TELKOMNIKA Indonesian Journal of Electrical Engineering
Vol. 12, No. 9, September 2014, pp. 6445 ~ 6453
DOI: 10.11591/telkomnika.v12i9.5163

Implementation of a File System with Encryption and De-duplication

Xiaoning Jiang, Wenhua Zhou, Yang Leng
College of Information and Electronic Engineering, Zhejiang Gongshang University
Hangzhou, P.R.China, 0571-28877758
Corresponding author, e-mail: jiangxiaoning@zjgsu.edu.cn, vanora.zhou@gmail.com,
feiyangenator@gmail.com

Abstract

With the rapid advance of society, especially the development of computer technology, network technology and information technology, there is an increasing demand for systems that can provide secure data storage in a cost-effective manner. In this paper, we propose a prototype file system called EDFS (Encryption and De-duplication File System), which provides both data security and space efficiency in storage systems. We adopt de-duplication technology called CDC (Content-Defined Chunking) to chunk the file and compute its unique fingerprint, lookup different blocks, then store different blocks on server to ensure storage efficiency. Thinking of security, we use convergent encryption to encrypt data blocks, which not only ensure data security but also improve the possibility of de-duplication. Finally, we describe a prototype implementation of EDFS based on FUSE and present an analysis of the potential effectiveness, using real data obtained from a runtime database.

Keywords*: chunking, convergent encryption, file system, backup*

1. Introduction

With the rising of information-nalized level of human society, the world's data capacity is growing at an alarming rate, which indicates that the era of great explosion of information has already come. According to statistics, until now the global amount of data has increased by 8 times from 2005 to 2012. It is expected that the global amount of data could reach 35 quadrillion in 2020. However, study found that up to 60% of these data is redundant and the traditional data compression [2] can only eliminate intra-file redundancy. So in order to solve the problem mentioned above, de-duplication, also known as single-instance storage, has been proposed as an efficient way to best utilize the given amount of storage. De-duplication identifies common sequences of bytes called chunks both within and between files, and only stores a single instance of each chunk regardless of the number of times it occurs. By doing so, de-duplication can dramatically reduce the space required to store a large data set.

Data security [3] is another field of great importance that must be taken into consideration in modern storage systems. But unfortunately, traditionally encryption and de-duplication are, to a great extent, diametrically opposed to each other. De-duplication makes use of data similarity to remove redundant information in storage space while the goal of cryptography is quite the contrary, which aims at making cipher text indistinguishable from theoretically random data. As a result, the goal of a secure de-duplication system is to ensure data security without compromising the space efficiency achieved from de-dupe techniques.

In this work, we develop a prototype file system for secure de-duplication, which allows data to be encrypted independently without invalidating data de-duplication. Also, we put particular effort in performing some physical experiments to analyze the performance behavior of the given technique. We examine the relationships among average chunk size, de-duplication ratio under different chunking algorithms, chunk sizes and data sets.

2. Related work

There mainly exist three approaches for eliminating redundant information: delta encoding, de-duplication, and compression. Current systems which aim at achieving efficient

Received November 21, 2013; Revised Apr 18, 2014; Accepted June 6, 2014

storage and highly utilized network bandwidth usually rely upon one of the mentioned techniques independently or use them in a combined manner. Delta encoding, also known as data differencing, is a way of storing or transmitting data in the form of differences between sequential data. It is used in many applications including source control and backup [4]. Rsync is a utility free software and network protocol that synchronizes files and directories from one location to another while minimizing data transfer by using delta encoding when appropriate [5].

In order to improve space efficiency, backup applications also take advantage of information redundancy to greatly decrease the amount of data to be backed up [6]. There are three primary chunking strategies of data de-duplication: whole-file chunking, fixed-size chunking and variable-size chunking. Whole-file chunking treats each file as a single block, typically uses the block's hash value as its identifier. Therefore, if more than one files hash to the same value, they are assumed to have identical contents and will only need to be stored once. Farsite [7] and the Windows Single Instance Store [8] both perform de-duplication on per-file bases. As for fixed-size chunking, varieties of preceding works exploit it for backup applications and large-scale file systems, such as [9] and [10]. Lessfs [11], a high performance inline data de-duplicating file system for Linux, is becoming more and more popular in enterprise solutions for reducing disk backups and minimizing virtual machine storage in particular. In addition, Opendedup, also called SDFS [12] is a file-system that supports in-line and batch mode de-duplication on both Linux and Windows, along with VMware virtualized environments. It can do de-duplication process either locally, on the network, or in the cloud (including Amazon S3). The variable-size chunking, also known as content-defined chunking splits files into variable-length chunks within a sliding window using a hash value. Variable-length chunking is widely used in various application domains of redundancy elimination such as backups [13], file systems [14], and data transfers [15]. In addition, some works proposed to adapt the most optimal chunking schemes based on the inner features of the file itself. According to this, Liu et al. proposed a scheme [16] that applies different chunking methods on the basis of the metadata of individual files. Jaehong et al. proposed context-aware chunking which shares the basic idea with Liu's work, but only use file extension rather than all file metadata [17].

Besides de-duplication, data security is another key factor that must be taken into consideration to build secure systems. Keyed encryption has been put into use to address data secrecy by many distributed systems, such as OceanStore [18] and e-Vault [19]. Cryptographic techniques utilized by the systems mentioned above make the assumption that all incoming data is already encrypted. Nevertheless, none of these systems attempt to exploit redundancy deletion to achieve storage efficiency. Some systems adapt security models at expense of space overhead rather than provide security and de-duplicated storage efficiency. For instance, PASIS [20] and POTSHARDS [21] achieve long-term security by using secret sharing, which leads to a very high storage overhead.

3. The EDFS Scheme

3.1. Chunking Module

Chunking is a method of scanning a file and splitting it into short elements. Each element is called a chunk and is a unit of redundancy detection. As mentioned above, there are three main chunking strategies of data de-duplication: whole-file chunking, fixed-sized chunking and content-defined chunking. Whole-file chunking is simple and fast, but it can only detect file duplicate since the entire file is viewed as a whole block [22]. For fixed-size chunking, a file will be break into fixed size pieces. However, because boundaries are chosen by offset, this method is very sensitive to the insertion and deletion operation. Inserting or deleting even a single byte will shift all the block boundaries following the modified point, resulting in entirely different blocks. To effectively address this problem, EDFS adopt the sliding window (a byte sequence of a given length) chunking algorithm (Figure 1) called content-defined chunking, also called variable-size chunking.

EDFS uses a fingerprint function which derives from Rabin's fingerprint algorithm [23] to compute fingerprints. These fingerprints are signatures for bounded window of byte stream. The fingerprint function is as follows:

$$\sum_{i=n-l}^{n} S_i \equiv r(\operatorname{mod} D) \tag{1}$$

Where S_i is a byte stream, $S = s_1 s_2 s_3 ... s_n$, l is size of the chosen sliding window, r and D are fixed values, and $r < D$. We call this equation a “target pattern”.

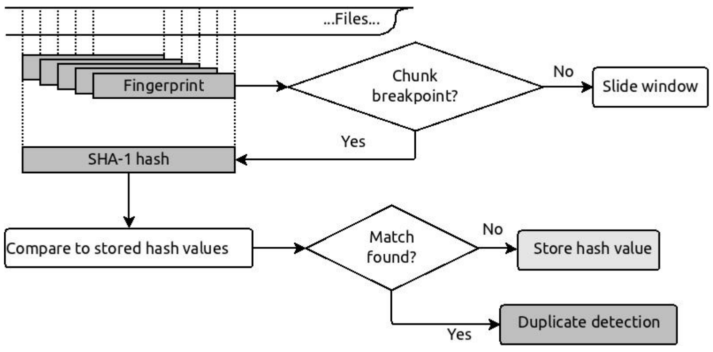

Figure 1. Sliding Window Chunking

If the sum of the l length bytes in the sliding window divides D with r remainder, EDFS will mark the current end of the window as a breaking point. The bytes between the current breaking point and the previous breaking point compose a chunk. Otherwise, the sliding window will be shifted by one byte to generate another fingerprint and compare again. As we can see, CDC algorithm holds a clear advantage over fixed-size since the boundary points are determined on the basis of the content of files, not the offset from the beginning. As a result, insertion or deletion will only affect the specific blocks where data inserted into or deleted from.

3.2. Encryption Algorithm

Traditionally, combining the space efficiency of de-duplication with the secrecy aspects of encryption is problematic since generally, different clients hold different keys, formula:

$$cipher\_text = encrypt(key, block) \tag{2}$$

Means after encryption, the same source block may generate different blocks. In this work, we adopt convergent encryption (Figure 2) to address the issue. Using this method, clients gain encryption key by using a function to calculate the hash value of the given block.

$$cipher\_text = encrypt(hash(block), block) \tag{3}$$

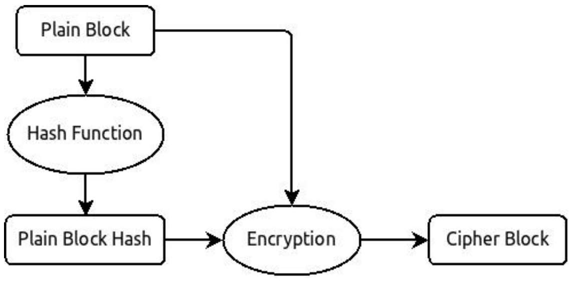

Figure 2. Convergent Encryption Module

Therefore, the same plaintext results in identical cipher text regardless of who does the encryption. In addition, EDFS utilizes XTS-AES to encrypt the chunk since it provides transparent encryption. We can insert an encryption/decryption module into an existing data path without changing data layout or message formats of other components under these paths. What's more, Dworkin [24] proves that “In the absence of authentication or access control, XTS-AES provides more protection than the other approved confidentiality-only modes against unauthorized manipulation of the encrypted data.”

3.3. FUSE

FUSE (Filesystem in userspace) is a framework (Figure 3) for unix-like operating systems. It was officially merged into the mainstream Linux kernel tree in kernel version 2.6.14. What’s more important, FUSE has distinctive features as follows: simple library API, secure implementation, usable by non privileged users and be proved very stable over time. Using FUSE, non-privileged users can create/access their own file systems without modifying kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces. What’s more, FUSE has already integrated several programming languages, which provides more choices for developers.

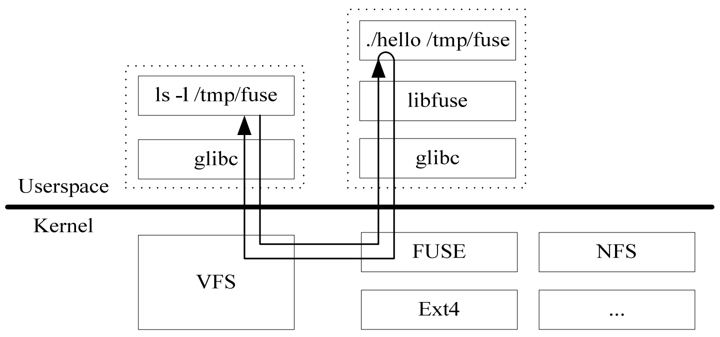

Figure 3. Architecture of FUSE

4. Prototype System

We develop a prototype file system called EDFS based on FUSE. The targets are as follows:

security: the ability to resist attacks at a moderate intensity level;

efficiency: detect and eliminate data redundancy without restricting to files of fixed set format;

performance: better read and write speed compared to some file systems that has already exists;

usability: transparent de-duplication and encryption that under control;

probability: be available across different platforms without considering validity of underling file system.

4.1. System Organization

The system is composed of several modules shown in Figure 4. De-duplication consists of chunking module, tmp_repository, map_db and meta_db. Buffer poll offers several service procedures for the incoming data. Each procedure serves an individual byte sequence (file). Further more, chunking module partitions the file into a number of chunks and store them in tmp_repository. Map DB and Metadata DB ate all in-memory databases for mapping the current block in memory to the right place on disk and recording file metadata in RAM before de-duplication and encryption, respectively. Block Manager receives blocks from tmp_repository, generates their fingerprints and checks if they already exist in the fingerprint table. If not, deliver the block to compression module, otherwise, delete the block, updating metadata DB, blockusage DB and fileblock DB.

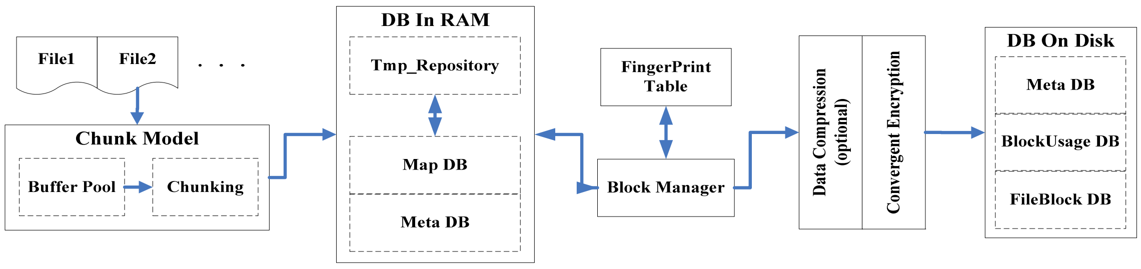

Figure 4. Architecture of EDFS

4.2. Data esngine

In order to improve efficiency of storage and query, we create secondary directories under the root storage directory, using the hash value of the block as the file's name. For example, if a block's hash value (SHA-256) is *9a32a228319266a0780e00bb6a9d7f6bf18e8dcd3a35a564* as h(b), then it will be stored in directory '/dta/9/A' where '9' stands for the first byte of h(b) and 'A' stands for the second byte of h(b).

4.3. Metadata Engine

Metadata engine consists of three individual files: blockusage.db, fileblock.db and metadata.db. We use commodity DBMS (Berkely DB) to implement them.

Blockusage.db is responsible for describe the attribute of every block. Its Key is the hash value of the stored block and its Value is structured as follow:

```
typedef struct {
    unsigned long long size;      /* size of the block */
     unsigned long long inuse; /* usage frequency */
} ;
```

Fileblock.db shows the relationship between a file and the blocks that compose it. Besides, it also records the inode information of the file on underlying file system. With inode and blocknr, we can easily restore the whole file using chunked blocks. Its Value is the hash value of the given block and the key's structure is as follow:

```
typedef struct {
   unsigned long long inode;      /* inode of the backed file */
   unsigned long long blocknr;              /* sequence of the block among all blocks */
} ;
```

Metadata.db is mainly used to store properties of backed files. It can be used to check the correctness of a newly restored file by comparing the recorded summary information. Its key is inode of the backed file and its value is structured as follow:

```
typedef struct {
   struct stat stbuf;                   /* the 'stat' struct */
   unsigned long long real_size;             /* file size after de-duplication and encryption */
   char filename[MAX_POSIX_FILENAME_LEN + 1];                   /* file name */
   unsigned long long md5sum;  /* md5sum of file as plaintext */
} ;
```

What's more, we use Tokyo Cabinet, a library of routines for managing a database, to implement metadata DB and map DB in RAM. By doing this, we can store some most frequently used blocks in memory, which will dramatically improve IO performance and speed up query evaluation.

4.4. Procedure of Basic Operations

In this section, we describe the algorithms of three mostly used basic operation functions of EDFS over files: open, read and write.

***Algorithm 1**:* edfs_open(path, fi)

```
1. procedure open(path, fi)
2. step 1: find the record where key=path in in-memory DB p2i
3.      if p2i[key]=NULL
4.          then go to step 4
5.      else i ← p2i[key].inode
6. step 2: find the record where key=i in in-memory DB mt
7.      if mt[key] ≠NULL
8.          then go to step 5
9. step 3: find the record where key=i in DB md
10.     if md[key]≠NULL
11.         then s ← md[key].stat
12. step 4: find the file f where f.stat=s from root directory recursively
13. step 5: update DB mt, p2i and in-memory DB o2b, set
14.     mt[key].stat ← s
15.     mt[key].maxblk++
16.     p2i[key].inode ← i
17.     o2b[key].of ← fi.offset
18.     o2b[key].blk ← mt[key].maxblk
17. step 6: allocate a idle circule buffer from the poll, and set
18.     tag ← i
19. end procedure
```

Algorithm 2: edfs_read(path, offset, fi)

```
1. procedure edfs_read(path, offset, fi)
2. buf[size] <-- 0
3. step 1: find the record where key=path in in-memory DB p2i, set
4.      i <-- p2i[key].inode
5. step 2: find the record where key.inode=i and key.of=offset in in-memory DB o2b, set
6.      blk <-- o2b[key].blk
7.      find the record where key.inode=i and key.blk=blk in in-memory DB wt
8.          if wt[key]=NULL
9.              then find the record where key.blk=blk in DB fb
10.               h <-- fb[key].hash
11.               find the block b where f.name=h under root directory recursively
12.               t <-- decode(b)
13.               t <-- decompress(t)
14.               buf <-- t
15.           else buf <-- wt[key]
16.     blk_offset <-- offset - o2b[key].of
17.     find the record where key=hash(blk) in DB blk_usage
18.         if blk_offset > blk_usage[key].size
19.             then offset <-- offset + blk_usage[key].size
20.             go to step 2
21.     return buf
22. end procedure
```

Algorithm 3: edfs_write(path, offset, fi)

```
1. procedure edfs_write(buf, size, offset, fi)
2. thread 1: find the record where key.inode=fi.inode and key.of=offset in in-memory DB
o2b, set
3.      blk <-- o2b[key].blk
4.      blk_offset <-- offset - o2b[key].of
5.      find the record where key.blk=blk in DB fb
6.      h <-- fb[key].hash
7.      find the block b where f.name=h under root directory recursively
8.      t <-- decode(b)
9.      t <-- decompress(t)
10.       tmp_buf <-- t
```

```
11.         for n <-- 0 upto blk_offset
12.             do write tmp_buf[n] to circule buffer tagged fi.node
13.         write size of data in buf to circule buffer tagged fi.node
14.         get the last stored checksum cs if any from in-memory DB wt
15.         if cs = NULL
16.             set first boundary point at the beginning of buffer
17.         else
18.             set the chunking point inherited from the last chunking operation
19.  repeat:   calculate cs of data within the given window
20.             if cs matches 'target pattern'
21.                 then store the finding block in wt
22.             else
23.                 if reache the end of the buffer
24.                     then stote a record in in-memory DB tmpct <fi.inode, checksum>
25.                 else sliding the window by one byte, go to repeat
26. thread 2: get a record r from wt
27.         h' <-- hash(r.block)
28.         inuse <-- blk_usage[h']
29.         if inuse=0 then
30.             cr <-- compress(r.block)
31.             er <-- encode(cr)
32.             insert a record of r.block in DB blk_usage
33.         else
34.             blk_usage[h']++
35.         insert a record <h', blk> into DB fb
36.         update the according record in DB md
37. end procedure
```

5. Experment

5.1. Experment Environment

Table 1 shows the testing environment we used to test the performance of EDFS vs. Lessfs and Rsync. We perform comprehensive analysis on various aspects of de-duplication.

Table 1. EDFS Performance Testing Environment

CPU	Intel(R) Core(TM)2 Quad CPU Q9500 @ 2.83GHz	Fuse	2.9.1
		Tokyocabinet	1.4.32
Memory	4GB	BerkeleyDB	4.8
Linux Core	2.6.38-8-generic	Lessfs	1.5.12
Gcc	4.5.12	Rsync	3.0.7

We use four sample data sets from a runtime database to measure the de-duplication efficiency of three different algorithms – Rsync. Lessfs and EDFS. We use the following command to export four data sets from the actual runtime Mysql every other day:

```
mysqldump –flush-logs –uroot –p zabbix > `date +'%m%d'`.sql
```

It is important to note that about one gigabit data will be inserted into the database every day, at the meantime, data of the earliest day will be dropped.

First, we do pair wise synchronization using rsync to see the difference between sample data sets. Table 2 shows the result. We can see that according to sliding window algorithm adopt by rsync, only a quarter data is different, which means that the de-duplication ratio will be up to about 75%.

Table 2. Pair Wise Synchronization of rsync

pair	total size(byte)	Sent(byte)	speedup
0914.sql vs. 0915.sql	4,289,154,207	942,146,224	4.55
0915.sql vs. 0916.sql	4,249,344,000	1,001,053,585	4.24
0916.sql vs. 0917.sql	4,300,787,825	1,050,324,618	4.09

We examine the degree of duplication of Lessfs for chunk size 128KB, 64KB, 32KB, 16KB and 8KB respectively (Table 3). The result shows that hardly any duplicate blocks were detected using fixed-size chunking method adopt by Lessfs. It's reasonable because for databases, changes caused by insertion are distributed across the whole DB system, which means that the fixed-size chunking blocks will be shifted.

Table 3. De-duplication of Lessfs on Sample Data Sets

file	size(byte)	chunk_no					dedup_no				
		128KB	64KB	32KB	16KB	8KB	128KB	64KB	32KB	16KB	8KB
0914.sql	4178862708	31883	63765	127529	255058	510116	0	0	0	0	0
0915.sql	4289154207	32724	65448	130895	261790	523579	0	1	2	4	8
0916.sql	4249344000	32420	64840	129680	259360	518719	0	0	0	0	0
0917.sql	4300787825	32813	65625	131250	262500	524999	0	1	2	4	8

As for EDFS, we chose to close the capabilities of compression and encryption since we are aiming at de-duplication. Table 4 describes the de-dupe result. We can conclude that within a DB file, just like fixed-size chunking, there are hardly any duplicate chunks. However, there's an obvious similarity between a pair. Two sample DB files, 4 GB each, will occupy only 5.5GB physical disk. For the second file, 2.5GB redundant data is removed, which means the de-dup rate can be up to 60%.

Table 4. De-duplication of EDFS on DB Set

file	Size(byte)	dedup_size(byte)	dedup_ratio	chunk_no	dedup_no
0914.sql	4,178,862,708	4,178,862,708	0	4,769	0
0915.sql	4,289,154,207	1,690,221,716	2.537	4,683	2,133
0916.sql	4,249,344,000	1,930,648,089	2.2	4,512	2,010
0917.sql	4,300,787,825	1,667,745,436	2.578	4,321	2,027

5.2. Testing Analysis

In database field, EDFS is not as good as Rsync on the aspect of de-duplication, but is much better than Lessfs. Besides, Rsync uses sliding window chunking algorithm which will generate so many small blocks called fragment, resulting in too much metadata which in turn severely degrade performance of IO and CPU. On the other hand, EDFS can provide transparent encryption while Rsync only transfer plain text, which is very dangerous over network.

Referring to writing and reading performance, Lessfs is better for the reason that CDC chunking will have to shift bytes one by one when searching boundary points. So as to reading, EDFS must do a DB query to find the inode of the block. What's worse, if data across several blocks, accordingly, query operation must be done a couple of times. However, to some extent, trading space for time really makes sense.

6. Conclusion

In this paper, we propose a prototype file system called EDFS (Encryption and De-duplication File System), which provides both data security and space efficiency in storage systems. This scheme demonstrates that security can be combined with de-duplication in a way that provides a variety of security characteristics. In addition, we describe several new techniques that result in storage efficiency and security. According to testing results, The EDFS scheme can be easily applied to backup/storage environment of database field.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (61272306), the Public Project of Science and Technology Plan in Zhejiang Province (2012C21001), the Public Project of Science and Technology Plan in Zhejiang Province (2013C31042), and the College Students' Science and Technology Innovation Activities Plan of Zhejiang Province (1130JQ4212058G).

References

[1] FUSE. *Filesystem in Userspace* [EB/OL]. http://fuse.sourceforge.net/.
[2] Haitang Wang. Research of Graph Compression in Information Storage. *TELKOMNIKA Indonesian Journal of Electrical Engineering*. 2011; 11(8): 4367~4371.
[3] Ali Hassan Sodhro, Ye Li, Madad Ali Shah. Novel Key Storage and Management Solution for the Security of Wireless Sensor Networks. *TELKOMNIKA Indonesian Journal of Electrical Engineerin.* 2013; 11(6): 3383~3390.
[4] Fanpeng Yu, Sidong Zhang, Xin Zhou. Incentive Mechanism Algorithm for P2P File System in Campus Networks. *TELKOMNIKA Indonesian Journal of Electrical Engineering*. 2012; 10(6): 1477~1484.
[5] Andrew Tridgell, Paul Mackerras. *The rsync algorithm*. TR-CS-96-05. 1996.
[6] B Zhu, K Li, H Patterson. *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, Proc.* FAST '08: Sixth USENIX Conf. File and Storage Technologies. 2008; 1-14.
[7] JR Douceur, A Adya, WJ Bolosky, D Simon, M Theimer. *Reclaiming space from duplicate files in a serverless distributed file system* Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02), Vienna, Austria. 2002; 617–624.
[8] WJ Bolosky, S Corbin, D Goebel, JR Douceur. *Single instance storage in Windows 2000.* In Proceedings of the 4th USENIX Windows Systems Symposium. USENIX. 2000; 13–24.
[9] S Quinlan, S Dorward. *Venti: A New Approach to Archival Storage.* Proc. Conf. File and Storage Technologies (FAST '02). 2002; 89-101.
[10] B Hong, DDE Long. *Duplicate Data Elimination in a San File System.* Proc. 21st IEEE / 12th NASA Goddard Conf. Mass Storage Systems and Technologies (MSST). 2004; 301-314.
[11] Lessfs. *Open source data de-duplication* [EB/OL]. http://www.lessfs.com/wordpress/.
[12] Opendedup. *Get more from your disk* [EB/OL]. http://www.opendedup.org/.
[13] LP Cox, CD Murray, BD Noble. Pastiche: Making Backup Cheap and Easy. SIGOPS Operating Systems Rev., 2002; 36(SI): 285-298.
[14] B Zhu, K Li, H Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. Proc. FAST '08: Sixth USENIX Conf. File and Storage Technologies. 2008; 1-14.
[15] JC Mogul, YM Chan, T Kelly. *Design, Implementation, andEvaluation of Duplicate Transfer Detection in http.* Proc. Symp. Networked Systems Design Implementation (NSDI '04),p. 4, 2004.
[16] C Liu, Y Lu, C Shi, G Lu, D Du, D Wang. ADMAD: Application-Driven Metadata Aware De-Duplication Archival Storage System. Proc. Fifth IEEE Int'l Workshop Storage Network Architecture and Parallel I/Os (SNAPI '08). 2008; 29-35.
[17] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient Deduplication Techniques for Modern Backup Operation, Computers. *IEEE Transactions.* 2011; 60(6): 824 – 840.
[18] S Rhea, P Eaton, D Geels, H Weatherspoon, B Zhao, J Kubiatowicz. *Pond: the OceanStore prototype.* Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST). 2003; 1–14.
[19] A Iyengar, R Cahn, JA Garay, C Jutla. *Design and implementation of a secure distributed data repository.* In Proceedings of the 14th IFIP International Information Security Conference (SEC '98). 1998; 123–135.
[20] GR Goodson, JJ Wylie, GR Ganger, MK Reiter. *Efficient Byzantine-tolerant erasure-coded storage.* In Proceedings of the 2004 Int'l Conference on Dependable Systems and Networking (DSN 2004). 2004.
[21] MW Storer, KM Greenan, EL Miller, K Voruganti. *POTSHARDS: secure long-term storage without encryption.* In Proceedings of the 2007 USENIX Annual Technical Conference. 2007; 143–156.
[22] W ANG Can, QIN Zhi-guang, PENG Jing, W ANG Juan. *A Novel Encryption Scheme for Data Deduplication System.* In Proceedings of International Conference on Communications, Circuits and Systems (ICCCAS 2010). IEEE Computer Society. 2010: 265-269.
[23] Micheal O Rabin. *Fingerprinting by random polynomials.* Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
[24] Morris Dworkin. NIST Special Publication 800-38E .NIST Special Publication. 2010; 800: 38E.