A Privacy-Protecting File System on Public Cloud Storage

Zhonghua Sheng Zhiqiang Ma Lin Gu Ang Li
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Email: {szh,zma,lingu,laxab}@cse.ust.hk

***Abstract*—With the development of cloud-based systems and applications, a number of major technical firms have started to provide public cloud storage services, and store user data in datacenters strategically positioned across the Internet. However, when users store private data in shared datacenters, they lose control over how the data are stored and accessed. Multiple classes of personnel may access the physical storage media and potentially read the data. While strong cryptographic methods can protect user files from unauthorized accesses, they incur computational overhead, and make it difficult for the infrastructure provider to optimize the storage space with effective compression and deduplication. To provide strong protection on user data, we design a new file system called BIFS (Bit-Interleaving File System). Focusing on the privacy protection of the on-disk state, BIFS re-orders data in user files at the bit level, and stores bit slices at distributed locations in the storage system. While providing strong privacy protection, BIFS still retains part of the regularity in user data, and thus enables the infrastructure provider to perform a certain level of space optimization (e.g., compression). We implement BIFS on the Amazon Simple Storage Service (S3), and examine its performance characteristics. The comparison with several existing network or Internet-based file systems shows that BIFS provides robust file system functions with satisfactory throughput on S3.**

I. Introduction

In recent years, cloud-based services and applications have emerged to be a new computing paradigm and lead to the establishment of global data storage and computation platforms. However, one major challenge is to construct a well-defined technical solution to store private user data on a shared and uncontrolled infrastructure. When end users release their data to remote datacenters, they lose control over the data after the bits leave their client computers. Existing scalable storage solutions, such as GFS [1], Dynamo [2], Bigtable [3], and PNUTS [4], scale up the data storage capacity and throughput, but have largely left privacy protection as a non-goal or future work. We list some potential privacy hazards below.

First, user data are exposed to operators who have access to the media in the datacenter, programmers who develop code with the low-level block interface, and third-party service staff who repair the hard drives when they are broken [5], [6]. Second, the storage media can be lost or stolen, and the storage server may be compromised [7]. Finally, many legitimate accesses to the physical media, such as debugging activities, may reveal and manipulate user data in a way that is in conflict with the end users' privacy concern. Unfortunately, the end user may not even be aware of the existence of such debugging activities.

The characteristics of sharing obscures the boundary of ownership, complicates data management, and makes traditional solutions, including encryption and access control, ineffective and undesirable for large online storage infrastructure. Strong encryption, for example, could be used in traditional enterprise environments to protect proprietary digital assets, but imposes noticeable overhead on the storage system [8], [9]. Zadok et al. shows that the encryption overhead may reach 22.7% [10]. In an investigation using eCryptfs [11] on modern compute servers, we observe an overhead of about 30% when enabling an encryption layer above the ext4 [12] file system.

Moreover, compression and deduplication have become increasingly important in very-large-scale storage systems [13]. However, these features work poorly on encrypted data [14]. Though it is possible to use file-specific keys to generate identical ciphertexts (convergent encryption) [15], [16], such approaches incur significant space overhead, and are not applicable to data with minor differences. Overall, traditional techniques effectively used in personal and enterprise computing environments, such as encryption, discretionary access control and obfuscation, fall short of providing an adequate solution to privacy protection in large-scale cloud storage systems.

To overcome these challenges, we design a privacy-preserving file system to protect the on-disk state of user data stored in cloud-based storage services. The file system service runs on end users' client computers, but stores data on public online storage services, such as Amazon S3 [17]. Although the on-disk state is materialized in a shared storage infrastructure that the end user has no control over, our design makes it practically infeasible for unauthorized users, including the system administrators, to identify user data on the storage media. Yet the storage system can still perform a certain level of compression on the user data bits. Departing from the traditional file system designs, the new file system, called BIFS, re-orders data bits in a way that retains certain regularity in the file data, and stores them at distributed locations. The regularity prevents a dramatic increase in the entropy of the data, and enables compression and deduplication operations. In the mean time, the bit-interleaving mechanism ensures that it is very difficult for unauthorized users to take advantage of the regularity to recover user files and directories.

The BIFS design abandons the long-time wisdom of main-

978-1-4577-1637-9/11/$26.00 ©2011 IEEE

taining data locality on the physical media. We believe that data locality is a less important determinant of file system performance in the cloud-oriented design context, and that efficient designs can mitigate a significant part of the overhead. We implement BIFS on the widely used Amazon S3 (Simple Storage Service) [17], and compare its performance with several existing network file systems and Internet-based file systems. The evaluation shows that BIFS is sufficiently responsive for interactive users, and provides a higher I/O throughput than some network file systems that do maintain data locality.

We highlight several contributions of BIFS as follows.

1) To the best of our knowledge, BIFS is the first file system that performs aggressive randomized bit-level reordering for privacy protection and achieves the required strength without bit substitution or full encryption.
2) Protecting the on-disk privacy, the BIFS provides a solution for users to store proprietary, sensitive, or non-public data on public cloud storage controlled by third-party vendors.
3) The design and evaluation of BIFS show that we can "hide" user data so that unauthorized users cannot identify them, yet the storage system can still perform a certain level of compression.
4) We implemented BIFS on Amazon S3 and show that the bit-interleaving design is efficient enough to provide a robust file system service with decent performance.

The rest of the paper is organized as follows. Section II proposes the basic design principles of BIFS. Section III presents the design of BIFS, followed by Section IV discussing implementation details. Section V evaluates the performance of BIFS. Section VI surveys related work. Finally, Section VII concludes the paper.

II. Approach

To illustrate the challenges in protecting data privacy in a cloud environment, we first study the trust relationships among players in a cloud application system. There are three major entities in a cloud-based application system: the *infrastructure provider*, the *application provider*, and the *end user*, which are roughly equivalent to the *Cloud Provider*, *SaaS Provider* and *SaaS User* described by Armbrust et al. [18]. Figure 1 illustrates the roles of these entities in a cloud-based system. Both the application and the *end user* require access to user data, which is stored in the shared datacenter infrastructure.

The *end user* must trust the *application provider*, otherwise the user would not elect to use the application in the first place. However, the *end user* can neither control nor trust the shared infrastructure. While a few aspects of the interaction between the *application provider* and the *infrastructure provider* can be bound by commercial terms, it is essentially impossible for the *infrastructure provider* to agree and guarantee that all accesses to the storage media are conducted in a way that eliminate privacy hazards. Though it is possible that, in some cases, the application provider may not be fully trusted, the more

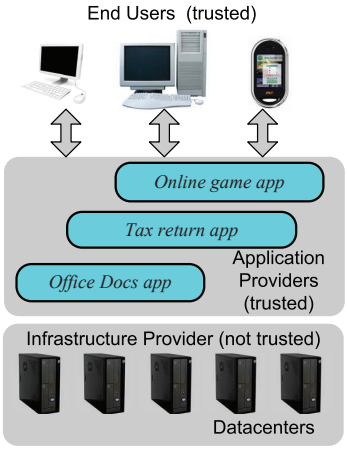

Fig. 1. Participants of an online application system. The *infrastructure provider* is the operator of a large storage and computation infrastructure, including datacenters, replication sites, and network facilities. The *application provider* develops application services on the shared infrastructure. The *end user* is the customer who use the application service.

critical applications, such as accounting, tax computation, e-business, are more often handled by well-known firms and regulated by commercial or even legal terms. Moreover, the end users are able to build their own application with full trust, but seldom have the opportunity to control the cloud computing infrastructure.

It is a challenge to ensure data privacy on an infrastructure that the user cannot control or trust. Our approach is to treat the infrastructure as a "dumb" bit storage device, and store data bits in a form that is practically illegible to the *infrastructure provider*. This leads to our first design principle:

Let the user handle data, and the infrastructure handle bits.

It may appear that we could use encryption to achieve this goal. However, encryption introduces much overhead to the system, and deprives opportunities for compression and deduplication (refer to Section V-C for an empirical investigation). Unlike the encryption algorithm which shuffles and mutates user data, BIFS mainly re-orders (transposes) bits to remove the semantic correlation that could be used by unauthorized users to recover the original data. This reflects the second principle BIFS follows:

Hide data by re-ordering, not substitution.

This approach not only reduces computational overhead, but also preserves some structural regularity in user data to facilitate compression and deduplication.

However, without substituting data bits, an unauthorized user may attempt to permute the bits to recover the original data. BIFS's bit re-ordering process makes sure such decoding is very difficult. Moreover, BIFS strengthens the protection against decoding attempts by storing re-ordered bits in distributed locations in the storage system. This is, in fact, the third principle in the BIFS design:

Improve strength by distribution, not entropy.

This approach can provide very strong protection for user data—if an unauthorized user cannot collect the bits belonging to a file, it is generally impossible to decode the data no matter

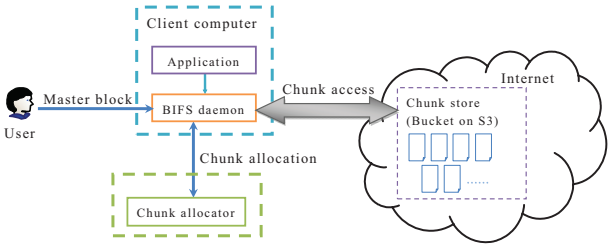

Fig. 2. The organization of BIFS

how much computational power the unauthorized user has.

Following these principles, BIFS presents a file system abstraction above the online storage infrastructure. By distributing data bits into different locations of the storage infrastructure, it ensures that private user data cannot be identified, in practice, by unauthorized users even if all bits and sectors on the storage media are visible. In addition, we require that the BIFS provide a robust file system service with decent performance in the wide-area network (WAN) environment.

It is worth clarifying that BIFS focuses on protecting the privacy of on-disk state. There are other types of possible privacy infringements, e.g., directly peeking the CPU or memory state, eavesdropping the network to record the collection of data bits a user reads or writes, or compromising the end user's client computer. Many of such threats are also present in traditional computing paradigms with solutions proposed and tested. While it is potential future work to port these solutions (e.g. secure communication [19], [20], trusted instruction execution [21], [22]) to the cloud environment, this work focuses on the more challenging problem of protecting the privacy of data stored on the physical media controlled by the *infrastructure provider*.

III. Design

BIFS consists of three components—the *BIFS daemon*, the *chunk store*, and the *chunk allocator*, as shown in Figure 2. The three components run in different control domains, and reflect the trust relationship described in Section II.

The *BIFS daemon* runs on the client computer. Hence, the user can control, verify, and trust the *BIFS daemon*. It implements the file system abstraction, and transforms the file system operations to accesses to the chunks in the *chunk store*. When using the BIFS, the user provides the *BIFS daemon* with a 128-byte "*master block*", which contains several parameters including the user's credential (e.g. a password or a key). The user can update the credential when it is necessary, and should follow known good practices to protect the *master block* (e.g. making backups and storing them safely). The BIFS daemon uses information in the master block to locate user data and perform file system operations.

The *chunk store* can be any addressable online storage services. Our current implementation uses Amazon S3, but the design can be easily extended to the blob storage [23], the table based entity store [4], [24], and distributed hashing storage [2], [25]. More often than not, the *chunk store* is

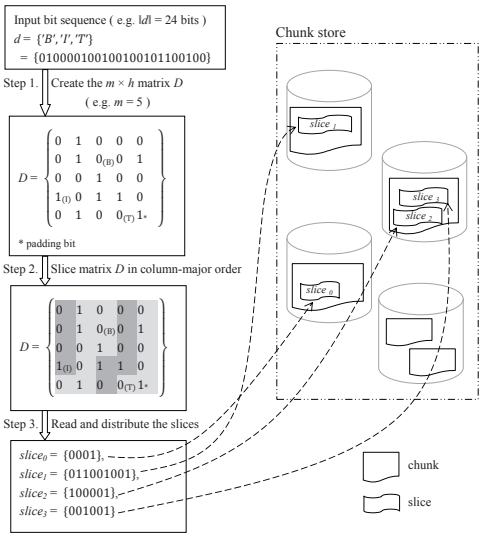

Fig. 3. An example of the bit-interleaving process

controlled and managed by an *infrastructure provider*, which is not fully trusted as described in Section II.

The *chunk allocator* records the allocation information of all chunks in a *chunk store* for the purpose of chunk allocation and deallocation. The *chunk allocator* is actually a component for optimizing system performance, since crash or demise of it does not result in data loss in BIFS. The user can elect to run a *chunk allocator* on the client computer for maximum control, or on a remote server for reducing client-side complexity. Note that, except for chunk allocation and deallocation, the BIFS daemon communicates directly with the *chunk store* for reading and writing data without involving the *chunk allocator*.

Independent of how chunks are stored, striped, or replicated, BIFS should ensure effective file system functions with strong privacy protection. Specifically, we design BIFS to meet the following requirements.

- A user holding the *master block* can use the file system efficiently with reasonable performance.
- Individual files on the physical media are practically illegible to unauthorized users including "insiders" without the *master block*, unless the unauthorized users already have a complete copy of the files.
- Files are practically unrecoverable from the physical media after being deleted.

This privacy protection in BIFS cannot rely on strong encryption. Instead, it is achieved by the bit-interleaving process, which is presented in detail in Section III-A.

A. Bit-interleaving

BIFS treats a datum in the file system (file, directory, metadata block, etc.) as a bit sequence d with length $\|d\|$. We first append $m \times h - \|d\|$ padding bits at the end of d to form a new bit sequence d^* with length $m \times h$. Here m is a system parameter, and $h = \lceil \frac{\|d\|}{m} \rceil$. We organize the bits in d^* to be an $m \times h$ matrix D in row-major order. Then we

serialize the bits in D in column-major order to obtain a new bit sequence d'. Hence, $d'[i] = d^*[(i \ mod \ m) \times h + \lfloor \frac{i}{m} \rfloor]$, where $d^*[i]$ and $d'[i]$ are the i^{th} bit of d^* and d', respectively $(0 \leq i < m \times h)$. d' is further divided into variable-length bit slices with lengths chosen between the minimum slice length (l_{min}) and the maximum slice length (l_{max}). The slice cutting and distributing process will be discussed later in Section III-B. We call this bit re-ordering and slicing process bit-interleaving, and call d' the bit-interleaved sequence. The parameters m, l_{min} and l_{max} shall be chosen to avoid values that could weaken the privacy protection strength, which is to be discussed in further detail later.

Figure 3 shows an example of the bit-interleaving process. The input bit sequence d is first transformed into a matrix D with 5 columns and 5 rows with one padding bit appended. Scanning D in column-major order, the bit-interleaving process cuts d' into 4 variable-size bit slices, and distributes them into 3 chunks in the chunk store.

It is obvious that there would be a potential risk if m is not chosen properly. Specifically, in the case that the bit sequence is a serial of ASCII characters with $\|d\| \geq 64$ and m is 8, the $slice_0$ contains all '0'. An unauthorized user may reveal the user data using brute-force methods by speculating that $slice_0$ is a collection of the most significant bits of ASCII characters. Such brute-force attacks and speculation become extremely difficult when the number of slices is large, and the design details of BIFS makes such attacks practically infeasible. Nevertheless, we shall choose system parameters to avoid predictable boundary alignment. We require that no slice contains two bits from the same byte in d. Hence, we require $m \geq 8$, and $l_{max} \leq \frac{\|d\|}{m}$, and m should not be a multiples of 8.

Furthermore, BIFS mixes slices from different files when storing them on chunks. Because slice lengths are not fixed, it is extremely difficult for an unauthorized user to identify the boundaries of the slices and retrieve them for matching and analysis.

B. On-disk data organization

The bit-interleaving process disassembles continuous data bits, creates variable-size slices, and stores data in distributed locations. Traditional I/O optimization techniques can hardly apply in this new data organization scheme. Hence, it is very important to design the on-disk data organization so that the aggressive re-ordering does not drastically increase the overhead in storage space and I/O performance.

In BIFS, there are three levels of storage units: files, blocks and slices as shown in Figure 4. Each file is divided into a number of fixed-size blocks, which are processed by the bit-interleaving process and stored as a number of variable-size bit slices on chunks distributed in the system.

User files and their metadata are stored in a number of chunks. Figure 5 depicts the layout of a chunk. One chunk is a fixed-size sequence of bits, and each chunk logically belongs to one user. A chunk consists of three parts—*Current Slice ID* (*CSID*), *Slice Mapping Table* (*SMT*) and a *slice area*

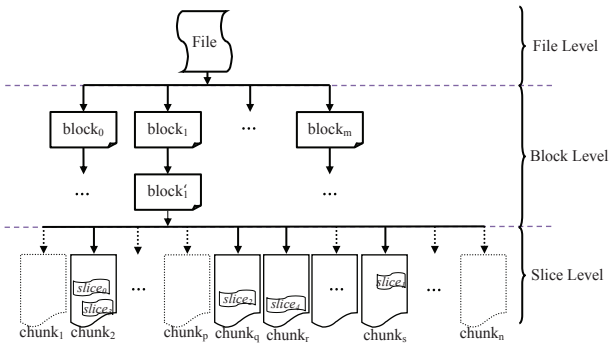

Fig. 4. Data organization in the BIFS file system

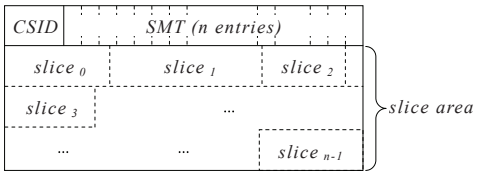

Fig. 5. Structure of a chunk

for storing bit slices. The CSID stores a 64-bit integer used for the slice allocation, and the SMT is a vector with each entry corresponding to a physical slice. We will introduce the functions of the CSID and SMT in detail later with the slice allocation algorithm.

In BIFS, a block is composed of slices stored on multiple chunks in the slice areas. The slice area of a chunk is partitioned into n physical slices—$p\_slice_{c,k}$, where c is the chunk ID and k represents the physical slice number $(0 \leq k < n)$. These physical slices are of variable sizes and their lengths are calculated with a function g with three parameters—the user's credential (r), the chunk ID (c), and the physical slice number (k), i.e., $\|p\_slice_{c,k}\| = g(r, c, k)$. The function g is an irreversible hash function [26], [27] with the weak collision resistance property. The starting address of a physical slice, $p\_slice_{c,k}$, in the slice area is the sum of the lengths of all the physical slices before it $(\sum_{i=0}^{k-1} \|p\_slice_{c,i}\|)$. Hence, a physical slice number uniquely identifies the slice's physical position in the chunk. In a chunk c containing n physical slices, $p\_slice_{c,0}$ and $p\_slice_{c,n-1}$ are filled with random bits, in order not to reveal the boundary of the effective data area in a chunk.

To store and retrieve a block, we need an efficient mechanism to record the position and order of each slice of the block. The intuitive approach of recording the ⟨chunk ID, physical slice number⟩ for all the slices constituting a block incurs significant storage overhead. We introduce a space-efficient approach using fixed-size *block descriptors* in the next section.

C. Space efficiency on recording slices

To enhance the strength of privacy protection, the BIFS mixes the slices from different blocks and stores them in a permutated order on a chunk. To facilitate the slice re-ordering,

the BIFS uses a logical slice ID, instead of the physical slice number, to address a slice. We record the mapping from a logical slice ID to a physical slice number in the SMT. Each physical slice has a corresponding entry in the SMT, and the logical slice ID mapped to this physical slice is recorded in the entry. Hence, we can use ⟨chunk ID, logical slice ID⟩ as the slice reference. Though the mapping from the logical slice ID to the physical slice number is stored in SMT as plaintext, it does not harm the privacy protect strength of BIFS because the physical slice number of a slice does not provide information to determine the position and length of the slice. In the remainder of the paper, we refer to the logical slice ID as the slice ID.

To facilitate the allocation of slice IDs, we record a CSID on each chunk. The CSID reflects the largest slice ID currently used in this chunk. It starts from 1 upon chunk initialization, increases monotonically, and may grow beyond the number of all physical slices in the chunk. When allocating a new slice on a chunk, BIFS allocates a vacant physical slice on the chunk, assigns the CSID on that chunk to the new slice as its slice ID, records the slice ID in the SMT entry corresponding to the physical slice, and increases the CSID by one. De-allocating slices does not decrease CSID, and a chunk is considered full when all the physical slices on the chunk are allocated. This technique, in fact, also optimizes the space efficiency of the BIFS upon file deletes.

It is obvious that the slice IDs of the successively allocated slices on a chunk form a sequence of contiguous numbers starting from the slice ID of the first allocated slice. This slice ID of the first slice for the block is called *FSID* and it is always equal to the CSID of that chunk before the BIFS allocates the slices. Hence, we may use the ⟨chunk ID, FSID⟩ to represent all the slices allocated on a chunk for a block. Therefore, the information we need to store for retrieving a block is a fixed-size vector of M elements, each element being a ⟨chunk ID, FSID⟩ pair. In BIFS, we call such a vector a *block descriptor* (denoted as *blk_desc*). Thus, the order and position of bits in a block are unambiguously determined by a block descriptor. Using block descriptors can reduce 70% space overhead compared to the intuitive approach.

To summarize the on-disk data organization, we list the procedure of writing a new file as follows: 1) cut the file into a number of fixed-size blocks; 2) apply the bit-interleaving process to obtain variable-length slices for each block; 3) store each block's slices in M randomly chosen chunks; 4) save the block descriptor for each block.

D. Metadata organization

As described in Section III-B, BIFS uses block descriptors to uniquely specify a block. As a sequence of block descriptors together with a number representing the file size can unambiguously define the content of a file, BIFS stores the sequence of block descriptors, the file size, and the other attributes for a file. BIFS defines a special type of blocks called the *index block*. In each index block, there are multiple entries for block descriptors, which can store the block descriptor of either a data block or a next-level index block. The index blocks are organized in a tree structure and the index block at the leaf of the tree stores the block descriptors for data blocks. In this way, all the block descriptors for the data blocks of a file can be retrieved when the root index block is provided.

Algorithm 1 Slice allocation

Input: B': a bit-interleaved file block
Output: *blk_desc*: a *block descriptor*

1: Randomly pick up M chunk, denoted as $(C_0, C_1, \ldots, C_{M-1})$;
2: **for** $i = 0$ to $M - 1$ **do**
3: *blk_desc.chunk_id*$[i] = C_i$;
4: *blk_desc.FSID*$[i] = CSID_{C_i}$;
// $CSID_{C_i}$ denotes the CSID of chunk C_i
5: **end for**
6: *offset*=0;
7: $s = 0$; // slice order
8: **while** *offset*$< \|B'\|$ **do**
9: Allocate a chunk for the new slice. c_s = *f(r, blk_desc, s)*, where $c_s \in \{C_0, C_1, \ldots, C_{M-1}\}$;
10: Randomly pick up a physical slice ($p\_slice_{c_s,m}$) on chunk c_s, and store $CSID_{c_s}$ in the entry for $p\_slice_{c_s,m}$ in the SMT;
11: Calculate the length of the physical slice, *slice_len*$_{c_s,m}$ = *g(r, c_s, m)*;
12: Calculate the offset of the physical slice, *slice_offset*$_{c_s,m} = \sum_{j=0}^{m-1}$ *slice_len* $_{c_s,j}$;
13: Copy *slice_len*$_{c_s,m}$ bits from *offset* in B' to chunk c_s starting from *slice_offset*$_{c_s,m}$;
14: $CSID_{c_s}++$;
15: $s++$;
16: *offset*+ =*slice_len*$_{c_s,m}$;
17: **end while**

To store file attributes, another type of descriptors is added to the system—*file descriptor*, which stores file name, file attributes and the block descriptor of the root index block. Naturally, directories can be implemented as special files containing a number of file descriptors, each file descriptor referring to a sub-directory or a file in the directory. Suffices for it, hence, to store only one file descriptor (the master block)—the file descriptor of the root directory—and protect the file descriptor with measures that the user considers necessary and sufficient, in order for a user to access the entire file system.

E. Resistance to privacy infringements

The BIFS stores a block in two steps:
1) Shuffle the data bits of the block with the bit-interleaving process and cut the shuffled bit sequence into slices.
2) Use the slice allocation algorithm to distribute slices into M randomly selected chunks.

To analyze the resistance of BIFS to privacy infringements, we first look into the brute-force attack. We assume that the adversary has some knowledge of the file system—it is possible that the system configurations are exposed to the adversary so that the adversary knows the bit-interleaving interval m, minimum slice length l_{min}, and maximum slice length l_{max}. We can find the lower bound of the number of slices in one block and one chunk are $B = \lceil \frac{\|b\|}{l_{max}} \rceil$ and $S = \lceil \frac{K}{l_{max}} \rceil$ respectively, where K is the size of a slice area

in bit. We can further assume that all the slices can be retrieved from the chunks though it is, in practice, extremely difficult for adversaries to obtain the slices mixed with each other in the chunks. Even if we do not consider the slice boundary obfuscation on chunks, the lower bound of the number of possible combinations that a brute-force attack must consider is $C_Q^M \times A_{S\times M}^B$, where the slices in one block is distributed into M chunks, and the total number of chunks in the file system is Q.

Suppose we store 8 KB blocks with l_{min} = 4096 bits and l_{max} = 5120 bits in 4 MB chunks (K = 32 Mb). Conservatively, we assume that we have 1 GB storage space, totally 256 chunks ($Q = 256$), and that each block is distributed into 4 chunks ($M = 4$). The number of the possible slice sequences of a single block is larger than 2^{218}, which is not practically feasible for current conceivable computational capability.

The analysis shows strong strength of privacy protection under brute-force attacks even if the adversary is able to obtain all the chunks. In reality, it is highly unlikely that an adversary can access all the storage media (e.g. hard drives) in a datacenter. Even an insider can only access part of the storage system in usual situations. A further extension is that we distribute the chunks among multiple clouds to make it essentially impossible for the adversary to collect all the data bits in order to compromise the system. Nevertheless, the analysis suggests that, even if all the chunks were placed in one bucket, BIFS could still provide sufficient resistance against privacy infringements.

Considering a more intelligent attack, we assume that the adversary may arbitrarily choose some files and insert them into the BIFS. This is much similar to the chosen-plaintext attacks. In this scenario, the adversary may choose a piece of data d_1 and apply the bit-interleaving process to obtain d_1'. Then, if some user writes d_1 to the BIFS, the adversary may detect the on-chunk slice allocation scheme of d_1 by comparing the bits on chunks with bits in d_1'.

However, knowing the slice allocation scheme of some blocks does not hurt the BIFS because the property of the irreversible hash function f guarantees that it is nearly impossible to obtain the original data from the hashing outputs. Thus, the user's credential is secure. Meanwhile, the cutting and locating scheme of slices in different blocks is generated from different block descriptors. That ensures the adversary may not apply the same slice allocation scheme to other blocks.

The analysis above suggests that it is tremendously difficult for an adversary to identify the slices of a block for any reasonably non-trivial file system without knowing the block descriptors which are controlled by the user. As BIFS follows the design principle of "hide data by re-ordering, not substitution", an adversary may be able to reveal some information of the user's files in some pathetic cases (e.g. a file system with only two files, one containing all 0s and one containing all 1s). These pathetic cases are obviously rare in the real-world systems. Although it is possible to further strengthen the system by introducing some artificial files, we leave the protection of these trivial file systems as a non-goal for BIFS. Meanwhile, some statistical characteristics of the file system, such as the ratio of 0s to 1s, may be exposed to the adversaries. In a very-large-scale storage system, a large number of files from different users are stored together. This prevents the statistical characteristics of the whole file system from exposing the information of individual user files.

F. Opportunity of compression and deduplication

Retaining certain regularity of data bits constituting the on-disk state is an advantage of BIFS. As files sharing majority of content with slight differences are common in file systems [28], there remain opportunities for compression and deduplication for such files. We will evaluate the compression and deduplication opportunity using real-world examples in Section V-C.

Meanwhile, the compressibility does not compromise privacy protection. As users' credentials are different among users, which may along with the chunk ID and the physical slice number determine the length of each slices cut from the bit-interleaved sequences, the distribution of slices has significant variation. This prevents unauthorized users from recovering user data that they should not access.

IV. Implementation

BIFS presents a file system abstraction similar to traditional UNIX file systems. The current implementation of BIFS uses Amazon S3 (Simple Storage Service) as the chunk store. The BIFS daemon is implemented on Linux using the FUSE (File System in User Space) interface, which is a loadable kernel module that enables a file system to run in user mode without modifying the kernel code. Through the bit-interleaving process, file data are re-ordered and stored on chunks which are implemented as key/value pairs in the S3 storage service.

Amazon S3 provides the "eventual consistency" [29], which does not provide a guarantee of when the update can be seen. Reading an object after an update may not always return the most up-to-date state. This is a critical issue for the file systems on S3. Instead of immediately deleting the chunk after updating it to S3, we keep it in the cache for a while and verify whether the latest updates are visible on S3. This verification mechanism enhances the robustness of BIFS.

The sizes of some data structures in the design affect the balance between metadata and file data as well as the efficiency of the file system operations. We design the chunk ID and CSID to be 64-bit integers in BIFS. An 8 KB block may contain up to 128 block-descriptors which may describe chunks containing 1 MB data. That also means that files of no more than 1 MB do not need to use second-level index blocks, benefiting the performance on small files. Based on the block size and bit-interleaving interval, we specify the suitable range for slice length to be 4096 – 5120 bits.

V. Evaluation

In this section, we first introduce the configuration of our experiment testbed. Then, we measure the performance of

TABLE I
BIFS OPERATION THROUGHPUT

Operation	Throughput
sequential read	2.3 MB/s
sequential write	18.3 MB/s
file creation	4688 ops/s
file deletion	6052 ops/s

the BIFS, and compare it with several existing file systems. Finally, we verify that the bit-interleaved on-disk state can still support effective compression and deduplication operations.

To construct similar work environments for the file systems, the S3 buckets used for the BIFS are all created in the "US Standard" region. The server end of the other file systems in the comparison are installed on Amazon EC2 large instances in the Amazon "US East" region (US East – Virginia datacenter).

We evaluate the throughput of BIFS in Section V-A with a benchmark program we have designed. In Section V-B, Bonnie++ and PostMark are used to compare the performance of BIFS with several existing file systems. Bonnie++ measures the performance on large files in our experiment, and Postmark evaluates the file system performance with a number of small files. While many current distributed file systems focus on optimizing the performance on large files, small files do exist and have an impact on the perceivable performance of a file system. Hence, both small and large files are used in our evaluation. Finally, we analyze the compression and deduplication capability preserved by BIFS in Section V-C.

A. Operation throughput

The missing of locality and the eventual consistency model of S3 penalize the BIFS performance. However, a cloud-based storage solution should be able to overcome these constraints and achieve acceptable performance. We want to determine the raw performance in terms of throughput. We designed a simple benchmark program which measures the throughput of sequential read, sequential write, file creation, and file deletion of BIFS.

In the test for file creation and deletion, our benchmark program creates 10,000 empty files and deletes them.

The measurement of our benchmark program (Table I) shows that BIFS provides acceptable performance as an Internet-based file system using cloud storage as backend.

B. Performance comparison

We use Bonnie++ and PostMark benchmark programs to evaluate the performance of the BIFS, and compare it with several existing solutions including S3FS, NFS, GlusterFS, and MooseFS. Limited experiments are also conducted on HDFS—a variant of GFS—to examine its read and write throughput. However, its programming interface does not allow us to run the Bonnie++ and Postmark benchmark programs on it. The S3FS, GlusterFS, MooseFS and NFS client are mounted on the same testing computer as BIFS.

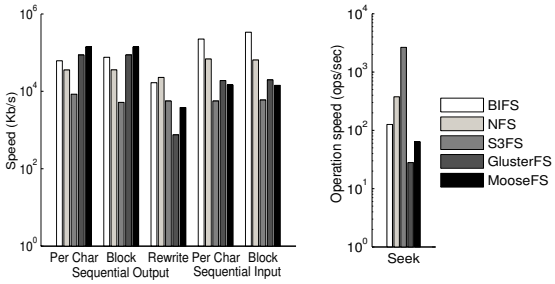

Fig. 6. Bonnie++ results

a) Bonnie++: We use Bonnie++ to compare the speed of BIFS and other file systems on read, write and seek operations. In this test, Bonnie++ is configured to perform these operations on a relatively large file. The results are shown in Figure 6, and indicate an acceptable performance for BIFS.

We also installed the HDFS client on our testing computer and set up a NameNode and a DataNode on the EC2 large instances. As HDFS does not provide users the UNIX-like file system interface and the existing FUSE-based solutions for porting HDFS [30] on the local file system hierarchy reduce its throughput, we do not evaluate it with Bonnie++ and Postmark. Instead, we "get" and "put" a large file from/to HDFS to measure its throughput in the Internet-based environment. Our measurement shows that HDFS achieves a reasonable performance of 5.1 MB/s for reading and 7.2 MB/s for writing. Through the method for testing HDFS is quite different from that for the other file systems, which made it not comparable with the measurement results from Bonnie++, this measurement of HDFS can provide some knowledge of HDFS for the sequential read and write on large files.

b) PostMark: PostMark evaluates the performance of a file system with a number of small files. We configure Postmark to create 20,000 files and perform 100,000 transactions in 10 directories, which is the typical and recommended setting for file system benchmarks [31]. The moderately large number of files and the random-access transactions may cause a large number of cache misses. Although S3FS is more stable than several other cloud-based file systems we have tested, we observed several faults in the experiment of S3FS. This is caused by the eventual consistency of Amazon S3. In order to compare the performance of S3FS, we reduce the test size to 200 files and 1,000 transactions for S3FS. In contrast, BIFS eliminates this problem as described in Section IV. This also shows the robustness of BIFS.

The test results (Figure 7) show that, existing solutions are not optimized for small files across a wide-area network. Meanwhile BIFS provides an acceptable performance on this workload because of the optimized caching scheme and the partial locality mechanism.

C. Compression and deduplication opportunities

We are interested in verifying the compression and deduplication capability that the bit-interleaving process preserves

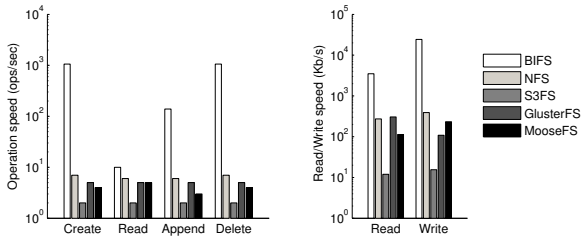

Fig. 7. PostMark results

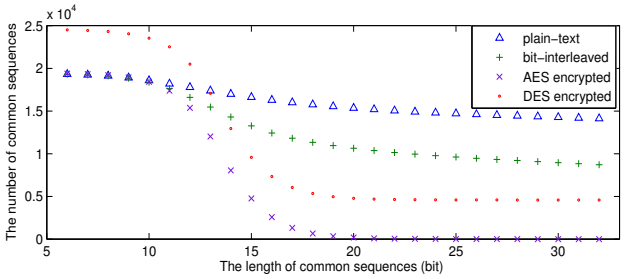

Fig. 8. Number of common bit sequences in two versions of the "/kernel/mutex.h" file in the Linux kernel

after the bit-level slicing and re-ordering. Using BIFS, is it still possible for a cloud storage provider to perform efficient compression and bit-level deduplication across multiple users' data? In this part, we answer this question by simulating the compression and bit-level deduplication with simple compression tools. More complex and efficient compression and bit-level deduplication mechanisms can be implemented by the cloud storage service provider to further reduce the storage space consumption.

Figure 8 shows the matching bit sequences in the two slightly different files ("/kernel/mutex.h" file in two versions of Linux kernels) with the length of the sequences varying from 6 bits to 32 bits. Our results show that the bit-interleaving process can preserve common bit sequences in the two files. After bit-interleaving, the files share 60% as many common bit sequences as those in the clear. In contrast, in the files encrypted by AES or DES, the number of long common bit sequences decreases dramatically.

Retaining common bit sequences provides opportunities for compression. We examine whether these tools may take this advantage. We store the two versions of "/ipc" directory from different Linux kernels in one volume, and transform the volume with AES, DES, and bit-interleave process. We use five compression tools—7-zip,bzip2, gzip, rar and zip—to compress the volume with their best compression configuration. The experiment results suggest that the compression rate of the volume transformed with bit-interleave process is 1.3 to 2.3 times higher than those transformed with AES or DES. This shows that the bit-interleave process is effective in retaining some structural regularities in user files, which makes BIFS potentially capable of effective compression and bit-level deduplication.

VI. Related Work

With the advent of datacenters and globalized computing application systems, a number of distributed file systems have been implemented to store very large data sets. Ghemawat et al. designed GFS to provide a scalable file system with very high throughput on a constellation of commodity PC servers [1]. The MooseFS [32], Lustre [33], Ceph [34] and HDFS [35] follow a similar architecture, which comprises one or multiple active master nodes and a large number of "chunk servers". Along another direction, GlusterFS [36] keeps the server simple: the server exports an existing file system and leaves it up to client-side translators to structure the store. These scalable file systems can potentially manage petabytes of data in datacenters or enterprise IT facilities. However, they cannot provide adequate privacy protection to end users. The user data are exposed to whoever can access the storage media, and many such file systems overlay on local Linux file systems [1], [37], making it technically easy for unauthorized users to examine private data. BIFS complements these file systems by providing strong privacy protection on uncontrolled public storage.

Traditionally, data privacy is protected by either restricting the access or anonymizing the data. Access restriction can be achieved by encryption, mandatory access control, or discretionary access control. However, the access control cannot protect data from those who can read the physical sectors on the hard drives. It is technically possible to employ client-side cryptography to protect user data on public online storage. The SWALLOW file system [38], for example, encrypts files on the client side before uploading them to a remote server. Blaze designed Cryptographic File System (CFS) [39], which encrypts not only file data but also the more sensitive metadata. Farsite [40], SCARED [41], and SUNDR [42] ensure the secrecy of file content with cryptographic techniques on potentially untrusted nodes.

However, encryption incurs noticeable computational overhead, and deprives deduplication and compression opportunities. Though it is possible to use file specific keys to generate identical ciphertexts (convergent encryption) [15], [16], such approaches can neither provide the required privacy protection strength nor handle the common case of files with slight variations. For data with minor differences, the encryption keys generated from the file contents become different, which result in the different ciphertexts that cannot be compressed. Moreover, the convergent encryption approach introduces additional overhead on the key storage and management. In contrast, BIFS follows the principle of *"re-order, not substitute"* so that not only is the computational overhead reduced, but also regularity in user data is preserved to facilitate compression by cloud storage providers across multiple users' data.

VII. Conclusion

Focusing on the protection of on-disk state, BIFS takes a bit-inter-leaving approach to providing strong privacy protection. Implementing BIFS in a Linux client and Amazon S3 storage configuration, we evaluate the performance of BIFS, and prove

that a file system can protect user privacy, provide reasonable performance, yet allow storage system to conduct a certain level of compression.

ACKNOWLEDGMENT

We wish to thank Amazon for providing us the Amazon AWS research grant for running experiments on the Amazon AWS/EC2 platform. We also thank BGI for their help in our research. This work is supported in part by HKUST research grants SBI08/09.EG03, REC09/10.EG06 and DAG11EG04G.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 205–220.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, 2008.

[5] J. Kincaid, "This is the second time a Google engineer has been fired for accessing user data," http://techcrunch.com/2010/09/14/google-engineer-fired-security/, last access: Oct. 28, 2011.

[6] M. Creeger, "CTO virtualization roundtable: Part II," *Commun. ACM*, vol. 51, no. 12, pp. 43–49, 2008.

[7] M. Bubbers, "Hack leaks Honda, Acura customer info," http://autos.sympatico.ca/auto-news/7296/, last access: Oct. 28, 2011.

[8] R. Pletka and C. Cachin, "Cryptographic security for a high-performance distributed file system," in *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 227–232.

[9] C. P. Wright, J. Dave, and E. Zadok, "Cryptographic file systems performance: What you don't know can hurt you," *Security in Storage Workshop, International IEEE*, 2003.

[10] E. Zadok, I. Badulescu, and A. Shender, "Extending file systems using stackable templates," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1999, pp. 5–5.

[11] M. A. Halcrow, "eCryptfs: An enterprise-class encrypted filesystem for linux," in *Proc. of Linux Symposium*, 2005, pp. 201–218.

[12] A. Mathur, M. Cao, and S. Bhattacharya, "The new Ext4 filesystem: Current status and future plans," in *Proc. of Linux Symposium*, 2007, pp. 21–34.

[13] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, ser. FAST'11, 2011.

[14] D. Geer, "Reducing the storage burden via data deduplication," *Computer*, vol. 41, no. 12, pp. 15 –17, Dec. 2008.

[15] J. Douceur, A. Adya, W. Bolosky, P. Simon, and M. Theimer, "Reclaiming space from duplicate files in a serverless distributed file system," in *Proc. of the 22nd International Conference on Distributed Computing Systems*, 2002, pp. 617 – 624.

[16] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller, "Secure data deduplication," in *Proc. of the 4th ACM international workshop on Storage security and survivability*, ser. StorageSS '08, 2008, pp. 1–10.

[17] "Amazon Simple Storage Service (Amazon S3)," http://aws.amazon.com/s3/, last access: Oct. 28, 2011.

[18] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," *UC Berkeley Technical Report UCB/EECS-2009-28*, February 2009.

[19] S. Kent and R. Atkinso., "Internet draft of iSCSI security protocol," http://www.ietf.org/rfc/rfc2401.txt, Nov. 1998, last access: Oct. 28, 2011.

[20] M. Szeredi, "SSH File System," http://fuse.sourceforge.net/, last access: Oct. 28, 2011.

[21] M. Milenković, A. Milenković, and E. Jovanov, "Using instruction block signatures to counter code injection attacks," *SIGARCH Comput. Archit. News*, vol. 33, pp. 108–117, March 2005.

[22] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *Proc. of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 1–16.

[23] "Windows Azure Platform," http://www.microsoft.com/windowsazure/, last access: Oct. 28, 2011.

[24] "Google App Engine," http://code.google.com/appengine/, last access: Oct. 28, 2011.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001, pp. 149–160.

[26] D. E. Eastlake and P. E. Jones, "US Secure Hash Algorithm 1 (SHA1)," http://www.ietf.org/rfc/rfc3174.txt, last access: Oct. 28, 2011.

[27] R. Rivest, "The MD5 Message-Digest Algorithm," http://www.ietf.org/rfc/rfc1321.txt, 1992, last access: Oct. 28, 2011.

[28] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, 2001, pp. 174–187.

[29] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, pp. 40–44, January 2009.

[30] "hdfs-fuse," http://code.google.com/p/hdfs-fuse/, last access: Oct. 28, 2011.

[31] J. Katcher, "PostMark: a new filesystem benchmark," Technical Report TR3022, Network Appliance.

[32] "MooseFS," http://www.moosefs.org/, last access: Oct. 28, 2011.

[33] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *Proc. of Linux Symposium*, 2003, pp. 380–386.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06, 2006, pp. 307–320.

[35] "HDFS Architecture," http://hadoop.apache.org/hdfs/docs/r0.21.0/, last access: Oct. 28, 2011.

[36] "GlusterFS," http://www.gluster.org/, last access: Oct. 28, 2011.

[37] P. Vajgel, "Needle in a haystack: efficient storage of billions of photos," http://www.facebook.com/note.php?note_id=76191543919, last access: Oct. 28, 2011.

[38] D. Reed and L. Svobodova, "SWALLOW: a distributed data storage system for a local network," in *Local Networks for Computer Communications*, 1981, pp. 355–373.

[39] M. Blaze, "A cryptographic file system for UNIX," in *Proc. of the 1st ACM conference on Computer and communications security*, 1993, pp. 9–16.

[40] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: federated, available, and reliable storage for an incompletely trusted environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 1–14, 2002.

[41] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long, "Authenticating network-attached storage," *IEEE Micro*, vol. 20, no. 1, pp. 49–57, 2000.

[42] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proc. of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.