KVEFS: Encrypted File System based on Distributed Key-Value Stores and FUSE

Giau Ho Kim, Son Hai Le, Trung Manh Nguyen, Vu Thi Ly, Thanh Nguyen Kim, Nguyen Van Cuong, Thanh Nguyen Trung, and Ta Minh Thanh

Le Quy Don Technical University
No 236 Hoang Quoc Viet Street , Hanoi, Vietnam
trungthanh@lqdtu.edu.vn

Abstract. File System is an important component of a secure operating system. The need to build data protection systems is extremely important in open source operating systems, high mobility hardware systems, and miniaturization of storage devices that make systems available. It is clear that the value of the data is much larger than the value of the storage device. Computers access protection mechanism does not work if the thief retrieves the hard drive from the computer and reads data from it on another computer.
Encrypted File System (EFS) is a secure level of operating system kernel. EFS uses cryptography to encrypt or decrypt files and folders when they are being saved or retrieved from a hard disk. EFS is often integrated transparently in operating system There are many encrypted filesystems commonly used in Linux operating systems. However, they have some limitations, which are the inability to hide the structure of the file system. This is a shortcoming targeted by the attacker, who will try to decrypt a file to find the key and then decrypt the entire file system.
In this paper, we propose a new architecture of EFS called KVEFS which is based on cryptographic algorithms, FUSE library and key-value store. Our method makes EFS portable and flexible; Kernel size will not increase in Operating System.

Keywords: File System in User Space (FUSE), Key-Value store, Encrypt File System, KVEFS, Data Protection

1 Introduction

Security of the stored data on disk is an important area. The theft of the stored data may cause losing of personal information. It can be done through copying data from the system via any thumb devices. To ensure security from such kind of theft, the obvious solution through restricting users to use any thumb device especially pen drives. But such kind of restriction causes many problems because now a day use of thumb devices is a must for working properly, there is a huge amount of data transfer regularly on such devices. Imagine for a day, you lose a computer, if you think the access control methods to prevent the thief from getting the data in the computer then you are wrong. They only need to get the hard disk from your computer and put it into another one, so all data is readable. The solution for that is to encrypt all the data on your hard disk. There are many encrypt filesystem on linux such as encfs, ecryptfs. These systems have shown the effectiveness of protecting hard drive data against hackers. However, for systems like encfs, the fact that the user opens the encrypted folder will see the number of files, directories, subdirectories (even if they are encrypted), and also the time last modification, date of creation of the directory, file, the disclosure of the directory structure is also a certain limitation of the existing file encryption system. It provides several important informations for the hacker to attack our file system. Therefore, our idea is to implement a

DOI: 10.5121/ijnsa.2019.11204

file system where the encrypted folder does not display the same directory structure as the root directory, which prevents attempting to decrypt a file with a dump. The attackers do not know where the file actually was, what the folder was.

A file system using a database is a solution to this problem, where the entire structure, content of the file, directory structure is stored in several database files. From the efficiency and speed of key-value systems to SQL archives, the building of file systems based on key-value storage, a program that was built on the basis of this approach is levelfs.

1.1 Our contributions

The main contributions of this research are summarized as follows:

- In this research, we propose a new architecture of encrypted file system based on a high performance distributed key-value store and Advanced Encryption Standard (AES). The product of this research is called KVEFS which is used in a linux distribution called mtaOS.
- This architecture is more flexible than existing solutions. Users can choose various underlining key-value store in the same host or from remote host.
- This research can help user to protect sensitive data in both local disk or remote machine. The directory structure is secured and hidden in key-value stores.

The rest of this paper is organized as follows: Section 2 presents basic knowledge to build our systems like libfuse, key value store and openssl. Section 3 introduces the study of fs encryption used in linux. The design and implementation of KVEFS on linux platform are shown in Section 4. In Section 5, we setup test-cases with different key-value store to evaluate KVEFS filesystem used in linux. Section 6 concludes the paper.

2 Backgrounds

In this section, we present three basic components of our encrypt file system: FUSE - a interface between application with kernel operating system, database key-value store where data is stored and retrieved, and encryption algorithm to encrypt/decrypt data.

2.1 Filesystem in Userspace

Filesystem in Userspace (FUSE) [4] is a framework of Linux operating systems. The FUSE provides an API library that lets non-privileged users create or access their file systems.

The FUSE module provides a "bridge" between the users and kernel interfaces. FUSE is available on many enviroments such as Android, Linux distribution, and macOS.

When a new file system is implemented, a handler program creates a linking to the FUSE library (called libfuse). This program determines how the state of file system responds when reading/writing/statistic is requested. The handler is also registered with kernel when the file system is mounted. If a user executes reading/writing/statistic requests for a mounted file system, the kernel forwards these IO requests to the handler and sends the handler's response back to the user.

Unmounting a FUSE-based file system with the fusermount command FUSE is particularly

useful for writing virtual file systems. The virtual file systems don't store data themselves. They are implemented as a transformation of an existing file system.

In principle, any resource available to a FUSE implementation can be exported as a file system.

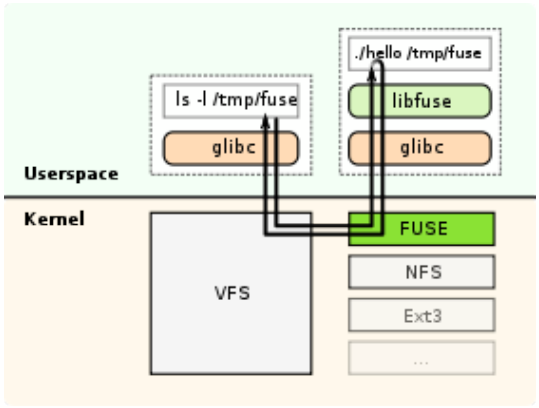

Fig. 1. The architecture of FUSE [2]

FUSE Architecture Figure 1 shows FUSE's high-level architecture.

When a user creates the "customfs" file system on the user space, the "customfs" file is compiled to a binary file. After that, it is mounted to $/tmp/fuse$ (illustration in the upper right-hand corner). If the user implements reading/writing that file, the Virtual file system (VFS) [1] forwards the request to FUSE's driver. This driver executes the request and responds back to the user. For example, the user performs a request: $ls\ -l\ /tmp/fuse$, this request gets by the kernel to the VFS through glibc library. The VFS then forwards the request fo the FUSE kernel. The FUSE contacts the binary file system corresponding "customfs" binary file. The binary file system responds back the results to FUSE, and finally back to the user through the VFS that originally made the request. However, Some file system can perform without communicating with the FUSE driver. For example, reads from a file whose pages are cached in the kernel page cache.

FUSE API implement FUSE API offers two APIs: a "high-level" synchronous API, and a "low-level" asynchronous API. With two APIs, the requests from the kernel are forwarded to the main program using callbacks. When using the high-level API, the callbacks may work with file names and paths instead of inodes, and processing of a request finishes when the callback function returns. When using the low-level API, the callbacks must work with inodes and responses must be sent explicitly using a separate set of API functions.

The high-level API that is primarily specified in fuse.h. The low-level API that is primarily documented in $fuse\_lowlevel.h$.

The callbacks are a set of functions that we wrote to implement the file operations, and a struct fuse_operations containing pointers to them:

```
struct fuse_operations {
int(*getattr)(const char *,struct stat *);
int(*mknod)(const char *, mode_t, dev_t);
int(*mkdir)(const char *, mode_t);
int(*rmdir)(const char *);
int(*rename)(const char *, const char *);
int(*chmod)(const char *, mode_t);
int(*open)(const char *,
struct fuse_file_info *);
int(*read)(const char *, char *,
size_t, off_t,struct fuse_file_info *);
int (*write)(const char *, const char *,
size_t, off_t,struct fuse_file_info *);
....
};
```

The fields of this structure are function pointers. Each one of them will be called by FUSE when a specific event happens on the file system. For instance, when the user writes on a file the function which is pointed by the field "write" in the structure will be called. To implement our file system, we need to use this structure and we need to define the functions of this structure then to fill the structure with the pointers of your implemented functions. Most of the functions here are optional; you don't need to implement them all.

2.2 Distributed Key-Value Stores

Key-value store is a type of nosql databases. We previously had deep researches on distributed high performance key-value stores and applications [10, 11] They have simple interface with only one two-column table. The *key* and the *value* are fiels of each record. The type of value is string/binary or structure, the type of key can be integer or string/binary. There are many implementation and design of key-value store including in-memory based and disk persistent[10]. In-memory based key-value store is often used for caching data; disk persistent key-value store is used for storing data permanently in file system[10].

In this research, we use the key-value store database for storing file information, data file and directory structure. The OpenStars database is a fast key-value storage library written by ThanhNT *et. al.* that provides multi key-value database such as LevelDB[9], RockDB[5], KyotoCabinet and ZDB [10]so on. We use it to build KVEFS.

OpenStars architecture OpenStars database use some abstract class to provide access specific class such as leveldb, rocksdb, ZDB[10]. *AbstractKVStorage* class is the base class, define function get(), put(), multiget(), multiput(), remove() to write or get data. Inherit from this class, specifics class implement these interfaces to handle data operations of key-value store. *AbstractCursor* class is base pointer, this pointer point to each record of database. *KVStorageFactory* class use to create the *AbstractKVStorage* based on specific option string.

Figure 2 shows that to create specific key-value database. We will use *KVStorageFactory* to create an instance of *AbstractKVStorage*, parameter input pass to constructor of *KVStorageFactory*

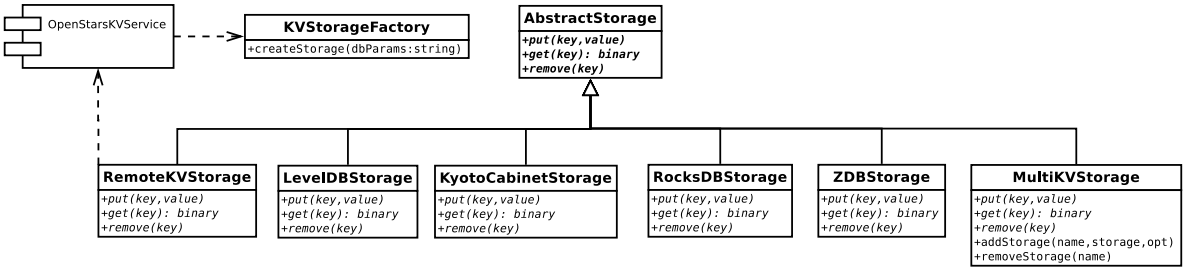

Fig. 2. The architecture of OpenStars database

is option string to specific type of database is used. The developers only do on *AbstractKVStorage* to handle with key-value database. The version of OpenStars Storage used in this paper support various types of key-value stores: leveldb , rockdb , kyoto cabinet, bigset[11], ZDB [10]. These key-value store can be used simultaneously by using *MultiKVStorage*. Key-value store can be served in a remote backend service. We can read and write data remotely by using *RemoteKVStorage*. These classes are shown in Figure 2

OpenStars api implement We can use type of key-value database by passing on the configuration string to initialize the specified database.

```
AbstractKVStorage* database =
factory.createStorage(configString,
name,rwmode);
```

After initializing the database, we can read and write data by calling the put and get functions as follows:

```
string sVal;
string sKey;
// read data from database
database->get(sVal,sKey);
// write data to database
database->get(sVal,sKey);
```

2.3 Encryption algorithm

Encryption algorithms are widely used in data protection, communication over network.,etc. In this research, we use them to encrypt data before storing and decrypt the retrieved data from underlining key-value store.

The encryption algorithm and the amount of security needed will decide the type and length of the keys. In popular symetric cryptographic algorithms, both encryption phase and decryption phase use the same key while asymetric encryptions use different keys in those phases.

In this paper, we use symmetric encryption because of performance and usability issues, asymmetric encryption is often used in communication. In our system, there is no need to transmit the key to outside receiver because the objectives are protect user data in local disk

without transfering via network. Therefore, the use of symmetric encryption is safe enough for the system. We use AES[13, 3] encryption algorithm. AES is based on a design principle known as substitution–permutation network, and is fast in both software and hardware.

3 Related work

3.1 EncFS

EncFS [7] is also a FUSE-based cryptographic filesystem like our work. EncFS encrypts file in local disk directly. It supports various encryption algorithms, key-size and security levels. However, due to its design, It lacks ability to distribute data to nodes in network and its folder structure can be shown to every user.

3.2 LevelFS

LevelFS [14] is a FUSE-based file system backed by LevelDB[6]. It implements a filesystem where data are stored in LevelDB key-value store. File paths, directories are organized in the *keys* of LevelDB, and the *values* store file contents.

It transparent stores and retrieves data from leveldb key-value store. However, this simple filesystem does not support encryption, so data is not protected safely.

3.3 eCryptfs

eCryptfs [8] is a EFS implemented as a stand-alone kernel module of Linux. Every file in eCryptfs has a metadata in the header which store cryptographic information. They can be copied to another machine and can be decrypted using proper keys. This filesystem does not support distributing data to outside node in the network.

4 Proposed Methodology

In this section we propose a method for building an encrypted file system using the libfuse, openssl, and openstars libraries, called KVEFS.

4.1 KVEFS architecture

Figure 3 shows the architecture of KVEFS. We have divided our system into 4 layers: The *GUI and Parameter Input Layer* is the graphical user interface of the system to communicate with the user for ease of use and acquire parameters of key-value store and encryption options; the $FuseLayer$ is the main layer of the system to communicate with the kernel of the operating system to manipulate files and folders; the $CryptographyLayer$ is for encryption and decryption data when read or write from a key-value stores; the $Key - ValueLayer$ for storing encrypted information and data about file and folder structure. The details about these layer are presented below.

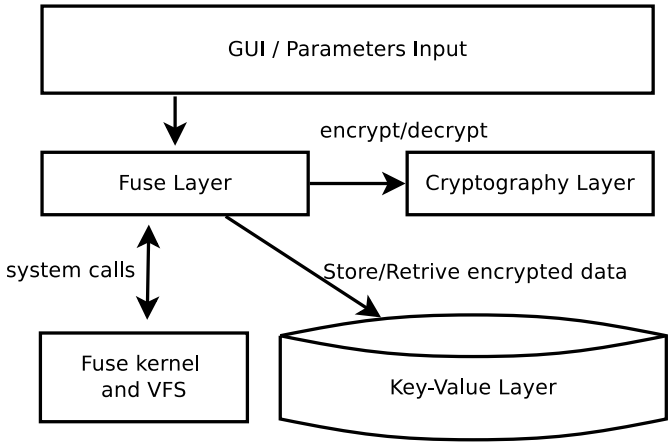

Fig. 3. The architecture of KVEFS

Key-Value Layer We use key-value store in this layer for storing filesystem metadata and data of files. The key column stores information about the file name of the file or directory path, the value column of the file contents or the items contained in the directory. In our implementation, we use OpenStars library with dynamically choosing key-value stores. This layer can be customized to use variety type of key-value stores.

Cryptography Layer In order for data to be secure and not readable by another system, we need a module responsible for encrypting and decrypting data, which encrypts the data before the data is saved to the database. To do this, we used the AES encryption algorithm in the openssl library. All keys and values in *Key-Value Layer* are encrypted before storing into key-value store.

Fuse Layer This layer is the core of the research, which is responsible for communicating with the operating system kernel to perform operations on files and directories, such as opening an open file, writing to an executable file write, create directory (mkdir), and so on. We used the libfuse library to do this work. This layer also communicate with *Cryptography Layer* to encrypt and decrypt data when writing and reading.

GUI and Parameter Input Layer Command-line interface is difficult to use for people who have little knowledge of linux, so creating a simple, easy-to-use interface is essential for users to see how easy it is to use. The GUI module is responsible for building a simple, easy-to-use GUI, using the Qt Framework to build the GUI. The mounting file system, unmounting file system are implemented here.

4.2 How KVEFS work

When creating a new File System using the GUI and Parameter Input Layer, the system initializes the functions in the FUSE Layer. Functions in the Fuse Layer are required to have capability that get attribute files or directory (getattr), read all items in a directory (readdir),

read files (read), write files (write), and more, including directory creation (mkdir), directory deletion (rmdir), file deletion (unlink), etc. Functions are initialized after the fuse_main () function is run, after fuse_main () is launched, an infinite loop is created to satisfy any real-time operation of the user. That means functions that take file attributes or read files, write files are invoked repeatedly whenever a user interacts with a file system.

```
int main(int argc, char **argv)
{
  struct fuse_args args =
     FUSE_ARGS_INIT(argc, argv);
  memset(&conf, 0, sizeof(conf_t));
  fuse_opt_parse(&args, &conf, opts,
   opt_parse);
  return fuse_main(args.argc,
     args.argv, &KVEFS_oper, NULL);
}
```

Our objective is to perform the file manipulation functions through libfuse, libfuse using the callback mechanism and the existing prototypes. All encryption and decryption operations are implemented in these callback function. The functions in libfuse have one thing in common: they have a file or directory path parameter, so we have designed the information stored in the database key column as the path name of the file or directory, and in the value column, we store the contents of the file or the file list and the subdirectory contained in the directory. Both the key and the value are encrypted before storing to key-value store. So we can protect the directory structure efficiently. To distinguish a file from a directory, we use a prefix before the path name, the FILE prefix is added if the path points to the file, the DIR prefix is added if the path points to the directory. The following listing shows how to perform the read and write functions:

```
int KVEFS_read(const char *path, char *buf,
size_t size, off_t offset,
struct fuse_file_info *fi)
{
  key = path_to_key(path, &keylen, 0);
  val = db_get(CTX_DB, key, keylen,
      &vallen, &err);
  memcpy(buf, val+offset, size);
}

int KVEFS_write(const char *path,
  const char *buf,
  size_t bufsize,
  off_t offset,
  struct fuse_file_info *fi) {
    key = path_to_key(path, &klen, 0);
    val = db_get(CTX_DB, key, klen,
      &vlen, &err);
    ....
```

```
    memcpy(val, buf, bufsize);
    db_put(CTX_DB, key, klen,
      val, val_len, &err);
}
```

In the read function, we find the contents of the file through their path, because the path is their key, the value is the content of the file, the data is read from the database and decoded through. The encoding scheme, from which it is displayed to the user, reads the contents of that, in the write function, we encrypt the data before storing it into the database.

```
void db_put(db_t *db, const char *key,
size_t klen,
const char *val, size_t vlen) {
...
//encrypt
char *ciphertxt;
ciphertxt = (char*)calloc(vlen,sizeof(char));
encrypt_stream(val,ciphertxt,vlen);
db->put(db->db, opts, key,
klen,ciphertxt, vlen);
....
}

char *db_get(db_t *db, const char *key,
  size_t klen,size_t *vlen,) {
    ....
  val = db->get(db->db, opts, key,
  klen, vlen, errptr);
  //decrpyt ;
  char *plaintxt =
       (char*)calloc(   *vlen,sizeof(char));
  decrypt_stream(val,plaintxt,*vlen);
  ...
  return plaintxt;
}
```

4.3 Key management

As presented above, this research uses AES algorithms to encrypt and decrypt data. An important question is "Which key is used to encrypt and decrypt? ". At the initial of the session, we generate a random keys for AES.

$$aesDataKey = generateRandomKey() \tag{1}$$

User must enter a password, and we use a key derivation function such as [12] to genreate a key called $tempAESKey$from password.

$$tempAESKey = kdf(userPassword) \tag{2}$$

we use encrypt $aesDataKey$ using $tempAESKey$

$$eeKey = aesEncrypt(aesDataKey, tempAESKey) \tag{3}$$

We store $eeKey$ to key-value store with a special key. Before mounting the file system user must enter password to decrpyt the $aesDataKey$ from stored $eeKey$. To change password, user firstly decrypt and get $aesDataKey$ using old password, and encrypt it again using new password and finally store new $eeKey$ key to key-value store.

5 Evaluation

In this section, we will compare underlining key-value stores used in this research: LevelDB, RockDB , KyotoCabinet of OpenStars Storage for our file system and a popular encrypt file system is EncFS about read/write speed.

5.1 Data and environment

We used two types of data : 1000MB document file. Each file has size less than 1MB, and 1000MB media file , each file is lager than 5mb.

We used environment linux os on laptop dell inspiron 5547 intel core i7, 8GB ram, hard disk HDD 1TB.

Three encrypt file systems for our experiment: KVEFS with Storage LevelDb, KVEFS with Storage RocksDb, and KVEFS with Kyoto Cabinet. The following criteria we use for testing:

- Write 1000MB file documents/ media
- Extract the 100MB document/media linux-3.0.tar.gz archive
- Recursively delete the extracted files

5.2 Performance Comparison

Table 1. Read/write speed document files statistics

KVEFS use type Storage Database	Stream write	Extract	Delete
Our KVEFS /w LevelDB	1002KB/s	77 s	5 s
Our KVEFS /w RockDB	1024KB/s	74 s	4 s
Our KVEFS /w Kyoto Cabinet	1201KB/s	53 s	3 s

Table 2. Read/write speed media files statistics

KVEFS use type Storage Database	Stream write	Extract	Delete
Our KVEFS /w LevelDB	2001KB/s	44 s	3.4 s
Our KVEFS /w RockDB	2203KB/s	42 s	3.4 s
Our KVEFS /w Kyoto Cabinet	3005KB/s	32 s	2.2 s

LevelDB and RockDB are same rate. RockDB is essentially a database developed from LevelDB so it's easy to see that. Kyoto Cabinet has faster read and write speeds. The results are shown in Table 1 and Table 2. In practice, EKVEFS is usable for sensitive data storing which is transparently protect data for end-user in key-value stores.

5.3 Theoretically comparison and security level estimation

In KVEFS, we use AES 256 bit for strong data protection. Due to using OpenStars key-value store library, we can dynamically choose various key-value store and data can be distributed with RemoteKVStorage. Encfs can only store data in local file system of operating system. The different is shown in Table 3.

Table 3. Theoretically comparison between KVEFS and Encfs

Capability	KVEFS	EncFS
Security Level	AES 256 bit	AES, Blowfish
Customizable storage	Yes, user can choose storage type	No, write to files directly
Distributed ability	Yes, with RemoteKVStorage	No, data must be fit in a local host
Hide directory structure	Yes, hide all directory structure in single Key-Value store	No, you can see numbers item in a folder and accessed-time

6 Conclusion

This paper introduces a new architecture for building encrypted file system using FUSE, high performance distributed key-value store and the AES encryption algorithm. We also show you how to manage the key by saving the hash directly to the database, changing the password without having to re-encrypt the entire data to avoid I/O overhead time. With KVEFS, end-users can have new choice of encrypted file system to protect sensitive data, the directory structure is secured and hidden when storing in key-value store. KVEFS can be a component of a secure operating system. In our future research, we will continue to develop this idea for file encryption in cloud storage, integrate with more block cipher algorithms, improve performance and try to optimize for big files.

References

1. MICHAEL ABD-EL-MALEK. File system virtual appliances. *Dept. of Electrical and Computer Engineering Carnegie Mellon University*, 2010.
2. Erez Zadok Bharath Kumar Reddy Vangoor, Vasily Tarasov. To fuse or not to fuse: Performance of user-space file systems. *USENIX Conference on File and Storage Technologies*, 2017.
3. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
4. SZMI et al. File system in user space. *URL: https://github.com/libfuse/libfuse*, 2011.

5. Siyu Chen Mengwei Hou Jeong-Uk Kang Sangyeun Cho Fei Yang, Kun Dou. Optimizing nosql db on flash: A case study of rocksdb. *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*.
6. Sanjay Ghemawat and Jeff Dean. Leveldb. *URL: https://github. com/google/leveldb,% 20http://leveldb. org*, 2011.
7. Valient Gough. Encfs encrypted filesystem. *Located at: https://github.com /vgough/encfs*, 13:22, 2003.
8. Michael Austin Halcrow. ecryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the 2005 Linux Symposium*, volume 1, pages 201–218, 2005.
9. Yulong Zhao Dingzeyu Wu Chengrui He Lei Wang, Guiqiang Ding. Optimization of leveldb by separating key and value. *2017 18th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2017.
10. Thanh Trung Nguyen and Minh Hieu Nguyen. Zing database: high-performance key-value store for large-scale storage service. *Vietnam Journal of Computer Science*, 2(1):13–23, Feb 2015.
11. Thanh Trung Nguyen and Minh Hieu Nguyen. Forest of distributed b+ tree based on key-value store for big-set problem. In *International Conference on Database Systems for Advanced Applications*, pages 268–282. Springer, 2016.
12. Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. Technical report, 2016.
13. NIST FIPS Pub. 197: Advanced encryption standard (aes). *Federal information processing standards publication*, 197(441):0311, 2001.
14. Zhi Hao Wu. A log-structured file system based on leveldb. In *Applied Mechanics and Materials*, volume 602, pages 3481–3484. Trans Tech Publ, 2014.