Enhancing File System Integrity Through Checksums

Gopalan Sivathanu, Charles P. Wright, and Erez Zadok
Stony Brook University

Technical Report FSL-04-04

Abstract

Providing a way to check the integrity of information stored in an unreliable medium is a prime necessity in the field of secure storage systems. Also in operating systems like Unix that allow a user to bypass the file system to access the raw disk, integrity checks not only detect data corruption, but also track malicious attacks. Checksumming is a common way of ensuring data integrity. Checksums that are generated using cryptographic hash functions prevent unauthorized users from generating custom checksums to match the malicious data modification that they have made. This report discusses the various design choices in file system checksumming and describes an implementation using an in-kernel database in a stackable encryption file system.

1 Introduction

When a file system does not trust the disk in which it stores its data and metadata, several interesting problems arise. This "untrusted disk" assumption is normally prevalent in network attached storage systems where the client file system communicates with the disk over an insecure network, and hence is potentially vulnerable to attacks on the network link if an attacker actively modifies or passively listens to file system traffic. However, even in the context of local disk file systems, such concerns may be valid and useful. For example, inexpensive disks such as IDE disks may silently corrupt the data they store, due to magnetic interference or even transient errors. Also, an attacker on a system could access the raw disk directly and write to file system metadata or data blocks, without the file system knowing it. Thus, making the file system robust to such data corruption, either as a result of a malicious attack or hardware malfunctioning, is useful.

The common method of detecting corruption is through the use of checksums. Applied in the context of file systems on untrusted disks, there are various design choices that arise in checksumming. First, if the file system wants to detect malicious attacks (and not just genuine hardware errors), it must ensure that the checksum cannot be forged to match some deliberately corrupted data. This might require using some secure hashing scheme so that obtaining the checksum would require some private key, perhaps derived from the user password or something similar.

One of the major limitations of file system checksumming is performance. This is the traditional security-performance trade-off that systems programmers are ready to live with.

In this project, we have evaluated the advantages and disadvantages of several ways of storing and retrieving data and meta-data checksums from the disk, and implemented a method that uses an in-kernel database [3] to persistently store and retrieve the checksums. The implementation is for NCryptfs, a stackable encryption file system [9].

2 Background

2.1 Secure Hashing

The use of cryptographic hash functions has become a standard in Internet applications and protocols. Cryptographic hash functions map strings of different lengths to short fixed size results. These functions are generally designed to be collision resistant, which means that finding two strings that have the same hash result should be infeasible. In addition to basic collision resistance these functions, like MD5 [6] and SHA1 [1], also have some properties like randomness, unpredictability of the output, proper mix etc. It is the combination of these properties that are attributed to cryptographic hash functions that make them so attractive for several uses beyond their original design as basic collision resistant functions.

HMAC is a specific type of a secure hashing function. It works by using an underlying hash function over a message and a key. It is currently one of the predominant means to ensure that secure data is not corrupted in transit over unsecure channels (like the Internet). This can be used in the context of storage systems also to ensure integrity during read.

Any hashing fuction could be used with HMAC, although more secure hashing functions are preferable. An example of a secure hash function (which is commonly used in HMAC implementations) is SHA-1. (Other common hashing functions include MD5 and RIPEMD-160). As computers become more and more powerful, increasingly complex hash functions will probably be used. Furthermore, there are several generations of SHA hashing functions (SHA-256, SHA-384, and SHA-512) which are currently available but not very widely used as their added security is not yet believed to be needed unless very high security is needed.

2.2 Stackable file systems

Stackable file systems [2] are a technique to layer new functionalities to existing file systems. With no modification to the lower level file systems, a stackable file system exists between the virtual file system and the disk file systems, intercepting calls for performing special operations and eventually redirecting them to the lower level file system. By way of a stackable file system, we can add new functionalities to existing file systems like encryption, tracing etc. Unlike traditional disk file systems like Ext2, a stackable file system mounts on a directory and instead of a device. Since this is "stacked" on top of an already mounted file system, it is called a *Stackable file system*. A stackable versioning filesystem, Verionfs [5] maintains versions of files at the stackable level.

A stackable file system has many advantages: significant among them being minimal development overhead, portability, no change to the existing file systems etc. When it is easy to develop a stackable file system, there are some operations that cannot be done using it. An example being block level operations. Since a stackable file system has the responsibility of being good to a variety of underlying file systems, it provides a higher level abstraction to the level of a file, which makes certain lower level operations almost impossible to perform from the stackable level.

The goal of this project is to implement checksumming in a stackable file system so that it can add checksumming to any lower level file system.

2.3 The Context: NCryptfs

Ncryptfs [8, 9] is a stackable cryptographic file system whose primary goal is to ensure security, confidentiality, while balancing security, performance and convenience. It encrypts file data and names before storing it to the disk. In NCryptfs, an encryption key is associated with a group of file and directories termed as an *attach*. Attaches are in-memory dentries visible at the root of ncryptfs file system. There can be any number of attaches in the file systems and these are in-memory structures. Once the file system is unmounted, the attach names are destroyed and while re-mounting it, a different attach name can be chosen for the same key and still the decrypted view can be seen. Thus an attach can be characterized as a "window" that is associated with a key that is used to see the decrypted view of the files and directories.

NCryptfs provides security and confidentiality, but does not currently ensure data integrity of the encrypted data that is stored on disk. A malicious user can always bypass NCryptfs and modify the encrypted data, and these changes might remain undetected. Thus ensuring data integrity in NCryptfs would be a useful goal. This project implements file data and metadata checksumming in Ncryptfs.

3 Threat Model

Filesystem checksumming is primarily aimed at detecting the following:

- Corruption of disk data due to hardware errors. Inexpensive disks such as IDE disks silently corrupt data stored in them due to magnetic interference or transcient errors. These erros cannot be detected by normal file systems.
- Malicious modification of data. In UNIX-like operating systems where accessing the raw disk is easy, a malicious user can attempt to modify file data by directly accessing the disk which the file system would have no knowledge of. These modifications can be detected through checksumming.
- In the particular context of a stackable file system where the underlying file system can always be directly accessed, checksumming metadata is required to determine if change of metadata like access time and modify time are made even if there is no change to the file data. This helps to detect breach of confidentiality

4 Design Choices

There are various ways to implement checksumming in a file system. Computation of checksums, storing and retrieving them should happen in the critical section of a file read and file write in order to ensure integrity. Thus, it is imperative that they are stored in the right place so that retrieving them during read does not impose any unreasonable overhead. That said, there are different design approaches which one can adopt that differ in where the file data and metadata checksums are stored. One of the important constraints that drastically limits the design choices is what is imposed by a stackable file system. A stackable file system provides a file level abstraction that prevents us from performing block level checksumming.

4.1 Block Level Checksumming

One of the methods to handle checksumming is to compute a per-block checksum for all data blocks of a file, indexed by the relative block number. The inode can be modified to introduce a new set of pointers that point to checksum blocks. Whenever a disk block is added to a file, its checksum would be computed and stored in the checksum blocks. The number of checksum blocks for a file would be:

$$No.of cksumblocks = \frac{No.of datablocks}{(blocksize/checksumsize)}$$

Typically the size of checksum would be 128 bits. The advantage of using this scheme is that the checksum blocks can be accessed just the way the data blocks are accessed from the inode and hence locality can be maintained. Moreover it does not impose any space overhead and uses the bare minimum space that is required.

Since our goal is to implement checksumming in NCryptfs which is a stackable file system, block level operations are not permitted in it and hence this method cannot be used.

4.2 Out of Bands

Since stackable file systems provide a file level abstraction, another design choice would be to have a unique hidden checksum file for every file in the file system. Since file level abstraction gives the notion of pages rather than blocks, a per page checksum can be generated for each page and stored in the checksum file. The number of pages in the checksum file would be:

$$No.of cksumpages = \frac{No.of datapages}{(pagesize/checksumsize)}$$

Though this method provides a convenient way of storing checksums, reading a file requires opening two files. Atleast every n page reads of every file would result in reading a checksum page, where n is defined as $\frac{pagesize}{checksumsize}$. Thus it imposes unnecessary overheads. Moreover it completely loses track of locality.

4.3 Inline Checksums

This is a variant of the out of bounds method. Here the checksum pages are interleaved with the data pages. There would be a checksum page after every n data pages, where n is defined as $\frac{pagesize}{checksumsize}$. This method makes effective use of locality, as most file systems try to store pages of the same file in a nearly contiguous fashion on disk. Though this method is good for retrieving checksums efficiently, updating checksums and deleting checksums during file truncation is complicated and might require unnecessary copying of data.

5 Approach

The approach we adopted is to use in-kernel databases to store data and meta-data checksums. KBDB [3] is an in-kernel implementation of the Berkeley DB [7]. Berkeley DB is a scalable, high performance, transaction protected data management system that provides the ability to efficiently and persistently store `(key,value)` pairs using hash tables, B+ trees, queues and indexed by logical record number. Since in this project, the persistent data that needs to be stored are the checksums, we have divided file data to the granularity of a memory page. Checksums are computed for the pages in memory and stored in the database. Since a checksum entry is associated with a file and the page index inside the file, we have designed the key as a stream containing the inode number and the page index of the page whose checksums is being stored. One of the main advantages of using an in-kernel database to store the checksums is locating and retrieving the checksums does not require complicated implementations. The database *get* services retrieves the checksum for the file in a pretty efficient manner as it stores it in a hash file. Similarly to checksum file metadata, important fields of the inode object are checksummed and stored in the database keyed by the inode number which is guaranteed to be unique. Checksums databases are now stored in hash file format.

6 Implementation

The implementation of filesystem checksumming was done in the Linux Kernel version 2.4 and it required addition of about 300 lines of code to the kernel. Since for this project, checksumming had to be implemented in the context of NCryptfs, certain new complications arose. In NCryptfs, files cannot be created in the root directory of the file system. To create a file in NCryptfs, an attach has to be created initially. An attach is an in-memory dentry which has an encryption key associated with it. It is a window through which all files that are encrypted with a particular key can be viewed in a decrypted fashion. Since logically an attach is the beginning point of the NCryptfs file system, We thought it fit to have per attach checksum databases. Since both data and metadata need to be checksummed, each attach would be having two databases associated with them — the `data_checksum_db` and the `metadata_checksum_db`. The pointer to these databases are implemented as the member variables of the in-memory attach structure, `attached_entry`.

Implementing checksumming required me to do the following basic steps:

1. Opening the data and metadata databases whenever a new attach is created in NCryptfs. This required modification to the `ncryptfs_do_attach()`

function.

2. Compute the checksum for a data page and store it into the data checksum database whenever it is being written to the disk. This required me to modify the `ncryptfs_commit_write()` and the `ncryptfs_writepage()` functions.
3. Retrieve the checksum for a data page from the database whenever it is read, compute the checksum for the read page and verify both.
4. Compute and store the inode checksum (meta data) whenever an inode is written to disk. For this we had to modify `ncryptfs_put_inode()` function.
5. Retrieve the inode checksum from the database when an inode is referenced, compute the new checksum and verify both. This required modifications to the `ncryptfs_lookup()` function.
6. Close the databases whenever an attach is removed from the file system. We have done this in the `ncryptfs_do_detach()` function.

The checksums generated by the HMAC function would be of length 16 bytes. For storing the file data checksums, the key value should uniquely identify the particular page of the file. For this we are using an 8-byte stream as the key whose first 4 bytes would be the inode number to which the page belongs to and the rest of the 4 bytes would be the page index of the page. Since the key value is just a number, the default hash function (the linear congruential method) followed by Berkeley DB to store hashed files is an efficient means.

Since, for metadata checksumming, fields of the inodes are to be checksummed, the key for these are just the inode numbers which is a 4-byte quantity. In the context of NCryptfs, two inode objects comes into picture which are the `ncryptfs_inode` and the `hidden_inode`. texttthidden_inode is the the inode object of the lower level file system on which NCryptfs is mounted on. During meta data checksumming, the fields of the hidden inodes are alone checksummed and not the higher level inodes. This is because we are interested in ensuring integrity of the encrypted data that is stored in the lower level file system. The fields of the inode that are needed to be checksummed are copied to a custom data structure called `ckstat` and then checksummed. The following is the `ckstat` structure:

```
struct ckstat {
        umode_t mode;
        nlink_t nlink;
        uid_t uid;
        gid_t gid;
        loff_t size;
        time_t atime;
        time_t mtime;
        time_t ctime;
        unsigned int blkbits;
        unsigned long blksize;
        unsigned long blocks;
        unsigned long version;
}
```

Since the different times, atime, mtime, and ctime are also checksummed, NCryptfs can now detect even unauthorized access and modification of file even if the file data is not necessarily changed. Thus it can detect confidentiality breach also (i.e., unauthorized examination of cipher text).

Whenever there is a checksum mismatch for a file, NCryptfs reports the same and then returns an error. This is based on the premise that majority of the integrity issue arise due to disk malfunctioning. In the case of metadata checksum mismatch, the file system stops further operations like read or write on the file. All metadata checksum mismatches are caught at the time of a lookup operation.

7 Evaluation

We evaluated the performance of checksumming using both IO intensive and CPU intensive benchmarks. For IO intensive benchmarking, we used postmark [4], a popular file system benchmarking tool. For CPU intensive testing, we compiled the am-utils utility package. Compilation of am-utils, in addition to generating huge CPU load, performs many different file system operations including truncation, deletion etc.

We performed all benchmarks on Red Hat Linux 9 with Kernel version 2.4.24 running on a 1.7GHz Pentium 4 processor with 1GB of RAM. For all the experiments we used a 20GB 7200 EPM WDC IDE disk. We remounted the filesystem during each run of benchmarks so as to ensure that any cached is cleared.

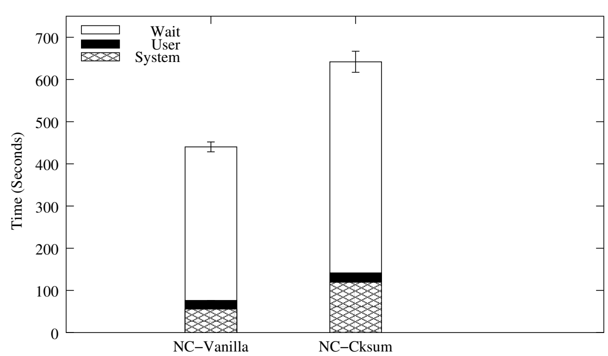

Figure 1: Postmark results

Figure 1 shows the results of Postmark. We measured the user time, system time and the elapsed time. According to the Postmark results, checksummed NCryptfs implementation had an overhead of 130% compared to the vanilla NCryptfs implementation.

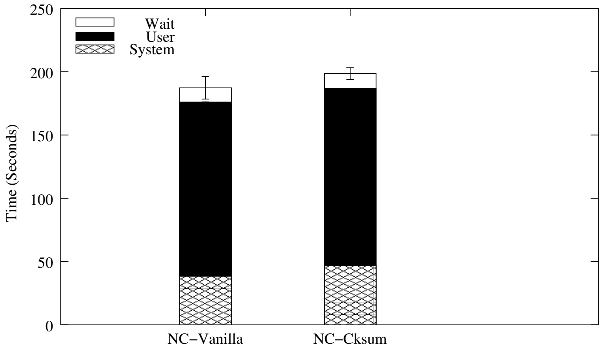

Figure 2: Am-utils compilation results

The results of Am-utils compilation in Figure 2 showed a 21% overhead due to checksumming. Microbenchmarks showed that almost 90% of the overheads in checksummed NCryptfs was caused due to the database operations. The difference in the overheads revealed by Postmark and Am-utils is because, Postmark creats a huge number of new files and directories which results in a large number of database operations whereas Am-utils does not generated that many database operations.

8 Conclusions

This project is intended to ensure integrity of data in a stackable encryption file system, NCryptfs. Thus most of the implementation specifics and part of the design was tied up with the constraints of the general stacking convention and the semantics of NCryptfs. In a regular filesytem this may not be the best way to implement checksumming. But given the conventions, the method followed is a reasonable approach.

8.1 Future enhancments

There can be potential optimizations and performance tuning possible in the implementation that would make use of suitable values for different database parameters. Two important parameters, the database page size, and the page fill factor have to be set with suitable values depending upon the key size and the data size so that the hash utilization is the maximum. These are to be fixed based on the result of further detailed performance testing, profiling and timing study.

9 Acknowledgments

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

References

[1] P. A. DesAutels. SHA1: Secure Hash Algorithm. `www.w3.org/PICS/DSig/SHA1_1_0.html`, 1997.

[2] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[3] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using Berkeley Database in the Linux kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`, 2004.

[4] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[5] K. Muniswamy-Reddy. Versionfs: A Versatile and User-Oriented Versioning File System. Master's thesis, Stony Brook University, December 2003. Technical Report FSL-03-03, `www.fsl.cs.sunysb.edu/docs/versionfs-msthesis/versionfs.pdf`.

[6] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. Internet Activities Board, April 1992.

[7] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, January 1991. `www.sleepycat.com`.

[8] C. P. Wright, J. Dave, and E. Zadok. Cryptographic File Systems Performance: What You Don't Know Can Hurt You. In *Proceedings of the 2003 IEEE Security In Storage Workshop (SISW 2003)*, October 2003.

[9] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, June 2003.