"1st International Symposium on Computing in Informatics and Mathematics (ISCIM 2011)"
in Collabaration between EPOKA University and "Aleksandër Moisiu" University of Durrës
on June 2-4 2011, Tirana-Durres, ALBANIA

Improving data integrity and performance of Cryptographic Structured Log File Systems

Genti Daci[1], Megi Shyle[2]

[1] *Departement of Information Technology, Polytechnic University of Tirana*
[2] *Departement of Information Technology, Polytechnic University of Tirana*

ABSTRACT

Modern File systems like CLFS (Cryptographic Log Structured File System) are aimed to provide security and confidentiality. Current deployments of such File Systems do not currently ensure data integrity of the encrypted data that is stored on disk. Due to Kernel bugs, racing conditions and arbitrary dead-locks, CLFS data on the disc can be damaged, also there is always the possibility that system users can modify the encrypted data. Our study aims toward ensuring data integrity on CLFS without compromising on overall performance. This paper considers the standard methods using file metadata check-summing in CLFS with the main goal to overcome one of its major limitations, low performance of File-System check-summing. CLFS matches our performance expectations, as it performs close enough to non-cryptographic file systems. To improve the performance of the check-summing process we try to study and examine various design choices and propose an in-kernel database for storage and reduction of check-sum verification once in N read requests.

INTRODUCTION

A file system is meant to store typically large amounts of data, which may be either critical or sensitive, so they need to be protected. Modern File Systems, such as CLFS [1] ensure confidentiality of data, encrypting them. Usually File Systems that comprise data encryption result to perform considerably slower than non-cryptographic File Systems. New techniques implemented in CLFS manage to reach its performance goals, as they go close to fast local file systems. Encryption is a native characteristic of this file system and it ensures that information is accessible only to those authorized to have access, while not being affected by the overhead of encryption with more than an order of magnitude [1].

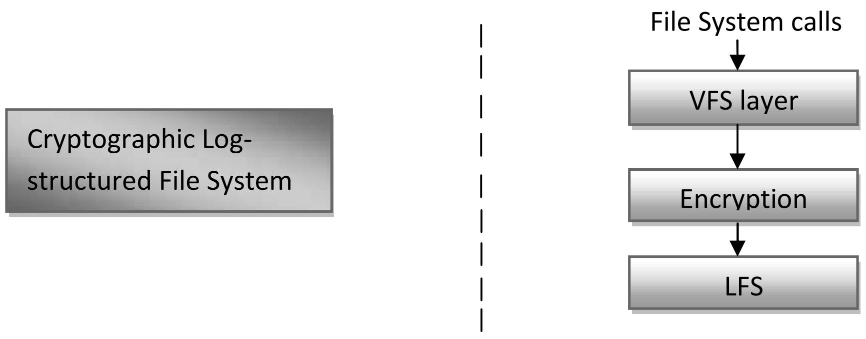

Figure 1. This figure shows the difference between the CLFS, which implements encryption as a native characteristic and a Layerd File System organisation.

Being native means including the data encryption code into the file system code. We are generally used to systems, where encryption is added as a layer, thus allowing it to be bypassed, but this is not our case, as figure 1 demonstrates.

Confidentiality is an important part of the overall system storage security, but it is not everything. Considering that hard disks malfunctioning, data on it can be damaged even though it is encrypted. It can also be affected by attackers, both physically or while communicating over an insecure network. Physical access on disk allows the attacker to change the unencrypted part, which in our case comprises the ifile, without the system knowing it. Thus, suggesting a way to make the file system immune from such data corruption, either as a result of a malicious attack or hardware failure, is our scope. We aim to protect our sensitive data, checking for inconsistencies, to obtain data integrity.

In this paper we will describe the context of our working environment, Cryptographic Log Structured File System, which is in turn based on LFS [2]. This work comes to life to improve this file system, thus a detailed explanation of the techniques it uses to avoid the latency encryption carries, is needed. Further, we will list the techniques used to achieve the integrity of data requirement. They will be examined in a selective prospective, to pick out the most congruent solution to our specific case, as shown in Figure 2.

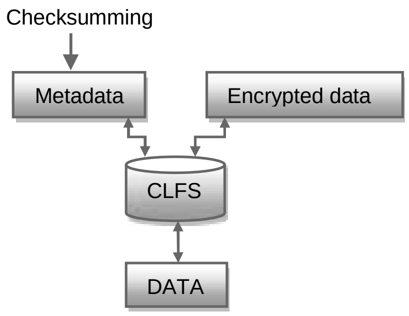

Figure 2. Overview of our scope, adding integrity checking to CLFS.

THE BACKGROUND, (C)LFS

In this section, we consider important to unfold the features of the Cryptographic Log-Structured File System, to justify our choice. To make a long story short, the crucial argument that supports CLFS is performance. Previous cryptographic file systems come in different implementation, the encryption can be block based, as well as disk based, on network file systems or stackable ones. Their common characteristic is bumping into the knot of being too CPU sensitive. To provide performance solutions CLFS concentrates on previous cryptographic File Systems weaknesses, being the speed of the encryption algorithm used and the writing latency.

2.1 The encryption algorithm

CLFS [1] considered that improving the speed of the algorithm used to encrypt would be useful both ways of read and write operations. Let's take a look at a couple of the most known cryptographic file systems, to check out for their deficiencies.

Blaze (1993) [3] implemented CFS as a network file system and its main drawback resulted the continuous context switching overhead. The encryption algorithms it can make use of are DES [4], SAFER[5], etc. A completely different approach was treated by Zadok (1998) in CryptFS [6], as it is a stackable file system. This provides it with portability allowing execution above any kind of native file system. It also comes out to be faster than CFS by a factor that fluctuates from 2 to 37 times. The bottleneck of this file system is precisely the encryption algorithm it uses, blowfish [7].

To overcome this drawback CLFS uses SEAL 3.0 [8], a software optimized encryption algorithm. Its main advantages that meet our requirements are its speed and the fact that it allows the data to be pre-computed. P. Rogaway and D. Coppersmith (1997) demonstrated that SEAL uses approximately 4 cycles to process a byte and results up to 10 times faster than DES. SEAL works as a stream cipher, where the encryption depends not only from the plain text and the encryption key, but also from the position of the data. The key is 160bit long and SEAL uses it to map a 32bit string to an L bit one, where L in our case is less than

64 kB. The trick that makes the work with SEAL that fast is pre-computing. Being LFS the ground upon which CLFS was built, allows us to know precisely the position of the next write and that's the reason why we can pre-compute the whole key stream, so that the encryption process is reduced to a simple XOR operation between the key and the plain text.

The underlying file system, LFS

The speed of I/O bound applications is limited by the write performance of the file system. A log-structured file system is designed for high write throughput. Rosenblum and J. Ousterhout (1991) showed how all updates to data and metadata are written sequentially to a continuous stream, called a log [2]. Conventional file systems put a great emphasis on spatial locality and make in-place changes to their data, which leads to slow seeks. LFS assumes that taking care of locality will no longer be effective, as memory size is increasing to the extends of satisfying all read operations using the cache. Storing data in a log avoids seeks, therefore minimizes the movement of disk's head and maximizes write throughput.

The benefits of using it as a base for CLFS is that we know where the position of the next write is located. Besides, file system writes are only performed at the end of the log which means that inconsistencies can only be located in the last segment of the log, which greatly speeds up crash recovery.

(C)LFS organization

The Cryptographic Log-Structured File System [1] is very similar to LFS, as much as concerns its overall format. It is organized into segments, only one of which is active at one time. Each segment has a header called a summary block. Summary blocks keep pointers to the next summary block, linking segments into one long chain that LFS treats as a linear log.

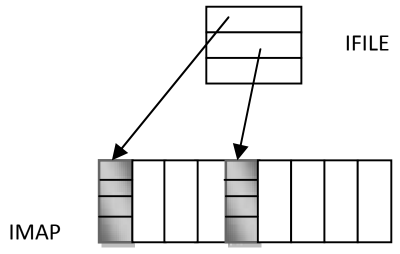

Figure 3. LFS segment organization

Upon creation of the file system a random key is automatically generated for the first segment. Afterwards the segment key and the key stream offset are stored encrypted in the corresponding segment summaries. The key stream is calculated increasing it with the segment size. To increase performance CLFS uses a cache to store the expanded keys and their positions. These keys are encrypted using AES [9] and a system key provided by the user.

In a read operation CLFS would act initially the same as LFS, by searching for the desired block to read. The search would start by the index file's inode stored in the superblock, where we would find the inode map, followed from the inode. These operations are the majority of times bypassed, since this data is stored in the cache. Obviously, CLFS to ensure effective encryption cannot stop at the point of LFS. It has to read the exact position of the segment summary from the ifile. After reading it, decrypt the segment key and the key offset, to generate the key stream. Finally the read data has to be XOR-ed with the key stream to decrypt them.

The write operation differs from LFS only in a couple of steps. The first chunk of data is removed from the list and its key is already known, so we can perform the XOR operation. This is done with all the chunks of data and afterwards the segment summary is changed.

DATA VERIFICATION TECHNIQUES

Ensuring data integrity is fundamental to computer systems. Several factors may induce to data errors, to mention media failures, kernel bugs and racing conditions. Even an attacker who has reached to gain administrator privileges can modify the data. Threats are multidirectional and our system cannot be left unprotected, so our approach is based on the existing cryptographic file system to which the integrity checking capability is added.

Mirroring

Making exact copies of our data, i.e. mirroring, can be a way of managing it reliably. The process would comprise the comparison of our data with the mirrored one, before operating with it. This method would easily detect changes in one of the replicas, providing integrity in case the changes occur because of hardware malfunctioning.

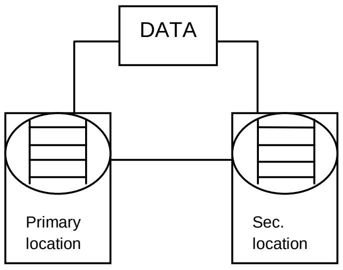

Figure 4 This figure shows an implementation of the mirroring technique using two different disks.

However, it doesn't tell which of the copies is the genuine one. It is not able to perform correctly in the case of an intruder, either. This is due to the fact that both of the replicas can be changed, so that the system lacks the tools to detect intrusion and integrity is not obtained. The inefficiency of this method also arises when we consider the storage space it requires and the time we need to spend checking both replicas.

Parity

Parity is a simple yet effective method to assure integrity protection. It adds one bit to the pattern and then requires that the modulo-2 sum of all the bits of the pattern and the parity bit have a defined answer. Parity bits are sufficient to catch all single errors in the pattern. However the system will not detect any double errors and these will be flagged as valid.

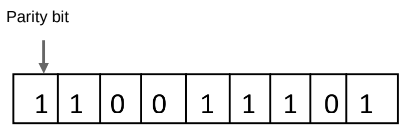

Figure 5 The parity technique, the parity bit added to a bit pattern.

Check-summing

Check-sums are exactly the same as parity with two changes: To create the check-sum of a pattern of entities of length n bits, an n-bit entity (the check-sum) is added and the modulo 2 sum of the entities and the check-sum is constrained to be zero. CRC is more secure than check-summing, but it needs more calculation, as it adopts a more complex technique.

Check-sums can be implemented in various ways, we can compute a per block check-sum, or a per file check-sum. The latter proposes two alternatives, storing check-sums apart from the data, otherwise interleaving data files and check-sums with the purpose of making more efficient use of data locality.

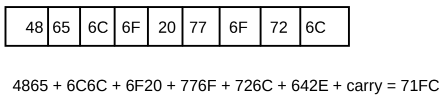
48 65 6C 6F 20 77 6F 72 6C
4865 + 6C6C + 6F20 + 776F + 726C + 642E + carry = 71FC

Figure 6. The check-summing method, the calculated check-sum at the end of the bit pattern

CRC

CRC is a method which seeks to improve on check-sums by increasing the complexity of the arithmetic. They use polynomial division to determine the value of the CRC. The basic idea of CRC algorithms is simply to treat the message as an enormous binary number, to divide it by another fixed binary number and to make the remainder from this division the check-sum.

OUR APPROACH

To constitute our design model we have developed our idea in several layers, each of which is built on answering one single question per layer. As we previously explained ensuring data verification in a file system is essential to its integrity. On the other hand we need to preserve as much as possible the major advantage of our native file system CLFS, which despite being cryptographic fully meets performance requirements. So, naturally the decisions we need to make involve which method of verification is more appropriate, what part of the data will be verified unless all of it, where will this extra information be stored and finally how frequently the verification will occur. Let's analyze them further.

The method used to provide data integrity

Check-summing is the most common method to ensure data integrity [10]. If we want our data to be protected from intruders as well as from transient errors, the check-sum need to be protected with a secure hashing scheme such as MD5 [11] or SHA1 [12]. Since the latter has proven to be more secure, it was the one we chose.

We considered efficient to compute check-sums for metadata. It comprises all the inodes and the ifile. Checking different fields of the metadata will allow us to find out if any malicious modification has been made to the data, because almost any modification to our data will be reflected to the metadata [13]. Furthermore, this decision is more advantageous as the amount of metadata is considerably less than data, and results to be efficient on timing, hence giving a better performance.

Benchmarks of improved LFS under stress test using Tar and Compilations

Knuttson (2002) [1], after implementing his cryptographic File System, made several tests to check its performance affected by the overhead encryption introduced. It resulted to perform close to non-cryptographic File-Systems, being an ideal solution for systems which store large amounts of sensitive data.

Now, apart from encryption we have added a new feature, file integrity checking, through metadata check-summing. We need to test our system to evaluate the latency, this further overhead cause. The tests made are on a .tar file and also tying a kernel compilation. We have submitted four different file sizes, respectively 100Mb, 250Mb, 500Mb and 1Gb.

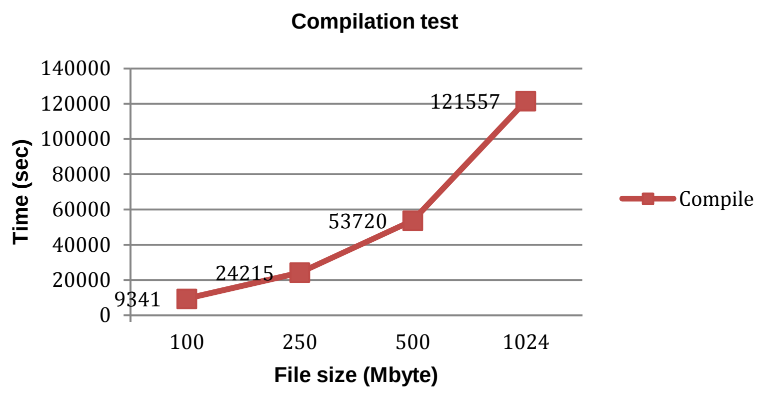

Figure 7. Results of the compilation test, the time of compilation of four different sized files, after adding integrity checking to metadata. After calculation the overhead results a mere 4%.

Knuttson tar tests completed in approximately 562sec for a 64Mbyte file. Our 100Mbyte file completed in 927sec, as shown in figure 8. After a comparative analysis of the tests, we notice that the overhead reaches the margins of 4% on compilation tests and 6% on tar tests. The first is due to the fact that compilation is a CPU bound application and is not particularly affected by the File System, whereas the latter is slightly affected.

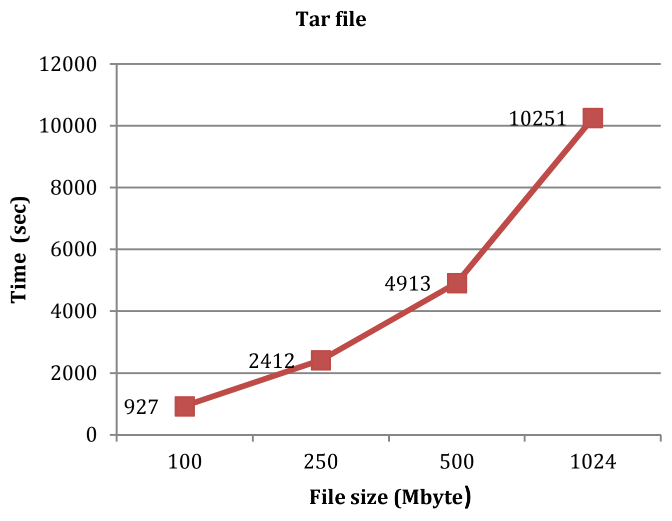

Figure 8. Results of the tar test, the results of four different sized tar files. They are all composed of a great number of small files and this test was chosen

considering it is a very frequent operation in a file system. Its overhead reached a 6%.

CONCLUSIONS

Trying to add integrity to a cryptographic File System, like CLFS, initially sounds risky from the performance point of view, but choosing the most appropriate method and a reduced amount of data to check is the clue to success.

Our benchmark showed an overhead that varies from 4-6%. This is an appealing result, considering the systems where CLFS is intended to work. We provide highly secure metadata integrity checking, i.e. the data stored in our system is correct, or at least cannot be modified undetectably.

We assumed that checking the integrity of both data and metadata would considerably affect performance, while not evidently improving integrity. Metadata check-summing was considered sufficient to meet our requirements. But, in spite of this, further work can be done on testing what this overhead would exactly be.

As a conclusion, we successfully managed to improve a cryptographic file system, enabling it with integrity without sacrificing performance.

REFERENCES

[1] Karl Knutsson (2002) Security Without Cost: A Cryptographic Log-structured File System. *Department of Software Engineering and Computer Science Blekinge Institute of Technology*, 1-26

[2] M. Rosenblum and J. Ousterhout (1991) The design and implementation of a log-structured file system. *Symposium on Operating System Principles Proceedings*, 1–15

[3] M. Blaze. (1993) A cryptographic file system for unix. *Proceedings of the 1st ACM Conference on Computer and Communication security*, 9–16

[4] Data encryption standard. (1977) *Federal Information Processing Standards Publication 46-2*

[5] J. Massaey (1994) Safer k-64: A byte-oriented block-ciphering algorithm. *Fast Software Encryption, Cambridge Security Workshop Proceedings*, 1–17

[6] E. Zadok (1998) Cryptfs: A stackable vnode level encryption file system. *Technical report, CUCS-021-98,* 1-14

[7] B. Schneier (1994) Description of a new variable-length key, 64-bit block cipher (blowfish). *Fast Software Encryption, Cambridge Security Workshop Proceeding,* 191–204

[8] P. Rogaway and D. Coppersmith (1997) A software-optimized encryption algorithm. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 1-14

[9] Advanced encryption standard. (2001) *Federal Information Processing Standards Publication 197*, p. 5-26

[10] Gopalan Sivathanu, Charles P. Wright and Erez Zadok (2004) Enhancing File System Integrity Through Check-sums. *Stony Brook University Technical Report FSL-04-04*, 1-5

[11] R. L. Rivest (1992) The MD5 Message-Digest Algorithm. *Technical Report RFC 1321, Internet Activities Board*, 1-21

[12] SHA1: Secure Hash Standard (1997) *Federal Information Processing Standards Publication 180-1*, 1-11

[13] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok (2004) I3FS: An In-Kernel Integrity Checker and Intrusion Detection File. *Stony Brook University*, 67-77