RoVEr: Robust and Verifiable Erasure Code for Hadoop Distributed File Systems

Teng Wang*, Nam Son Nguyen *, Jiayin Wang †, Tengpeng Li *, Xiaoqian Zhang *,
Ningfang Mi ‡, Bin Zhao §, and Bo Sheng*
*Department of Computer Science, University of Massachusetts Boston, 100 Morrissey Boulevard, Boston, MA 02125
†Department of Computer Science, Montclair State University, 1 Normal Ave, Montclair, NJ 07043
‡ Department of Electrical and Computer Engineering, Northeastern University , 360 Huntington Ave., Boston, MA 02115
§ School of Computer Science and Technology, Nanjing Normal University, Nanjing, China

***Abstract*—Erasure Coding based Storage (ECS) is replacing tradition replica-based systems because of its low storage overhead. In an ECS, however, every task needs to fetch remote pieces of data for its execution, and data verification is missing in the current framework. As security issues keep rising and there have been security incidents occurred in big data platforms, the compromised nodes in a computing cluster may manipulate its hosted data fed for other nodes yielding misleading results. Without replicas, it is quite challenging to efficiently verify the data integrity in ECS. In this paper, we develop RoVEr, which is an efficient and verifiable ECS for big data platforms. In RoVEr, every piece of data is monitored by its checksums stored on a set of witnesses. Bloom filter technique is used on each witness to efficiently keep the records of the checksums. The data verification is based on the majority voting. RoVEr also supports a quick reconstruction of Bloom Filter when a node recovers from a failure. We present a complete system framework, security analysis, and a guideline for setting the parameters. The implementation and evaluation show that RoVEr is robust and efficient against the attack from the compromised nodes.**

I. Introduction

Big data applications and platforms have become popular in recent years. The common paradigm is to deploy a large scale cluster to process the data in a parallel and collaborative approach. In the past few years, however, big data platforms have been targeted by security attacks, and some breach incidents have been reported in [1], [2]. Besides losing the data and computing resources on the compromised machines, the adversary could cause more damages to the entire cluster with the knowledge of the big data platforms. This paper studies the merging Erasure Coding based storage framework in a big data processing system, and aims to provide security protection against the attacks from internal compromised nodes.

Traditional distributed systems such as HDFS (Hadoop distributed file system) are designed based on data replication. An input file is split into equal-sized data blocks, and copies of these blocks are distributed among the cluster. In replica-based storage, lost data blocks can be recovered immediately, at the cost of a large storage overhead. Recently, Erasure Coding based storage (ECS) system is emerging as a more efficient alternative. In an ECS, an input file is stripped into data cells, and a certain amount of parity cells will be calculated based on the EC policy. When some cells are lost, the original files can still be reconstructed with the remaining cells in the same group. This scheme reduces the storage overhead and increases fault tolerance significantly.

However, the original EC scheme does not consider the security concern in an untrusted environment leaving vulnerabilities for potential attacks. First, in ECS, every task's input data includes pieces from remote nodes, i.e., there is no local data block as in the traditional HDFS. A compromised node can affect a lot of tasks running on other nodes by manipulating the data it hosts. Second, without replicas, it is more challenging to verify the data integrity in ECS.

In this paper, therefore, we develop a new Erasure Coding scheme called RoVEr with enhanced security protection that allows each node to verify the received data before executing its tasks. Basically, we assign every node a set of witness nodes that monitor the data cells it hosts. When a node fetches a data cell from a remote node, it will check with the remote node's witnesses to verify the data. To be feasible and efficient, each data cell is represented by its checksum and our solution use Bloom Filter structure on the witness nodes to record all the data cells they monitor. In this paper, we present a complete system design, security analysis against potential attacks, and the guideline for setting the parameters. The performance is evaluated by simulation with various configurations. The results confirm the robustness and efficiency of our solution

II. Related Work

There has been existing work on EC implementation in the literature such as Zebra [3] and Quantcast [4] that are based on RAID [5]. To improve the performance of an EC based system, the authors in [6] developed HACFS, which dynamically converted policies between two erasure codes according to read/write operation frequencies of files. CherryPick [7] leveraged Bayesian optimization with Gaussian process to minimize customer's cost in a cluster by looking for a satisfied configuration with limited calculation overhead. EC-Cache [8] designed a Erasure Coding based caching strategy to overcome the load imbalance caused by big data block

This work was partially supported by National Science Foundation grant CNS-1552525, National Science Foundation Career Award CNS-1452751, and UMass Boston Oracle Doctoral Research Fellowship.

caching. Recently, Hitchhiker's guide [9] is introduced as a new EC scheme. It produced three different erasure coding schemata based on a combination of Reed-Solomon and XOR calculation considering various factors. All of these work can be easily integrated with our security framework in this paper.

Furthermore, security issues have also been explored from different aspects. AONT-RS [10] is a method based on All-Or-Nothing and Reed-Solomon algorithms. They started by adding one known word at the end of each data block, then checking the last bytes of re-constructed data to verify this block. Verifying Distributed Erasure-coded Data [11] proved the fingerprinted cross checksums can be consistent with data fragments. It indicates that the verification that can be switched to the checksum sequence instead of the fragment itself, which makes a better usage of the computation and network resources under the guarantee of accuracy. Many distributed storage systems, AVID [12] for example, can benefit from this solution, especially tolerate Byzantine clients [13]. Two-Server-Multi-Clients (TSMC) verifiable computation service [14] was introduced to verify outsourcing data focusing on high accuracy criteria. In [15], the authors proposed a protection scheme for distributed storage system based on encryptions where the focus is on key distribution and management. These work targeted on different security problem, and none of them considered the attacks of manipulating the data from a compromised node.

Finally, our solution use Bloom Filter which is an efficient tool for checking the existence of an element in a set. It has been well studied in the literature, and new alternatives are also proposed, such as Cuckoo filter [16] and BfMR [17]. All the variant implementations of Bloom Filter or similar filters can be integrated with our solution in this paper.

III. Background

In this section, we introduce essential knowledge of two major components in our design, Erasure Coding (EC) and Bloom Filter. We use Hadoop Distributed File System (HDSF) as an example in the following introduction.

A. Erasure Coding

For a storage framework serving big data platforms, there are two major performance metrics: (1) the number of simultaneously fault tolerance, and (2) the storage efficiency. The traditional HDFS is based on replication. It splits every file into HDFS blocks, making copies of every HDFS block and distribute them among a cluster. By default, HDFS makes two more copies for every data block, which tolerances 2-block loss with a 200% space overhead.

Comparing with traditional replication strategy, Erasure Coding based Storage (ECS) is a new scheme with lower storage overhead and higher fault tolerant. In ECS, the system splits an input file into data cells, then calculates a certain amount of parity cells. This process is called encoding. Even if some machines in the cluster fail, the original file can still be recovered with the remaining cells, either data or parity cells. This reconstruction process is called decoding.

Every EC based storage system can specify its own policy. Each policy defines a cell size and an EC Scheme. An EC Scheme includes the number of data and parity cells in one EC group (e.g., 3+2, 6+3 or 10+4), as well as one codec algorithm (e.g., XOR or Reed-Solomon [18]). The default EC setting of HDFS3.0.0 [19] is (6+3, Reed-Solomon,1MB). With HDFS3, a 1GB file will be stripped into 1024 data cells, each 6 data cells should generate 3 parity cells, so totally $1024/6 \cdot 3 = 512$ parity cells. These cell exist in many HDFS blocks. One HDFS block can host either data or parity cells only, and we call it either data block or parity blocks. If current HDFS block size is set as 64MB, one HDFS block will contain at most 64 EC cells, either data or parity. Cells on a data (parity) block will be assigned to 6 (3) different machines.

B. Bloom Filter

Bloom Filter is a data structure that we use to test if an element exists in a set. A common design of a Bloom Filter is a binary array with κ Hash functions and two basic operations: adding and testing.

Initially every bit in the array is '0'. When being added, each element will be hashed κ times, κ locations in the Bloom Filter will be set into '1'. After adding in the whole set, the Bloom Filter becomes an array with multiple '0's and '1's. To test the existence of an element, we hash it κ times, and check if all corresponding κ bits are all '1's. If they are, the Bloom Filter confirms that this element exists in the set. However there is a chance for this decision to be wrong. This probability is the false positive rate of the Bloom Filter, noted as α.

We use Bloom Filter in RoVEr because its space cost is low. The length of the Bloom Filter, noted as $\mathcal{M}$ can be affected by the number of elements in the set, noted as n, together with κ and α. Usually the optimal $\mathcal{M}$ and κ are calculated as

$$\mathcal{M} = \frac{-n \cdot log\alpha}{(log2)^2},\ \kappa = round(\frac{\mathcal{M}}{n} * log(2)) \qquad (1)$$

Bloom Filter is also highly time efficient. Since there is no need to iterate through the whole Bloom Filter, time complexity for both adding and testing is $\ominus(\kappa)$.

IV. System Model and Problem Formulation

In this section, we briefly introduce the system model considered in this paper, and formulate our target problem.

Hadoop YARN/HDFS Setting. We consider a Hadoop YARN cluster consisting of one master node and n slave nodes. All slave nodes are data nodes forming the HDFS, i.e, each slave node hosts partial data in HDFS. The HDFS data block size is indicated by B. We assume the HDFS adopts the erasure coding scheme for hosting the data. The erasure coding parameters are represented as $EC(d, p)$ indicating that p parity cells are generated per d data cells, and any d out of these $d+p$ cells can reconstruct the d data cells. We use C to denote the cell size in $EC(d, p)$, and generally $C \ll B$. The current implementation of EC in HDFS considers striped data layout

(as shown later in Fig. 1). Thus, for any data block that will be processed by a task, the executing node has to fetch data cells from other nodes and concatenate them to form the data block. If the node holds a part of the cells, it needs to contact the other $d-1$ nodes in the group, otherwise, the data cells will be transferred from other d nodes.

Adversary Model. We assume that at most $\mathcal{A}$ slave nodes could be compromised by the adversary. The adversary can access all the data hosted on the compromised nodes. In addition, the adversary can manipulate the data and messages sent by the compromised nodes to other slave nodes and the master node. In particular, we consider that the adversary may launch the following three attacks,

- *Manipulate attack*: This is the major attack we consider in this paper, where a compromised node may manipulate the hosted data cells sent to a victim node, and convince the node to accept them;
- *Replay attack*: This is a special case of manipulate attack. Instead of manipulate arbitrary data cells, the adversary may send to the victim another legitimate cell.
- *Undermine attack*: This is another minor goal of the adversary, that is to undermine the data verification process when it exists, e.g., mislead other nodes to consider a legitimate node as a compromised node.

Objectives. In this paper, we aim to develop a verifiable and robust storage system with the following two requirements: (1) a legitimate slave node is able to verify the data block received from other slave nodes with a certain security guarantee; (2) the system can tolerate F simultaneous node failures or data cell corruptions. In addition, we mainly consider two performance metrics: (1) the storage overhead incurred by the protection scheme; (2) the impact on data reconstruction when node failures occur. We will specify these objectives and performance metrics with more details after we introduce our design.

Table I is a summary of the notations we will use later.

n	number of data nodes
d, p	EC parameters, d data cells and p parity cells
B/C	HDFS block size / EC cell size
S	the total data size on each data node
$\mathcal{A}$	number of compromised nodes
F	number of failed nodes that can be tolerated
M	bit length of the Bloom Filter
k/K	number of hash functions in a BF / hash function pool size

TABLE I: Notations

V. Design of RoVEr

In an HDFS with EC framework, a node running a data processing task, e.g., a map task in MapReduce, needs to fetch data cells from other nodes to form the input data block. In an untrusted environment considered in this paper, the node needs to verify the received cells before conducting the process task.

A. Motivation

There are two traditional approaches to verifying the data cells. The first approach is based on public key infrastructure assuming each node's public key is known by other nodes in the cluster. When dispatching the data cell, the system can attach a signature by an authorized node, either the client node that submits the data or the name node of the cluster. In the latter case, the name node can sign the hashed value of the data cell to avoid transferring the data contents. Then, when a node fetches the needed cells from other nodes, the signatures will be received and the node can easily verify the integrity of the data cells. In practice, however, the number of cells in a HDFS block is large, e.g., in a regular HDFS setting where $B = 256M$ and $C = 64K$, there are $\frac{B}{C} = 4096$ cells. Conducting such a large number of cryptographic operations yields a large computation overhead when uploading and processing the data.

The second approach is to use checksum which is a common technique for detecting data errors. When a data cell is written to the HDFS, the system calculates and stores its checksum. Then, when a node receives a data cell, it will generate the checksum of the received data contents and compare to the checksum stored in the system. The data integrity is verified if they match. Our solution in this paper adopts the second approach aiming to incur the minimum overhead into big data processing jobs. However, the complete system design is still challenging because of the following two concerns.

Security Concern. The challenge of using checksums is where to store the initial checksums. In regular file systems, checksums are often stored with the data blocks as a part of the meta data. In our setting with possibly compromised nodes, however, the checksums cannot be stored on the same node with the data cells because the adversary can easily replace the checksums to match the manipulated data cells. Therefore, the system needs to store a data cell and its checksum on different nodes. Since our solution adopts the similar framework, we first define the following three roles involved in this approach for a given data cells,

- **Host**: the node that stores the data cell;
- **Witness**: the node that stores the checksum of the data cell;
- **Verifier**: the node that fetches the data cell for executing a task and needs to verify its integrity.

Basically, a verifier fetches a data cell from the host, and verifies the integrity with the assistance of the witness. In the untrusted setting we consider in this paper, the compromised nodes are also the witnesses of some data cells and they may manipulate the hosted checksums to undermine the verification process. Therefore, for each data cell, we need a group of witnesses to hold the checksums, and when they provide inconsistent checksums, the majority is taken and the nodes presenting different checksums are considered as compromised nodes. Therefore, in our setting, considering the maximum number of compromised nodes ($\mathcal{A}$) and concurrently failed nodes (F), each data cell needs at least $W = 2 \cdot \mathcal{A} + F + 1$ witnesses to guarantee that the legitimate nodes outnumbers the compromised nodes in any witness group.

Efficiency Concern. Since the verification is a frequent process with every task, it is desired that each node loads the checksum table in the memory to avoid disk I/O overhead.

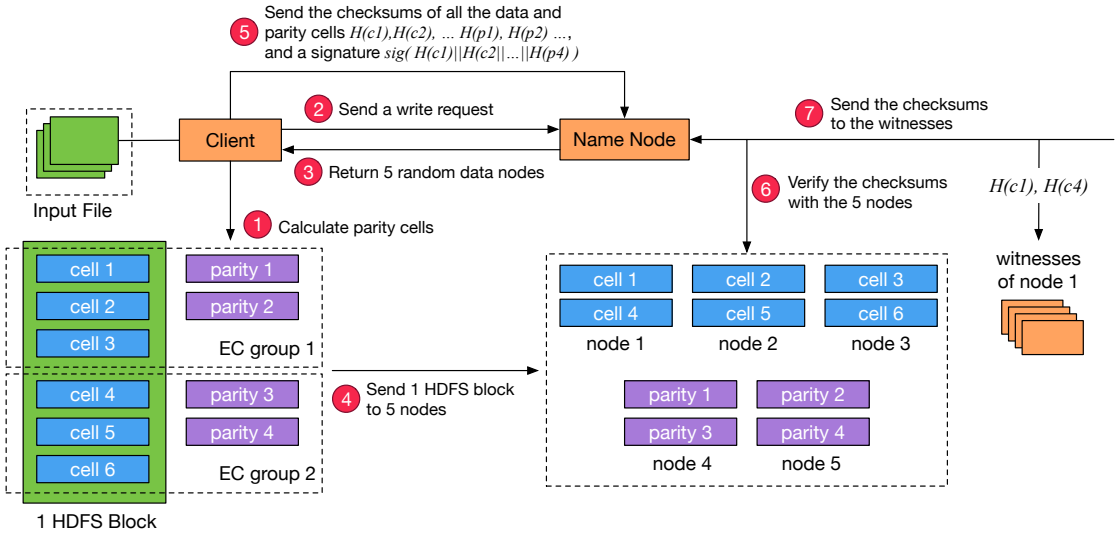

Fig. 1: Write operation ($EC(3+2)$)

However, as a big data processing platform, the HDFS usually hosts a large volume of data consisting of many data cells. The checksum table could be too big to fit in the memory space without affecting regular computation tasks.

For example, consider a setting with $B = 256M, C = 64K, \mathcal{A} = 3, F = 3$, each cell needs $w = 10$ witnesses. Assume we use $EC(6+3)$ erasure coding scheme, i.e., $d = 6$ and $p = 3$, and SHA256 to calculate the checksums for all the cells including data cells and parity cells. Each node maintains a checksum table that includes file ID, cell ID, and the checksum for each data cell the node witnesses. Assume the file ID and cell ID take 4 bytes, we need to store 36 bytes for each cell. Assume each node host $S = 1T$ data, the total additional checksum data kept on each node will be $S \cdot n \cdot \frac{d+p}{d} / C \cdot w \cdot 36/n = 8.85G$. This is a quite heavy burden for big data platforms, as regularly the memory allocated for executing a task is about a few Gigabytes.

In our design, therefore, we use Bloom filters to represent a set of checksums instead of keeping a checksum for each data cell. The benefit is apparently the small memory size. But it also inherits the false positive of Bloom filter that may be utilized by the adversary. In the rest of this section, we present the basic design of RoVEr, and the two major operations in the process.

B. Design

In our design, when a data cell is written to the HDFS, its checksum is calculated. Multiple replicas of the checksum are distributed across the cluster. We use w to indicate the number of replicas. In addition, each data node i is assigned a set of w witnesses represented by $W_i(|W_i| = w)$.

For node i that is a witness of j, i.e.g, $i \in W_j$, it keeps a Bloom filter j, indicated by BF_{ij}. The Bloom filter represents all the data cells stored on node j. We assume that the Bloom filter uses k different hash functions, f_{ij} indicate the j-th hash function used by node i. The system defines a pool of K hash functions for each data node i, and each witness node j of node i ($j \in W_i$) selects k hash functions from the pool to construct its own Bloom filter. When a checksum is added into the Bloom filter, the following Algorithm 1 is applied. Besides the cell content, its host ID, the file ID and cell ID are also included in the hash function in line 2. The purpose is to distinguish the cells with the same content against replay attack. When queried with a checksum, each node will apply regular Bloom filter test operation and return the boolean result to the requesting node. The details will be presented in the following two subsections.

Algorithm 1: Add a checksum at node i

Input: $H(c)$: checksum of cell c, h: host of c,
FID/CID: input file ID / cell ID of cell c

1 **for** (f_{ij}, $j \in [1,k]$) **do**
2 $\quad x = f_{ij}(h||H(c)||FID||CID)$;
3 $\quad BF_{ih}[x] = 1$;
4 **end**

C. Write operation

The details of writing a file to the HDFS in our solution is illustrated in Fig. 1. The client submitting an input file first divides the file into cells and generate the parity cells according to the EC parameters. For example, in $EC(3+2)$ scheme, every 3 cells are grouped together and 2 additional parity cells are calculated (step 1 in Fig. 1). Multiple EC groups including data and parity cells form a HDFS block according to the setting of HDFS block size. Then in steps 2–4, following the current implementation of EC in HDFS, the client sends a request to the name node and receives a set of randomly selected data nodes to store the data. After that, an HDFS block will be sent to each data node in a stripped layout as shown in Fig. 1. Note that for $EC(d+p)$, the write request is sent when $d+p$ HDFS blocks are ready counting the size of data cells and parity cells. And the number of data nodes selected by the name node is also $d+p$.

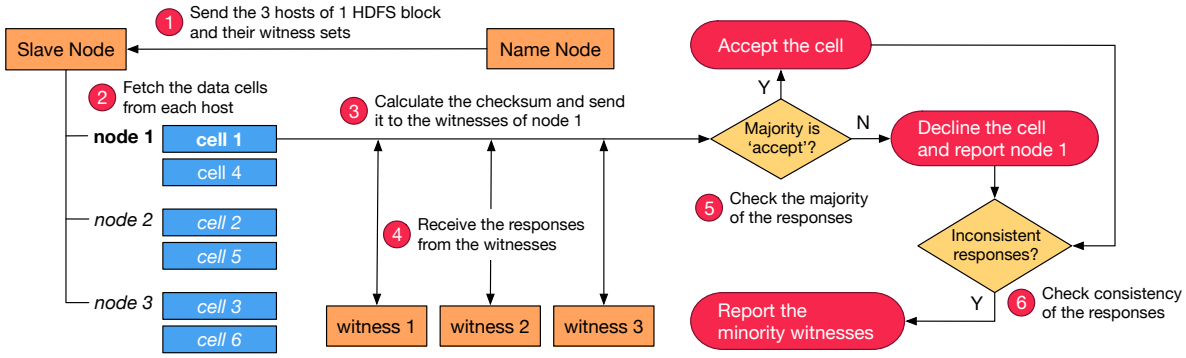

Fig. 2: Read operation ($EC(3+2)$ and 3 witnesses for each cell)

In RoVEr, for security protection, we have the following additional steps. In step 5, the client will send all the checksums from the input file, and a signature of the concatenation of them to the name node. We assume a public key infrastructure has been deployed between the name node and slave nodes. The name node is aware of each slave node's public key. In this step, the client submitting the input file, provides the evidence of the checksums, and the signature by its private key preserves the non-repudiation security property. Meanwhile (in steps 6–7), each data node hosting at least one new HDFS blocks from the input file will send the checksums of the new cells to the nodes in its witness set. The data node also contacts the name node, and verifies the checksums it calculates with the ones that name node received from the client. This step can be done before or after sending the checksums to its witness nodes. After the checksum is verified, it will be added into each witness node's Bloom filter following Algorithm 1.

D. Read operation

When a slave node is assigned a map task, it needs to fetch the corresponding input data from the HDFS. This read operation is illustrated in Fig. 2. The name node hosts the directory service maintaining the locations of each data block (HDFS block). With EC storage, each block is distributed across multiple data nodes. In step 1, the name node sends all the hosts of the input data block as well as their witness sets to the requesting slave node.

Then, in step 2, the slave node fetches the data cell from multiple data nodes, and assemble the HDFS block as the input data for the map task. If one or more data cells are missing, it will further fetch the parity cells in the same EC group, and launch the EC reconstruction process to recover the data cells. In RoVEr, each received data cell or parity cell has to be verified before using it in any computation. From the name node, the requesting slave nodes receives the witness set of each data node that hosts a data cell or parity cell. It calculates the checksum of the cell received from a data node, and sends it to the data node's witness nodes (step 3). Those witness nodes will check their own Bloom Filters, and respond 'accept' if the checksum exists, and 'decline' otherwise (step 4). The verifier node then checks if the majority of responses from the witnesses. If it is 'accept', the data cell is considered legitimate, otherwise, the data cell will be discarded and the host node is reported as a suspicious node. In any case when there are inconsistent responses, the minority witnesses will be marked as suspicious nodes.

E. Bloom filter reconstruction

When a node recovers from a failure, its Bloom Filters are lost and have to be reconstructed. In practice, it takes an extremely long overhead to re-add all the checksums. Therefore, in RoVEr, the Bloom Filter reconstructed by aggregating the Bloom Filters from other nodes. Specifically, the recovered node contacts each witness group it belongs to, and finds a subset of the witnesses whose union of their hash functions cover the hash functions it selects. The node then fetch the Bloom Filters of those witnesses, and merge them with OR operation. For example, assume node a, b, and c belong to the same witness set, and their choices of two hash functions are (H_1, H_2), (H_1, H_3) , (H_2, H_4). If node a recovers from a failure, it can fetch node b and c's Bloom Filters for reconstruction, because the union of hash functions include (H_1, H_2). While this reconstruction is quite fast, the down side is that the resulting Bloom Filter has a higher false positive because it is built with more hash functions than the original one. In RoVEr, we pick the subset of the witnesses that yield the fewest additional hash function. The increased false positive is one of the performance objectives we will optimize in the next section.

VI. Security Analysis and Parameter Optimization

In this section, we present the security analysis and use the result to derive the optimal parameters in RoVEr. We focus on the manipulate attack mentioned in Section IV. The other two attacks, replay attack and undermine attack, can be prevented by using signatures and unique identifier of each data cell, shown in Algorithm 1 and Algorithm 2. The detailed analysis is omitted due to the page limit.

Basically, we consider that when a compromised node receives a request for a data cell it hosts, it will try to manipulate the cell and pass the verification of its witnesses. The adversary is given a limited time to respond as a long

delay would be suspicious to the requesting node. In the rest of this section, we first present the best attacking strategy for the adversary, and then analyze the probability of a successful attack. At the end, we derive the parameters in RoVEr that fulfill the security requirement.

A. Adversary Strategy

The basic attacking strategy for the adversary is to keep generating a random data cell within the given time, and send the cell with the highest probability to pass the verification to the requesting node. The key component in the attack is how to determine the likelihood of passing the verification of the witnesses.

Before further analysis, we first define a **single-hashed** Bloom Filter, which is the Bloom Filter constructed by using one hash function. A regular Bloom Filtercalculated by k hash functions can be considered as the aggregation of k single-hashed Bloom Filter by the OR logic operation. To distinguish with single-hashed Bloom Filter, we also call the regular Bloom Filter multi-hashed Bloom Filter in this paper. For instance, as shown in Fig. 3, BF_1 is a multi-hashed Bloom Filter generated by $(H1, H_2)$. The two hash functions can create two single-hashed Bloom Filters respectively, noted as F_1 and F_2, and $BF_1 = F_1 \vee F_2$.

Given a target witness set, we assume that the adversary is aware of the hash function pool this witness set uses, but does not know the choice of each witness. The best strategy to attack is to find a cell that can pass the tests of the most single-hashed Bloom Filter. The more single-hashed Bloom Filter a data cell can pass, the more likely it will convince a witness. If the hash functions a witness chooses are all included in the passed set of single-hashed Bloom Filter, the manipulated cell will definitely pass the verification of this witness.

The details are presented in Algorithm 2. For each manipulated cell c, we enumerate each hash function H_i in the pool, and exam if c is included in the single-hashed Bloom Filter F_i. The variable $maxCount$ keeps the largest counter, and ret is the data cell that yields the bets value.

Algorithm 2: Attack strategy of a malicious node

```
while before the deadline do
    Manipulate a data cell c, count ← 0;
    for H_i in hash function pool do
        if isIncluded(F_i, c) then count++;
    end
    if count > maxCount then
        maxCount = count;
        ret = c;
    end
end
return ret;
```

B. Security Analysis

Our analysis includes the following three components: (1) Given the deadline (e.g., τ trials), how many single-hashed Bloom Filters (SHBF) can be compromised (passed)? (2) Given the number of compromised single-hashed Bloom Filters, how likely will a witness be convinced? (3) Finally, what is the overall success attack rate after convincing at least $\frac{w}{2}$ witnesses?

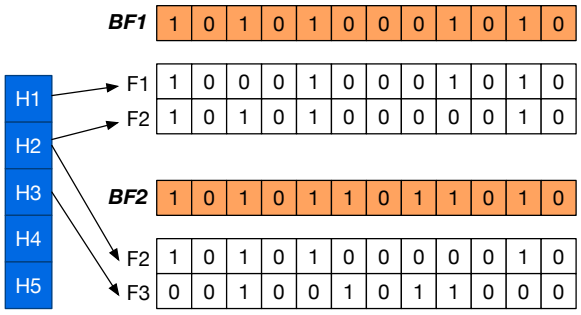

Fig. 3: Two Bloom Filters with Overlapping Hash Functions

Probability of compromising SHBFs. Recall that the bit length of the Bloom Filter is M. After adding one checksum, each bit has $\frac{1}{M}$ probability to become '1'. Therefore, for a given manipulated cell and a single-hashed Bloom Filter, the probability for this cell to be accepted is

$$\mathcal{P}_s = 1 - (1 - 1/M)^{\lceil \mathcal{S}/C \rceil}, \tag{2}$$

where $\frac{S}{C}$ is the number of cells hosted by each data node. Thus, the probability of the cell being accepted by ϵ single-hashed Bloom Filters is

$$\mathcal{P}(\epsilon) = \binom{\epsilon}{\mathcal{K}} \mathcal{P}_s^{\epsilon} \cdot (1 - \mathcal{P}_s)^{\mathcal{K}-\epsilon}. \tag{3}$$

After τ trials, the probability of finding at least one such cells is

$$\mathcal{P}(\epsilon, \tau) = 1 - (1 - \mathcal{P}(\epsilon))^{\tau}. \tag{4}$$

Probability of convincing a witness. Now consider that ϵ hash functions have been compromised by the adversary, and each witness' Bloom Filter is formed by k hash functions. Assume x of these k hash functions are compromised, and the remaining $k - l$ Bloom Filters are from the rest of the pool. The manipulated cell needs another specific $k - l$ bits to be '**1**' in the multi-hashed Bloom Filter, and these '**1**'s can be produced by either the x compromised hash functions or the other $k - l$ hash functions s as well. The probability of having all necessary '**1**'s is

$$\mathcal{P}_{mul}(\epsilon, x) = \frac{\binom{x}{\epsilon}\binom{k-x}{K-\epsilon}}{\binom{k}{K}} \cdot (1 - (1 - \mathcal{P}_s)^k)^{(k-x)}. \tag{5}$$

Probability of a successful attack. A compromised node needs to convince at least $\frac{w}{2}$ witnesses to launch a successful

attack. According to the previous analysis, therefore, the probability of a successful execution is

$$\mathcal{P}_{suc} = \sum_{\zeta=\frac{w}{2}+1}^{w} \sum_{x=1}^{k} \sum_{\epsilon=1}^{K} \mathcal{P}(\epsilon,\tau) \cdot \mathcal{P}_{mul}(\epsilon,x)^{\zeta} \cdot (1-\mathcal{P}_{mul}(\epsilon,x))^{w-\zeta} \quad (6)$$

C. Parameter Setting

The above analysis derives the attack success rate as a function of all the involved parameters. We aim to set the proper values for the parameters so that the attack success rate is lower than a user-specified threshold, i.e., $\mathcal{P}_{suc} < \delta$. However, the function of $\mathcal{P}_{suc}$ is a quite complex combination of parameters, and affects other performance metrics such as storage overhead. Therefore, in this paper, we simplify this step with a heuristic process that separately sets a group of parameters according to different factors.

First, the parameters for Bloom Filter, the bit length (M) and the number of hash functions (k), are determined by the number of cells on each data node and a system-specified target false positive. Second, the number of witnesses for each data node is configured based on the storage overhead. The size of the witness set is the same as the number of Bloom Filters each node has to maintain.

Finally, we focus on setting the hash function pool size (K). This is an important parameter in $\mathcal{P}_{suc}$, and it affects the performance of the reconstruction of Bloom Filters. Intuitively, if the hash function pool size is small, the false positive of the reconstructed Bloom Filter will be slightly increased. On the other hand, if the pool size is extremely large, then the hash functions selected by each node has few or even no overlappings. It will be hard, if not impossible, to reconstruct the lost Bloom Filters. In RoVEr, we set K to be the minimum value that satisfies $\mathcal{P}_{suc} < \delta$.

VII. Implementation and Evaluation

This section presents how we implement and evaluate RoVEr for both performance and cost perspectives. The results show that RoVEr can provide a feasible configuration with a specific cluster security requirement.

A. Implementation

We implement the process of RoVEr with Java 8 and run it on a dedicated machine with 8 CPUs (3.4 GHz) and 32 GB RAM. The simulator is built with following main components:
(1) **Core.** This component contains necessary material for any testbed, which includes Node simulation, logging and history, job Generator and Runner, cluster configuration. One can define the number of nodes and witnesses in a cluster, after that, witnesses will be assigned randomly and uniformly for every node. When a job needs to be processed, its tasks will be randomly assigned among the cluster. To optimize the performance, scheduling policies are out of the scope of this project. We also provide an interface as well as implementations for the attacking strategy. Moreover, with this interface, future designed attacks can be plugged-in easily. In the task processing stage, if a node is playing the role of an attacker, it will modify the data by guessing cells following Algorithm 2. Each instance of node is built in with a recovery mechanism. That is, when in offline mode, the node will contact as few nodes as possible to recover all Bloom Filters stored on it.
(2) **HDFS.** This component defines the essential elements of a distributed file system, including data block and cell, Logic_Block, Erasure Code Policy and lastly File object. A user can specify a file size, block size, cell size in Bytes and an Erasure Code policy. The files will be generated and filled up by random bytes according the user's requirement. After that, data blocks are distributed among the cluster randomly. Each file is considered as one job, so that its Logic_Blocks are taken into tasks which will be processed by the job Runner of Core component.
(3) **Security.** This component is the primary reflection of our RoVEr, simulates all the security related functions. We implement the Bloom Filter objects for the witnesses, based on a Hash-Assignment mechanism. In detail, every node has a certain amount of initial seeds which are random long numbers representing each hash function. When a Bloom Filter checks its cells, it calculates each cell's hashcode. Each hash function generates one random number, whose seed is set to be the initial seed plus hashcode. This random but deterministic number defines the bit location in the Bloom Filter that will be set to 1.

After file blocks are assigned among the cluster, Hash-Assignment mechanism creates a hash function pool for every node and assigns a portion of this pool (a number of hash functions in the pool) to every witness node. In the attack stage, a certain number of machines ($\mathcal{A}$) will be randomly chosen and set to be attackers in the cluster by utilizing our handful implemented attack types.

B. Evaluation

To show the performance of RoVEr, we examine two main parameters: witness number and hash function pool (pool size). In our default setting, we consider the adversary can apply $\tau = 10^7$ trials to manipulate a data cells, the number of adversary is $\mathcal{A} = 1$, the concurrent failures is $F = 2$. Therefore, the default number of witnesses for each data node needs to be at least $2 \cdot \mathcal{A} + F + 1$, thus $w = 5$. In addition, we consider a cluster of 50 nodes, and by default, each node hosts 20K data cells. In our evaluation, each parameter configuration is tested repeatedly with 100 independent jobs, and we show the average values in this section.

1) Impact of Witness Number: We first examine the impact of the number of witnesses assigned to each data node. Intuitively, the more witnesses there are, the harder the adversary's attack can be successful, because more witnesses have to be convinced by the manipulated data cell.

In Fig. 4, the number of witnesses ranges from 5 to 10. We compare the curves of $K = 10, 15, 20$. The trends of the curves are decreasing as expected. The change is sharper when the pool size is smaller (e.g., $K = 10$). In our tested case, with

10 witnesses, the successful attack rates of the three curves drop to 1% or lower. Note that our tests include 100 individual runs, so 1% is the smallest unit we can illustrate in the figures.

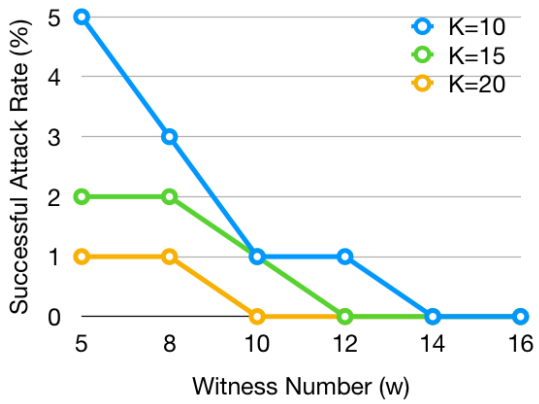

Fig. 4: RoVEr successful attack rate with various witnesses

Another concern with the number of witnesses is the memory cost. Fig. 5 shows that when a node hosts $1TB$ files, the memory cost for one single Bloom Filter can be as high as $1GB$. When a node witnesses $10B$ nodes at the same time, the memory cost can reach $10GB$, which can be a burden for certain types of machines.

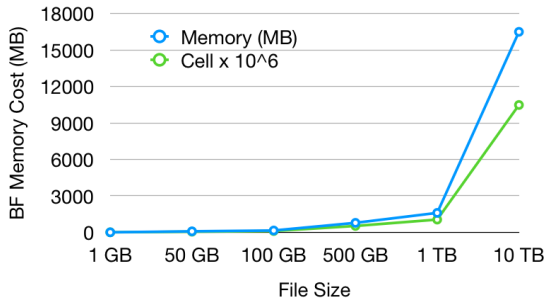

Fig. 5: Bloom Filter memory cost on each node

2) Impact of Pool Size: Next, we evaluate the size of the hash function pool (K) which is also an important factor for the security protection. We fix the witness number to be $5, 8$, and 10 and increase the pool size from 10 to 30 with an interval of 5. The results of successful attack rates are shown in Fig. 6. The curves are decreasing with larger hash function pools, and show a long tail after significant drops in the beginning. Given a deadline with a limited number of trials (τ), the number of compromised hash functions is certain. When the hash function pool is larger, it is less likely for a witness to pick a hash function from the set of compromised hash functions. And we observe that this improvement is more effective when the pool size is small. After the pool size is sufficiently large, including more hash functions does not help much with the security concern.

In addition, we also evaluate our theoretical analysis and guidelines for setting the pool size mentioned in Section VI. We omit the details due to the page limit. But our derived pool sizes satisfy the security requirements (we set target successful rate $\delta = 0.01, 0.0.2, 0.03$) referring to the results in Fig. 6.

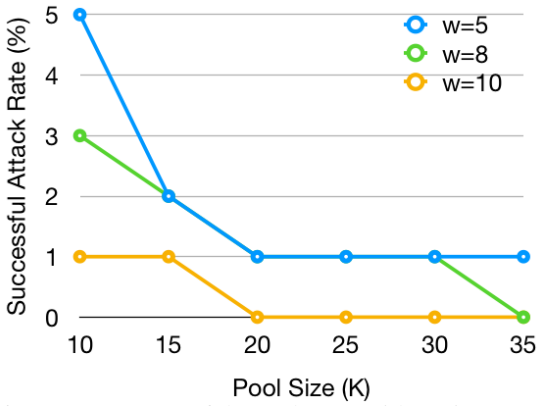

Fig. 6: RoVEr successful attack rate with various pool size

Should one notice that the capacity of each node is set to be $2 * 10^4$ cells in above tests. In real systems more cells can be expected. Fig. 7 indicates that RoVEr maintains stable successful attack rates even with more cells on each machine. By enlarging pool size from 10 to 15, the attacking rate can degrade from 4% to 0% for $3 * 10^4$ cells on each node.

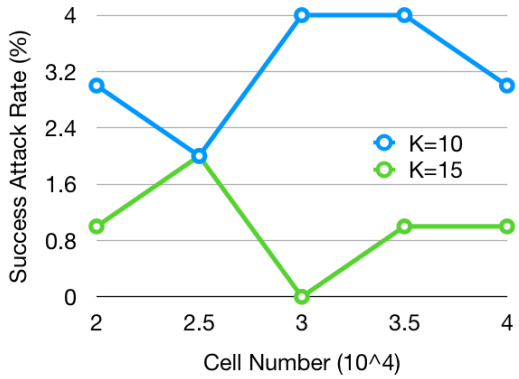

Fig. 7: RoVEr successful attack rate with various cell number

3) Recovery Cost: A larger Hash Function Pool can degrade the risk of successful attacks, however the pool size is also limited by the recovery phase. If the pool is larger than a certain threshold, it will take more Bloom Filters from other nodes to recover the lost one. Merging too many Bloom Filters will generate a high false positive Bloom Filter, which is a hidden danger in the future verification on that node.

In this test, we assign 5 witnesses for each node in a 50 nodes cluster and fix the false positive ratio to be 0.05. Then take down one node in each cluster, and use minimum number of Bloom Filters from other nodes to recover this one. We run 100 tests for each setting with pool size varies from 10 to 20.

A fully recovery can be achieved if the failure node can recover all Bloom Filters on it. Fig. 8 tells that the larger the pool is, the harder a fully recovery can be. With 5 witnesses, a size of 10 Hash Function Pool can guarantee a recovery chance of 10%. Taking Fig. 6 and Fig. 4 into consideration, with a successful rate target as 5% for this cluster, a size 15 hash pool is enough to provide a sufficient low attacking rate and a 90% recover rate.

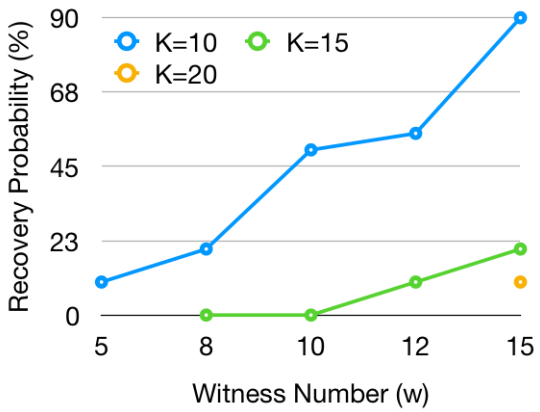

Fig. 8: Bloom Filter recovery possibility

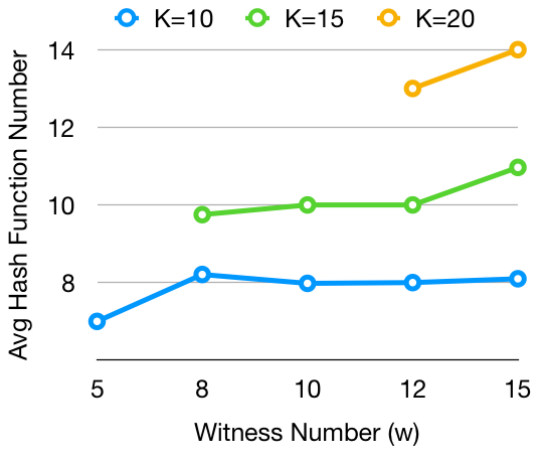

Fig. 9: number of hash function used in recovery

To recover a Bloom Filter from a large hash pool, it requires more single-hashed Bloom Filters from other witnesses. For a fixed size Bloom Filter, the more single-hashed Bloom Filters are merged, the larger the false positive rate can be. One can tell the average number of single-hashed Bloom Filters that are used in recovery and the accumulated false positive rate in fully recovery from Fig. 9 and Fig. 10. For example, to recover a Bloom Filter from a size 10 Hash function pool with 5 witnesses, averagely 7 single-hashed Bloom Filters will be merged together, which increase the false positive rate by 150.71%. This increase can arrives 748.73% for a size 20 hash function pool with 15 witnesses.

VIII. Conclusion

This paper develops RoVEr, which is a new strategy to protect data integrity in a cluster computing environment. By assigning witnesses to every machine in a cluster, every piece of data will be verified before being used in a task execution. We demonstrate the possible manipulate attacks from compromised nodes, and RoVEr can provide suitable cluster configurations to suppress the successful attack rate under a desired threshold.

References

[1] Attackers start wiping data from CouchDB and Hadoop databases. https://www.csoonline.com/article/3159534/security/attackers-start-wiping-data-from-couchdb-and-hadoop-databases.html.

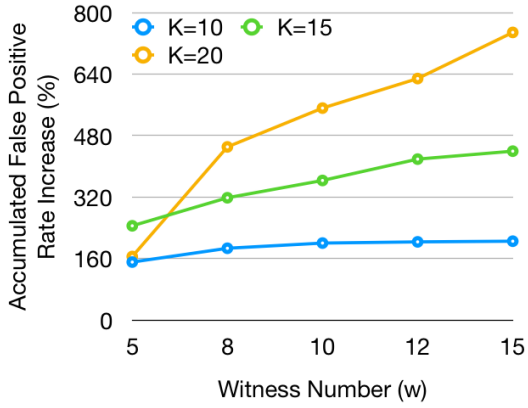

Fig. 10: Bloom Filter false positive increase after recovery

[2] Insecure Hadoop Clusters Expose Over 5,000 Terabytes of Data. https://thehackernews.com/2017/06/secure-hadoop-cluster.html.
[3] John H Hartman and John K Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.
[4] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.
[5] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.
[6] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. A tale of two erasure codes in hdfs. In *FAST*, 2015.
[7] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, pages 469–482, 2017.
[8] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding.
[9] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review*, 44(4):331–342, 2015.
[10] Proc. 9th USENIX Conference on File and Storage Technologies. Aont-rs: Blending security and performance in dispersed storage systems. In *FAST*, 2011.
[11] James Hendricks, Gregory R Ganger, and Michael K Reiter. Verifying distributed erasure-coded data. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 139–146. ACM, 2007.
[12] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*. IEEE, 2005.
[13] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 115–124. IEEE, 2006.
[14] Ying Wu, Rui Zhang, Rui Xue, and Ling Liu. Multi-client verifiable computation service for outsourced data. In *Web Services (ICWS), 2017 IEEE International Conference on*, pages 556–563. IEEE, 2017.
[15] Hsiao-Ying Lin and Wen-Guey Tzeng. A secure decentralized erasure code for distributed networked storage. *IEEE transactions on Parallel and Distributed Systems*, 21(11), 2010.
[16] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.
[17] Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim. Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pages 100–105. ACM, 2012.
[18] Stephen B Wicker. *Error control systems for digital communication and storage*, volume 1. Prentice hall Englewood Cliffs, 1995.
[19] https://hadoop.apache.org/docs/r3.0.0-alpha1/hadoop-project-dist/hadoop-hdfs/hdfserasurecoding.html.