender to receiver.

We now calculate the cost of an Extended Merkle Grid with $2^m \times 2^m = 2^{2m}$ objects. If only a single block is nonconforming, then the row and column Merkle trees will need twice $2m$ additional hashes to isolate the non-conforming row and column hashes, which suffice to uniquely identify the nonconforming block uniquely. This gives a total of $4m$ additional hashes. Recall that we always assume that the root hash is shown to be not compliant.

For two non-compliant blocks, we use the same four case distinctions as before. In Case 1, the two non-conforming objects share a row or a column, and no additional hashes from the grid are required. The non-conformance requires accessing $4m$ hashes in one Merkle tree and

$$a_m = 2\left(\frac{1}{2^m - 1} + 2\right)m - 4$$

hashes in the other. In Case 2, four hashes in the grid are needed as well as 2 a_n in the two trees. In total, we have

$$-4 + 4\left(2 + \frac{1}{-1 + 4^m}\right)m$$

accesses by taking the expected number.

For three non-compliant blocks, we again use the same case distinctions as before, see Fig. 9. In Case 1 (3 × 1 or1 × 3), no additional hashes in the grid itself are needed, and $2m$ hashes in one tree and τ_m in the other tree. In Case 2 (2 × 2), four hashes from the grid itself and $2a_m$ in the trees are needed. In Case 3 (2 × 3 or3 × 2),6 additional hashes in the grid itself and $a_m + \tau_m$ in the trees are required. Finally, in Case 4 (3 × 3), we need $2\tau_m$ accesses in the trees and nine in the grid itself. We already calculated the probabilities for these cases before assuming that non-conformity is equally likely for all objects. The resulting expectation simplifies into

$$\frac{-36 + 9 \cdot 2^{1+3m} + 4^m(47 - 72m) + 24m + 16^m(-29 + 36m)}{3(2 - 3 \cdot 4^m + 16^m)}.$$

This value is virtually indistinguishable from that for the binary Merkle tree.

Finally, we consider the effects of a burst error affecting l consecutive objects. In the basic Merkle grid, this cost is dominated by accessing all line hashes, whereas in the Merkle tree, it is almost as efficient as finding a single non-conforming object, see Fig. 9. An extended Merkle grid has almost the same performance as a Merkle tree. We obtained our results, given in Fig. 14, through a program that enumerated all possible locations of l bursts and collected the mean number of hashes accessed. The numbers are slightly worse than for the Merkle tree.

C. Our Findings

Looking at Fig. 14, we can see that the performance of the Merkle tree and the extended Merkle scheme are almost identical. For the basic scheme, we have better or almost identical performance for small sizes. Since we can assume that the probability of encountering non-conforming blocks is very low, that fact is of little practical interest.

An important consideration is the cost of authenticating all blocks. To perform this task, a Merkle tree must verify all its elements, which requires $2^{N+1} - 1$ steps. A basic Merkle grid would just have to authenticate its master hash and compute $2\sqrt{N}$ hashes. An extended Merkle grid would just have to verify its master hash and its two auxiliary trees, which would require $4\sqrt{N} - 1$ hashes. The savings can be quite considerable for large files or large collections of messages. Consider for instance a file consisting of 1,024 blocks. A Merkle tree authentication would require 2,047 hashes while a basic Merkle grid and an extended Merkle grid would respectively require 65 and 128 hashes.

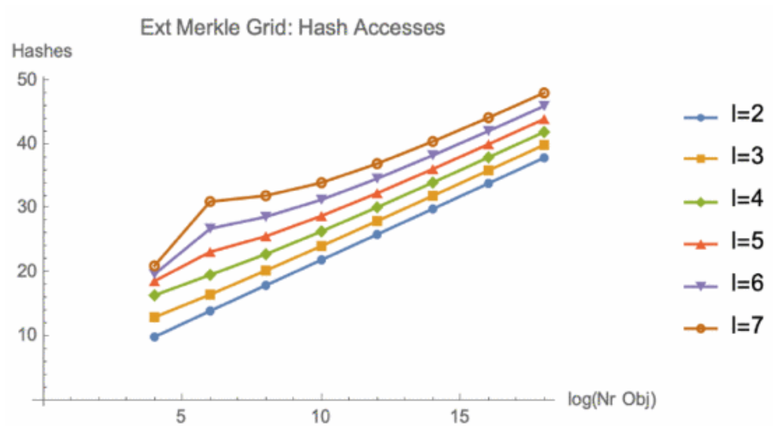

Fig. 14. Number of hashes accessed for a burst error of length l and a total of $2^n \times 2^n$ objects in an extended merkle grid. The x-axis has 2^n, that is the logarithm of the number of objects, and the y-axis the number of hashes.

Table I. Comparison of schemes for 22m objects

Costs	Binary Tree	Basic Grid	Extended Grid
No. of hashes	$2^{2m+1} - 1$	$2^{2m} + 2^{m+1} + 1$	$2^{2m} + 2^{m+2} + 1$
Authenticating all objects	$2^{2m+1} - 1$	$2^{m+1} - 1$	$2^{m+2} - 1$
Locating one non- conforming object	$4m$	2^{m+1}	$4m$
Locating two non- conforming objects	$\approx 8m$	$2^{m+1} + 4$	$\approx 8m$

Our findings are summarized in Table I. They highlight the excellent performance of extended Merkle grids that take slightly more space than basic Merkle grids-and much less space than Merkle trees, but still perform as well or better than Merkle trees in all the cases we investigated.

SECTION V. Related Work

Originally used for a public key signature scheme [2] [10]–[11][12][13], Merkle trees have been used in a wide array of settings. We only present here some of them. He, Xu, Fu, and Zhou [4] as well as Lin and Sung [8] use Merkle trees for authenticating the sender of multicast messages. In multicasting, the sender will not automatically retransmit lost packets and the communication between receiver and sender must be minimized. At the same time, the cost of authenticating messages should be amortized over a large number of messages. Both sets of authors propose to group the messages into the leaves of a Merkle tree and judiciously append partial matches to spread calculations over the processing of each message. As is usual, only the root hash is signed by the sender and authenticated by receivers. F errag and his colleagues [7] mention various applications of Merkle trees for authentication protocols for the Internet of Things.

File systems use checkpointing extensively [15] and often use checksums [5] to implement them. For instance, Open Solaris's ZFS file system uses checksums for detecting data corruption. ZFS organizes all on-disk data and metadata into objects that are further grouped into object sets. Checksums for on-disk blocks are kept separate by storing them into a parent block. They form part of a Merkle tree. In this manner, ZFS can detect all types of silent data corruption [16].

Another problem area where Merkle hash trees are used is the authentication of outsourced data, especially that of a database. Li and colleagues propose to embed Merkle trees in the nodes of a B+ tree [6] Niaz and

Saake [14] propose to replace the binary structure of Merkle trees with a B+ tree. In this context, Martel and colleagues [9] propose to replace binary Merkle trees with generalized directed acyclic graphs (DAGs) in the context of authenticating general data structures. The owner of a data structure provides first a deterministic search procedure that takes a query, searches the data structure, and returns the correct answer. The procedure is modeled as accessing nodes in a DAG. Each leaf node contains the hash of the data found there, and all interior nodes contain a hash of a concatenation of the hashes of their child nodes.

Auvolat and Traïani [1] have introduced Merkle Search Trees and proposed to use them to build causally consistent event stores that can provide causally consistent eventual delivery of updates to all connected nodes of a distributed system. Dahlberg, Pulls, and Peeters [3] investigated sparse Merkle trees and presented the first complete, succinct, and recursive definitions of sparse Merkle trees and related operations.

SECTION VI. Conclusion

We have presented a new data organization that replicates the functionality of Merkle trees while reducing their transmission and storage costs by up to 50 percent. All Merkle grids organize the objects whose conformity they monitor in a square array. They supplement this array with:

1. Row hashes that contain the hash of the concatenation of the hashes of all the objects in their respective row
2. Column hashes that contain the hash of the concatenation of the hashes of all the objects in their respective column.
3. A single signed master block that contains the hash of the concatenation of all row and column hashes.

Extended Merkle grids add to this basic scheme two auxiliary Merkle trees to speed up searches among both row hashes and column hashes.

While both basic and extended Merkle grids perform authentication of all blocks better than Merkle trees, only extended Merkle grids can locate individual non-conforming objects or authenticate a single non-conforming object as fast as Merkle trees. In addition, both basic and extended Merkle grid can authenticate N objects much faster than Merkle trees, as the number of steps they take is proportional to $\sqrt{N}$ instead of being proportional to N.

References

1. A. Auvolat and F. Taiani, "Merkle search trees: efficient state-based CRDTs in open networks", *Proc. 38th IEEE International Symposium on Reliable Distributed Systems (SRDS 2019)*, pp. 1-10, Oct 2019.
2. G. Becker, "Merkle signature schemes Merkle trees and their cryptanalysis", *Tech. Rep*, 2008.
3. R. Dahlberg, T. Pulls and R. Peeters Roel, "Efficient sparse Merkle trees: caching strategies and secure (non-)membership proofs", *Secure IT Systems: Proc. 21st Nordic Conference NordSec 2016 Springer LNCS #10014*, pp. 199-215.
4. J.-X. He, G.-C. Xu, X.-D. Fu and Z.-G. Zhou, "A hybrid and efficient scheme of multicast source authentication", *Proc. 8th ACIS International Conference on Software Engineering Artificial Intelligence Networking and Parallel/Distributed Computing (SNPD 2007)*, vol. 2, 2007.
5. A. Krioukov, L. Bairavasundaram, G. Goodson, K. Srinivasan, R. Thelen, A. Arpaci-Dusseau, et al., "Parity lost and parity regained", *Proc. 6 th USENIX Conference on File and Storage Technologies (FAST 2008)* , pp. 127-142, Feb. 2008.

6. F. Li, M. Hadjieleftheriou, G. Kollios and L. Reyzin, "Dynamic authenticated index structures for outsourced databases", *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 121-132, June 2006.
7. M. Ferrag, L. Maglaras, H. Janicke, J. Jiang and L. Shu, "Authentication protocols for internet of things: a comprehensive survey" in Security and Communication Networks, Hindawi, 2017.
8. I.-C. Lin and C.-C. Sung, "An efficient source authentication for multicast based on Merkle hash tree", *Proc. 6 th International Conference on Intelligent Information Hiding ond Multimedia Signal Processing (IIH-MSP 2010)* , Oct. 2010.
9. C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong and S. G. Stubblebine, "A general model for authenticated data structures", *Algorithmica*, vol. 39, no. 1, pp. 21-41, 2004.
10. R. C. Merkle, "Secrecy authentication and public key system", *Ph. D. thesis. Stanford University 1979 also Technical Report No. 1979-1*.
11. R. C. Merkle, "Protocols for public key cryptosystems", *Proc. 1et IEEE Symposium on Security and Privacy*, pp. 122-134, Apr. 1980.
12. *Method of providing digital signatures*, 1982.
13. R. C. Merkle, "A digital signature based on a conventional encryption function" in Advances in Cryptology - CRYPTO '87. Lecture Notes in Computer Science, vol. 293, pp. 369-378, 1987.
14. M. Niaz and G. Saake, "Merkle hash tree based techniques for data integrity of outsourced data", *Proc. 27th GI-workshop on Foundations of databases (Grundlagen von Datenbanken)*, 2015.
15. C. Stein, J. Howard and M. Seltzer, "Unifying File System Protection", *Proc. 2001 USENIX Annual Technical Conference*, pp. 79-90, June 2001.
16. Y. Zhang, A. Rajimwale, A. Arpaci-Dusseau and R. Arpaci-Dusseau, "End-to-end data integrity for file systems: a ZFS case study", *Proc. 8 th USENIX Conference on File and Storage Technologies (FAST 2010)* , Feb. 2010.