Using Homomorphic hashes in coded blockchains

Doriane Perard, Xavier Goffin, Jérôme Lacan
ISAE-Supaero, Université de Toulouse, France
firstname.name@isae-supaero.fr

***Abstract*—One of the scalability issues of blockchains is the increase of their sizes which can prevent users from storing them and thus from contributing to the decentralization effort. Recent works developed the concept of coded blockchains, which allow users to store only some coded fragments of the blockchains. However, this solution is not protected against malicious nodes that can propagate erroneous coded fragments.**

We propose in the paper to add homomorphic hashes to this system. This allows for instantaneous detection of erroneous fragments and thus avoids decoding with wrong data. We describe the integration of this mechanism in coded blockchains and we evaluate its complexity theoretically and by simulation.

I. Introduction

One of the most interesting properties of blockchains is their decentralized nature, making it possible not to use central authorities. Usually, each node participating in the blockchain must maintain it by participating in the consensus when inserting a new block and by storing the entire blockchain. However, the success of blockchains such as Bitcoin or Ethereum has highlighted a scalability problem. Indeed, the increasing size of these blockchains means further constraints for medium-capacity nodes, leading to consequences on the availability and decentralization of the blockchain.

In order to allow nodes with a limited storage capacity to participate in the storage of the blockchain, several works introduced and studied *coded blockchains* by using erasure codes or network coding. The main principle is to store only some *coded fragments* of each block. These fragments are obtained by first splitting a block into k fixed size fragments and then generating linear combinations of these fragments. These linear combinations can be randomly generated [1] or can follow a structured code such as Low-Density Parity Check (LDPC) codes [2] or Fountain codes [3]. The average number of source blocks included in a linear combination is called the degree d.

When a node wants to join the network, it needs to download and verify each block of the blockchain, generate coded fragments, and then delete the original blocks to keep its coded fragments only. When the node wants to restore a block, it downloads $k(1+\epsilon)$ coded fragments and then performs the reverse operation. The value of the real number ϵ can vary from 0 (ideal code) up to 0.5 according to d and the used code.

However, these propositions do not consider adversarial nodes that can provide maliciously formed coded fragments and thus prevent the correct decoding of the whole block.

To detect bad coded fragments, we propose in this paper to use *homomorphic hashing* functions that were introduced by [4] and improved by [5] in the context of peer-to-peer distributed storage systems. The main property of these hashes is that the hash of a linear combination of source blocks can be expressed as a function of the linear combination coefficients and hashes of source blocks. Thus, if the hashes of the source blocks are public and certified, any user can verify the validity of a received block by checking that its hash corresponds to the output of the verification function.

This paper is structured as follows. Sections II presents a description of erasure-code based low storage nodes using homomorphic hashes, the main contribution of this paper. Afterwards, Sections III and IV present the interest of our low storage blockchain node and an analysis of the available parameters of our system. Finally, Section V concludes this paper and exposes ways to further this topic.

II. Including Homomorphic Hashes in Coded Blockchains

In this article we propose to use homomorphic hashing functions on coded blockchains in order to detect erroneous coded fragments. It will then be possible to replace them by correct ones, and to list malicious nodes.

Before describing the coding operation, let us first define some notations. We denote by $N^{(i)}$ the nodes of the network, where i is an unique identifier characterizing each node. We denote by $B^{(j)}$ the j^{th} block of the blockchain. We consider that the first block is $B^{(0)}$.

We consider that hashing and coding operations are done on the finite field $\mathbb{Z}_p$, where p is a prime number of size $|p|$ bytes. Let s_B the maximum size of a block of the blockchain (in bytes). Let us define two integers k and r respectively corresponding to the number of fragments of a block and the number of coded fragments stored by a node. The size of a fragment is denoted by $m = s_B/|p|$. The choice of the values of system parameters as k, r and p will be discussed in Section IV.

A. Coding the data

1) Block Splitting: The block $B^{(j)}$ is split into k fragments $F_l^{(j)}$, with $l = 1, \ldots, k$. The fragments are themselves composed of m finite field elements. The elements of the fragment $F_l^{(j)}$ are denoted $f_{l,v}^{(j)}$, where $v = 1, \dots, m$.

The last fragment can be padded if needed, in case of variations in block size.

2) Homomorphic Hash of the Block: As defined in [4], we consider that the following public parameters define the system $G = (p, q, g)$ where q is a large prime number such that $q|(p-1)$. The vector g is composed of some elements $g_i \in \mathbb{Z}_p$, for $i = 1, \dots, m$. The hash of the block $B^{(j)}$ corresponds to the set of the hashes of its fragments : $h(B^{(j)}) = (h(F_1^{(j)}), h(F_2^{(j)}), ..., h(F_k^{(j)}))$ where:

$$h(F_l^{(j)}) = \prod_{v=1}^{m} g_v^{f_{l,v}^{(j)}} \mod p$$

3) Coded Fragments Generation: To build the coded fragment $\mathfrak{F}_u^{(i,j)}$, where $0 \leq \ u \ \leq \ r-1$, the node considers the k coefficients $\{\alpha_{k.u}^{(i,j)}, \dots, \alpha_{(k+1).u-1}^{(i,j)}\}$ and computes the following linear combination:

$$\mathfrak{F}_u^{(i,j)} = \alpha_{k.u}^{(i,j)}.F_1^{(j)} + \dots, \alpha_{(k+1).u-1}^{(i,j)}.F_k^{(j)}$$

We assume that the values of $\alpha_{k.u}^{(i,j)} + v$ can be deduced from i and j. From a more practical point of view, the v^{th} element of the coded fragment $\mathfrak{F}_u^{(i,j)}$ is defined by the finite field element $\mathfrak{f}_{u,v}^{(i,j)}$ computed as follows:

$$\mathfrak{f}_{u,v}^{(i,j)} = \alpha_{k.u}^{(i,j)}.f_{1,v}^{(j)} + \dots, +\alpha_{(k+1).u-1}^{(i,j)}.f_{k,v}^{(j)}$$

The k coefficients $\{\alpha_{k.u}^{(i,j)}, \dots, \alpha_{(k+1).u-1}^{(i,j)}\}$ depend on the chosen erasure code.

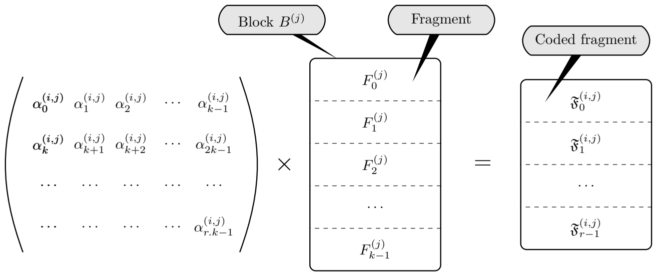
Block $B^{(j)}$
Fragment
Coded fragment
$\alpha_0^{(i,j)}$ $\alpha_1^{(i,j)}$ $\alpha_2^{(i,j)}$ $\cdots$ $\alpha_{k-1}^{(i,j)}$
$\alpha_k^{(i,j)}$ $\alpha_{k+1}^{(i,j)}$ $\alpha_{k+2}^{(i,j)}$ $\cdots$ $\alpha_{2k-1}^{(i,j)}$
$\cdots$ $\cdots$ $\cdots$ $\cdots$ $\alpha_{r.k-1}^{(i,j)}$
×
$F_0^{(j)}$
$F_1^{(j)}$
$F_2^{(j)}$
$F_{k-1}^{(j)}$
=
$\mathfrak{F}_0^{(i,j)}$
$\mathfrak{F}_1^{(i,j)}$
$\mathfrak{F}_{r-1}^{(i,j)}$

Fig. 1: Block coding process

4) Hash of the Coded Fragments: The hash of each of the r coded fragments is then computed. We denote it by $\mathfrak{h}_u^{(i,j)} = h(\mathfrak{F}_u^{(i,j)})$. Thanks to the homomorphic property, it can be proved that

$$h(\mathfrak{F}_u^{(i,j)}) = \prod_{v=1}^{k} h(F_v^{(j)})^{\alpha_{k.u+v-1}^{(i,j)}} \tag{1}$$

5) Storing Data: Finally, the node removes the block $B^{(j)}$ and replaces it by the r coded fragments $\mathfrak{F}_1^{(i,j)}, \dots, \mathfrak{F}_r^{(i,j)}$, their r hashes and the k hashes of the initial fragments.

B. Recovering the data

When a Low Storage node (LS node) wants to recover a block from coded fragments stored by different LS nodes, it executes the following steps illustrated on Fig. 2.

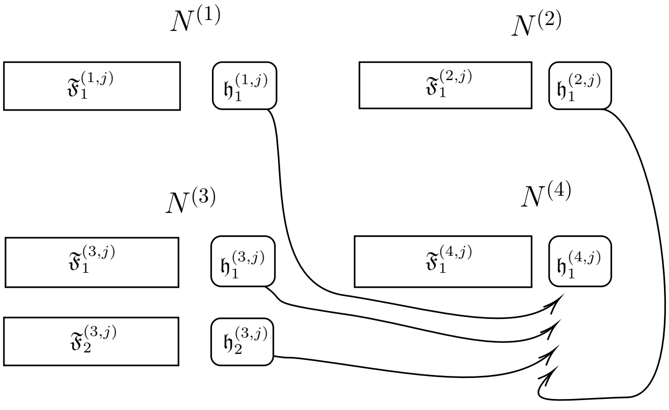

Fig. 2: Homomorphic hashes verification

1) Download the Coded Fragments Hashes: As represented in Fig. 2, node $N^{(4)}$ first downloads $k(1+\epsilon)$ hashes $\mathfrak{h}_z^{(i,j)}$ from different nodes.

2) Hash Check: The node checks that the downloaded hashes are correct by verifying Eq. 1. Recall that the values of $\alpha_{k.u}^{(i,j)}$ can be deduced from i and j.

3) Download the Coded Fragments: If the hashes are verified, the node $N^{(i)}$ downloads coded fragments associated with the previous hashes $\mathfrak{F}_0^{(i_0,j)}, \dots, \mathfrak{F}_{r-1}^{(i_{r-1},j)}$ from several nodes $N^{(i_0)}, \dots, N^{(i_{r-1})}$. Note that it is possible to request multiple coded fragments from the same node

4) Coded fragment Check: The node hashes each received coded fragment to verify that it matches the corresponding received hash.

5) Block Decoding: Once the fragment hashes are verified, the block can be decoded. After downloading a sufficient number of coded fragments, the node will have enough equations to invert the linear system and recover the k fragments (and thus the block) from the coded fragments.

III. Secure Low storage node interests

The main objective of traditional LS nodes is to allow any node to contribute to an entire blockchain with a reduced storage effort. The addition of homomorphic hashes increases the security of the distributed coding process and allows for the identification of malicious nodes.

A. Scalability

1) Storage effort scalability: With traditional coded blockchain, a node only store r/k data form each block. Let's define the compression factor $c = r/k$.

One of the interests of our system is its scalability. Indeed, each node can adapt r according to, for example, the age of a block by simply removing some of its stored coded fragments without re-calculating them.

Moreover, the number of coded fragments generated and stored on each node can independently be defined by each node, and should be adapted according to the desired storage effort of each node.

2) Availability: One of the main goals of our system is to improve the global availability and sustainability of a blockchain. By reducing the storage effort needed to participate, we expect more participants storing at least one coded fragment of every block. This means that for a system with a large amount of nodes, any node can then leave the system or be unreachable without significantly impacting the availability.

3) Network improvement: With the increase of the amount of nodes, we improve the distribution of the blockchain over the network. Our low storage nodes can allow for the decongestion of the network.

B. Malicious node identification

With homomorphic hashing, it becomes possible to identify malicious nodes in the network, providing incorrect coded fragments. A simple solution to avoid them is to locally blacklist them and avoid contacting them in the future.

But we can also imagine a network level impact, where cheaters are publicly denounced. An incentive system can be easily set up, by punishing malicious nodes and rewarding senders of valid denunciations. It can be done by using fraud proofs system, as presented in [6].

IV. Analysis of the parameters

One of the challenges is to determine k, r and the security parameters, with the best compromise between compression, complexity and security. In this section, we will present some consequences when varying these parameters.

A. Type and size of the finite field

The linear combinations of the code and the hash operations are performed on finite fields. Practically, the data of the blocks are grouped into bit vectors of fixed length which are associated to finite field elements and processed with the corresponding rules. The homomorphic property of the hash implies that the code and the hash use the same finite field. With the considered type of hash, a finite field of type $\mathbb{Z}_p$, where the operations are performed modulo a large prime number p must be used.

The choice of the finite field impacts the probability of block recovery from downloaded coded fragments (which is better with a large finite field) and the encoding and decoding complexities (which is smaller with a small finite field). The size of the finite field is also a security parameter because a minimal value is necessary to avoid collisions. Under these constraints, the choice of a value of p with $1024-bit$ length is chosen, as suggested in [4].

B. Processing coding complexity

The complexity of encoding consists in multiplying a $k \times r$-matrix by the k original fragments. Then, there is $d \times r \times s_B/k$ operations in the finite field, so when $d = k$, the encoding complexity is $r \times s_B$. So, the encoding complexity does not depend on k, but only on r.

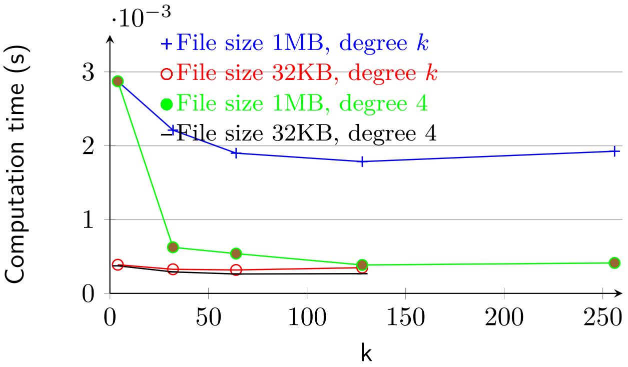

Fig. 3: Coded fragment time generation, from different size file and k.

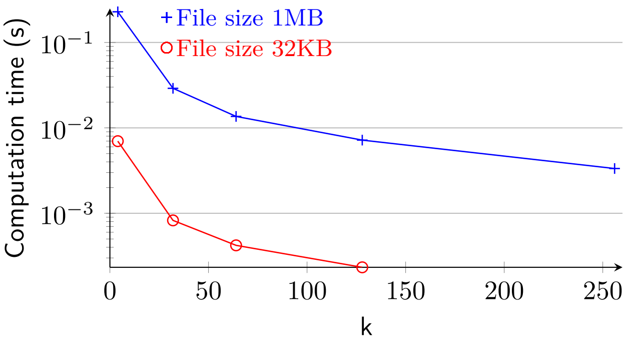

Fig. 4: Time to hash a fragment

Fig. 3 shows the encoding speeds of 1MB and 32 kB blocks, with different values of degrees and k. The implemented code was run in a Virtual Machine with operating system Debian 10. This personal computer runs Windows 10 and is equipped with an Intel Core i5-7300HQ @2.50GHz with 8GB of RAM. This graph allows to conclude that the processing cost is acceptable. Indeed, coding speed is always around milliseconds.

For decoding, the complexity consists in inverting a $k \times k$-matrix, and then multiplying it by the k coded fragments. The pseudo-random matrix inversion has a complexity in $O(k^3)$. So the number of operations is $O(k^3) + k^2 \times s_B/k = O(k^3) + k \times s_B$ and thus depends only on k. If the size of the block is large compared to k, then the matrix-vector multiplication ($k\times$ block size) is the most complex operation.

C. Homomorphic hashing functions complexity and parameters

According to [4] and [5], the complexity to hash a m-element fragment is $O(m)$, and thus a file of k fragments is hashed in $O(k.m)$. Fig. 4 confirms that, because when k increases, the fragment size m decreases, and so does the time.

To check the validity of the hash of a coded fragment from d source hashes, the complexity is $O(m)+O(d)$. Fig. 5 shows us that the fragment size m is not so important

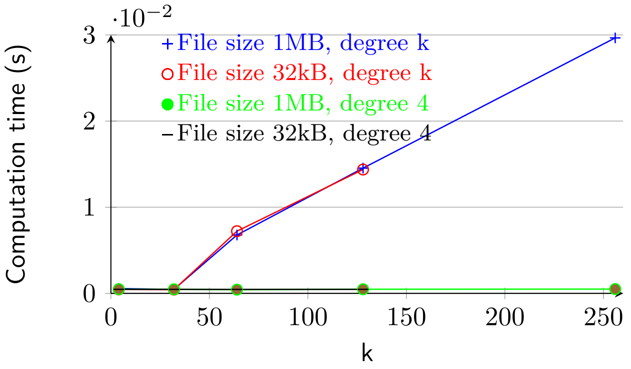

Fig. 5: Time to generate an homomorphic hash of a coded fragment from original fragment hashes

k	4	32	64	128	256
r = 1	0.251	0.0355	0.0239	0.0243	0.0369
r = 5	-	0.161	0.0870	0.0561	0.0529

TABLE I: Values of c for variation of k and r, with $s_B = 1MB$

comparing to the degree d, in terms of complexity. When this degree is low, for example 4, the time is low too (around 0.0004 s). But when it is equal to k (i.e. coded fragments are composed by linear combinations of every fragments), the time increases when k does.

The parameters are defined at the system level and are therefore the same on all nodes. This choice is important, because it will have a direct influence on the level of security but also on the complexity of the operations to be performed.

The time to perform homomorphic hash is independent of k.

D. Compression factor

With the homomorphic hashes system, the compression factor c changes. In this system we have to store extra data : homomorphic hashes of all the original fragments and homomorphic hashes of all the coded fragments. The new formula is so:

$$c = \frac{(k+r) \times S_H}{S_B} + \frac{r}{k} \tag{2}$$

To find the optimum of this equation, we can calculate:

$$k_{opt} = \sqrt{\frac{r \times S_B}{S_H}} \tag{3}$$

E. Chosing k and r

As described in Section II-A, each node, in order to generate its r coded fragments, will split the initial block into k fragments. The choice of this parameter can be different for each blockchain, but it must be the same for every user of the same blockchain.

When k increases, there is no impact on block hashing time and encoding speed, but in the end the nodes need to verify more coded fragments, so it will be longer.

The choice of r is up to the end user and will depend on the type of user. Choosing a large r will improve block recovery and reduce network load, as well as improve the overall blockchain availability as increasing r increases the storage effort of a node. It also improves the recovery block speed, because the nodes will verify less $(k+\epsilon-r)$ hashes and coded fragments. Choosing a small r will reduce coding complexity and compression factor. Globally r must be chosen according to the node's capacities.

If we want a better compression factor, we can use the formula 3, with $r = 1$, and we can calculate the optimal k. If we want an even better compression factor, we can also increase s_B by grouping some blocks before coding them. But during decoding, we will reconstruct more data than we need.

The Table. I presents some compression factors for different k and r values, for block size equal to 1MB like in Bitcoin. According to Eq. 3, the higher factor compression is 0.228, when $k = 88$ for $r = 1$, and 0.0512, when $k = 198$ for $r = 5$.

V. Conclusion

The main contribution of this paper is to introduce homomorphic hashes in coded blockchains. We explained how to compute, store and exchange these hashes in order to detect erroneous coded fragments. The impact of this mechanism in terms of additional storage and complexity was analyzed. A global analysis of the parameters was proposed in order to determine the parameters of the system. Future work will focus on the optimization of the parameters according to the considered blockchains and the types of nodes in order to find the best compromise between compression, complexity and security.

References

[1] D. Perard, J. Lacan, Y. Bachy, and J. Detchart, "Erasure code-based low storage blockchain node," in *2018 Cybermatics, IEEE Conference on Blockchains*. IEEE, 2018, pp. 1622–1627.

[2] H. Wu, A. Ashikhmin, X. Wang, C. Li, S. Yang, and L. Zhang, "Distributed error correction coding scheme for low storage blockchain systems," *IEEE Internet of Things Journal*, 2020.

[3] S. Kadhe, J. Chung, and K. Ramchandran, "Sef: A secure fountain architecture for slashing storage costs in blockchains," *arXiv preprint arXiv:1906.12140*, 2019.

[4] M. N. Krohn, M. J. Freedman, and D. Mazières, "On-the-fly verification of rateless erasure codes for efficient content distribution," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[5] C. Gkantsidis and P. Rodriguez Rodriguez, "Cooperative security for network coding file distribution," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006, pp. 1–13.

[6] M. Al-Bassam, A. Sonnino, and V. Buterin, "Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities," *arXiv preprint arXiv:1809.09044*, 2018.