UsiFe: An User Space Filesystem with Support for Intra File Encryption

Rohan Sharma, Prathmesh Kallurkar, Saurabh Kumar, Smruti R. Sarangi
Computer Science and Engineering, IIT Delhi, India

ABSTRACT

This paper proposes a new paradigm for the design of cryptographic filesystems. Traditionally, cryptographic file systems have mainly focused on encrypting entire files or directories. In this paper, we envisage encryption at a finer granularity, i.e. encrypting parts of files. Such an approach is useful for protecting parts of large files that typically feature in novel applications focused on handling a large amount of scientific data, GIS, and XML data. We extend prior work by implementing a user level file system on Linux, UsiFe, which supports fine grained encryption by extending the popular ext2 file system. We further explore two paradigms in which the user is agnostic to encryption in the underlying filesystem, and the user is aware that a file contains encrypted content. Popular file formats like XML, PDF, and PostScript can leverage both of these models to form the basis of interactive applications that use fine grained access control to selectively hide data. Lastly, we measure the performance of UsiFe, and observe that we can support file access for partially encrypted files with less than 15% overhead.

Keywords: File system security, Cryptographic file systems, Intra-file Security.

1. INTRODUCTION

Cryptographic file systems have been in use for over two decades now. They have been implemented in userspace, in the kernel, and also at the device level. Unfortunately, all of these systems treat a file as an atomic unit. They encrypt at either the granularity of a single file or at a higher level such as at the level of a directory, or an entire storage device. These approaches have proven to be very effective in commercial and enterprise scenarios, which require a high level of confidentiality. Unfortunately, such approaches are not very effective for novel application areas like cloud and grid computing. The reason for this is that nowadays data is shared at a much finer granularity, and the domains of trust in cloud computing scenarios are of an extremely diffuse nature.

Consequently, we try to make the case of intra file encryption in generic file systems. Banachowski et. al. [4] have observed the need for intra file encryption in scientific data. They have tested their idea on object based file systems. For example, large map files that contain GIS data might have regions such as defense installations that should not be visible to all users. These specific sub-components need to be hidden from some users. At this point, there are two choices available to users. They might see the specific component as an encrypted black box, or it might be an active element that can solicit a password from the user upon a mouse click. Several other researchers have looked at data sharing in large enterprises [6,7,8]. They have suggested partially encrypted XML files, in which certain nodes and their subtrees are protected. A lot of image and dataformats, which are based on XML such as scalable vector graphics (svg), openoffice documents, and Microsoft Word documents (in XML format), can potentially leverage this technology. The use of XML encryption has found its way into commercial products, and is also being currently considered for standardization by the World Web Consortium. They have posted a working draft of XML security specifications at [3].

Instead of having a separate platform for each application domain, we propose a generic filesystem, *UsiFe* (User Space Intra-File Encryption). Different technologies for partially encrypting XML data, image files, and enterprise documents, can build on UsiFe. Furthermore, UsiFe runs entirely in userspace, and does not require modifications to the kernel.

We provide two modes of operation: *display* and *stealth*. In the *display* mode, the user is aware that there are encrypted regions in the file. We envisage application software that will allow the user to indicate her interest to decrypt regions of the file interactively such as by clicking the left mouse button. The application can then pass on a message to the underlying UsiFe layer, and UsiFe can decrypt the region, if the user supplies a valid key. In *stealth* mode, the data that cannot be decrypted is completely hidden from the user. All the file access functions such as rewind, fseek, ftell, fgetpos, and fsetpos will treat the encrypted data as non-existent. The file will effectively look shortened to a user who does not have permissions to view parts of the file.

We use the open source file access library called FUSE [2] to intercept file access system calls. We don't modify the file structure. We use existing mechanisms in the ext2fs file system to store the metadata regarding file encryption. Consequently, we don't need to implement a new file structure on the disk. We can use a regular ext2, ext3, or ext4 filesystem underneath FUSE. Since the ext file systems are the default in Linux, and many other operating systems, UsiFe can be widely used. The metadata that we store corresponds to the regions of encryption, the encryption keys, and some special functions to recompute some values in the file called *active functions*. Depending on the file access mode (*display* or *stealth*), a view of a file is generated for each user. We show in Section 3 that we can encrypt a wide variety of files using UsiFe, and with a combination of different heuristics, the performance of UsiFe is competitive with other user level file systems. Section 2 describes the implementation of our system, Section 3 presents the results, Section 4 summarizes the related work, and we finally conclude in Section 5.

2. IMPLEMENTATION

2.1 The FUSE File System Library

FUSE is an open source file system library that can be run in user space [2]. It is typically used to create custom filesystems, or run specialized file systems like UsiFe. FUSE achieves this functionality through a loadable kernel module that is able to trap file system accesses.

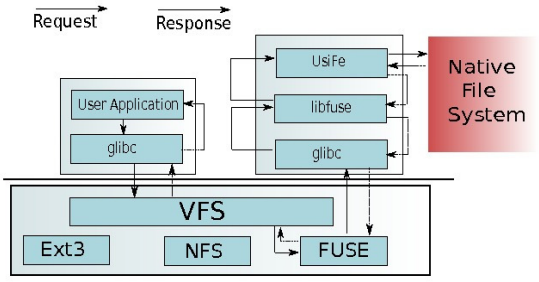

Figure 1. User Space Filesystem with FUSE

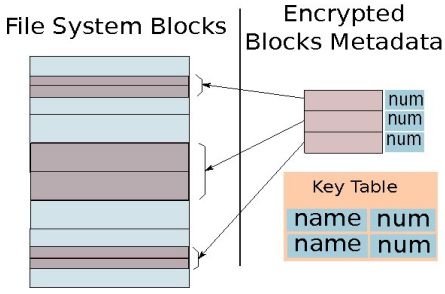

Figure 2. Metadata in UsiFe

Figure 1 shows the overall schematic of our system. The user application, does not perceive the fact that it is using a different file system. It makes regular system calls through the standard C libraries (*glibc*). The standard C libraries pass on the request to the kernel VFS layer in the Linux kernel. The VFS (Virtual File System) layer abstracts the functionality of a file system. It provides generic calls like open, read, write, and creat to work with files. The underlying file system might be one of the commonly used UNIX based file systems like ext2, or ext3, or specialized file systems such as HDF, Fuse, or ResierFS. Since the FUSE library is loaded as a kernel module, it can masquerade as a regular file system. For files that are part of the FUSE file system, VFS forwards the request to the FUSE layer in the kernel. Subsequently, this layer forwards the request through the standard C libraries to the user space file system library, which contains two parts: standard FUSE functionality (libfuse.so) and the specific user space file system (UsiFe).

The FUSE layer (libfuse.so) parses the request, and depending on the mode of file access parallelizes the request to increase bandwidth, and calls the relevant callback functions in the user space file library. We need to register these user space callback functions for some of the standard file system operations such as open, read, write, getattr, create, unlink, and fstat. Each callback function needs to perform the relevant file system action. In this case, the function needs to interact with a native or virtualized storage device, or a conventional file system loaded on it. The first method is typically used by performance critical applications like file servers and databases that require a custom storage layout. However, in our case, we are only concerned with implementing advanced functionality. Hence, UsiFe uses a standard file system, ext2, as the underlying file system. Nevertheless, we can always strip this layer from UsiFe, and make it a native file system by interfacing it with device drivers for storage devices.

2.2 Interfacing with the Ext2 File System

We use the publicly available ext2fs library [1] to interface with the ext2fs file system. We first define a file system image. We use a regular file to save the image. The ext2fs library saves the master block, and the table of inodes [1] in this image file. In Linux, if the libattr feature is enabled in the kernel, then most common file systems such as Ext(2,3,4), ReiserFS, and XFS support a feature known as extended attributes (*xattr*). Each attribute is a name-value pair. The *name* is a null terminated string, and the *value* is a pointer to a block of data in the file system. In the ext2fs filesystem, there is a predefined attribute data pointer, *i_file_acl*. This is meant to be used to implement sophisticated access control lists. We use this functionality to save encryption and decryption data instead of access control lists. We set i_file_acl to point to a data block that contains UsiFe metadata. If one 4kB block is not sufficient, then we put a pointer to another 4kB block at the end of the n^{th} block to point to the $(n+1)^{st}$ block. In this manner, we chain the UsiFe metadata blocks together.

2.3 Encryption and Decryption

Let us describe the structure of the blocks pointed to by the field i_file_acl in the inode, which contains UsiFe metadata. We assume that a block can be partially encrypted and an encrypted region can potentially span multiple blocks (see Figure 2). To save the co-ordinates of all the encrypted blocks, we need to store the start and end of the range of an encrypted region, and the identifier of the key that the block is encrypted with. We structure the metadata as follows. In the first block, we save the number of blocks in the metadata linked list, and a table containing the list of keys. Every user system is expected to contain a key store, where each key is uniquely identified by a string. The table of keys in the metadata will contain name-value pairs. *name* refers to the name of a key, which can potentially match a key in the system wide key store, and *num* refers to the number with which it is referred to in the file. Subsequently, we store the list of encrypted *regions* in a tabular format.

Each region contains the start and end locations of the encrypted region, along with the numeric identifier of the key. We use a symmetric AES algorithm for encryption. We use a 1024 bit key to encode a known piece of text. To encrypt *n* bytes, we XOR them with the first *n* bytes of the cipher text. In the *display* mode, UsiFe does not try to decrypt regions for which it does not have a valid key in the system wide key store. It just decrypts as much as possible and shows the entire contents of the file including the encrypted content. However, in *stealth* mode, UsiFe only shows those regions that it can successfully decrypt. We show in Section 3 that all glibc calls are completely oblivious to the fact that encrypted regions exist. All fseek, ftell, and rewind calls seamlessly skip them.

3. EVALUATION

3.1 Setup

System		Software	
Processor	Intel Clovertown	Linux Kernel	2.6.38
Frequency	2 Ghz	FUSE Lib.	2.8.6
Memory	3 GB	Ext2Fs Lib.	1.41.14

Table 1. Details Of The Platform

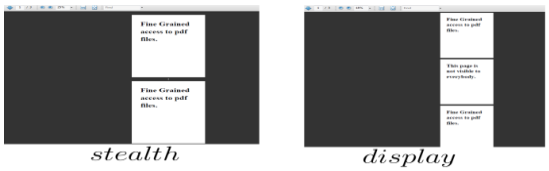

Figure 3. Viewing pdf files in UsiFe

We wrote approximately 4000 lines of C code, and linked the object file with the FUSE and ext2fs libraries.The ext2 file system image was stored in a flat file, whose size was limited to 2 GB. We performed a sanity check on the system by writing small testbenches to test all the regular file access system calls such as open, read, write, creat, close, and seek. Once, these sanity checks passed, we started the process of encryption. Table 1 shows the configuration of the system.

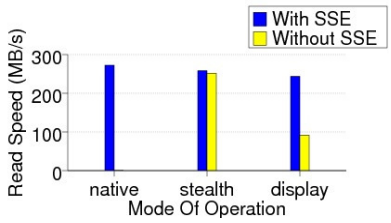

Figure 4. Effect of SSE on read performance

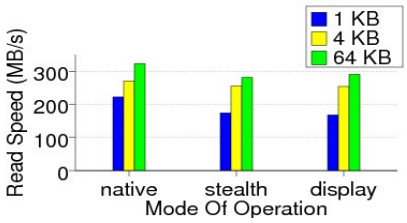

Figure 5. Effect of record-size on read performance

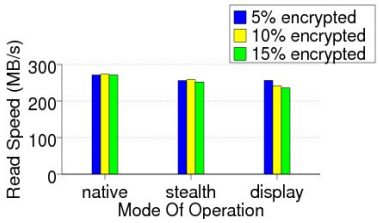

Figure 6. Effect of encryption on read performance

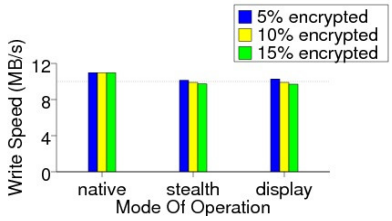

Figure 7. Effect of SSE on write performance

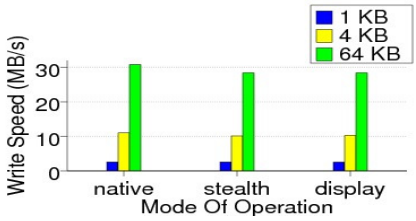

Figure 8. Effect of record-size on write performance

Figure 9. Effect on encryption on write performance

After the sanity checks, we performed the actual set of experiments with the widely used IOzone [5] suite of file system benchmarks. The benchmark measures a wide set of file system features. It uses synchronous I/O, asynchronous I/O, memory mapped I/O and multistream measurement. For encryption, we used the AES algorithm with symmetric 1024 bit keys. We took a piece of plain text, which was known in advance, and generated cipher text, which was 4 kilobytes long. To encrypt or decrypt a block we computed the XOR of the data in the block and the cipher text. We plotted our results for three modes: *native*, *stealth*, and *display*. To further improve the performance of encryption and decryption, we used SSE instructions that are available in all Intel or other X86 based processors. These are SIMD instructions that can perform 4 XOR operations per cycle. We did not require any extra libraries for this. We used the built in support provided by the gcc compiler.

3.2 Results

Figure 3 shows two views of a pdf file opened in Adobe Acrobat, which has one page encrypted. In the first experiment, we assume that we don't have the key for it. In this case, opening the file in *stealth* mode skips the encrypted page. In the second experiment, we assume that we have the key. Now, we open it in *display* mode, and we are able to see the encrypted page.

Figures 4 and 7 show the read and write performance for the three different modes of operation. The read performance decreases from 275 MB/s to 90 MB/s going from *native* to *display* mode. However, we are able to recover most of the lost performance by using SSE instructions. This makes the performance of *display* mode comparable to that with the *native* mode. Please note that for the *native* mode, the choice of using SSE or not is irrelevant. Hence, we just show one bar. We observe in the case of writes, the bottleneck lies at the side of the disk drive. The UsiFe code is fairly effective in splitting requests into multiple threads such that computation can be overlapped with communication. The bandwidth just decreases from 10.9 to 9.6 MB/s.

Figures 5 and 8 show the effect of block size on bandwidth. The default configuration has 4 KB blocks with SSE instructions. We observe that the performance improves by about 40%, when we increase the block size from 1 KB to 64

KB. However, in the case of writes, the bandwidth increases by 10X for the same range of block sizes. This is because we don't need to initiate many transfers to the hard disk for larger block sizes. We save on the seek time and rotational latency. Figures 6 and 9 show the bandwidth as a function of the percentage of blocks encrypted. We observe that the bandwidth decreases by upto 10% for 15% encryption in the *display* mode, which is a minimal overhead. Secondly, we observe that *stealth* and *display* have roughly similar performance with SSE instructions for almost all the configurations.

4. RELATED WORK

[4] is the most closely related to our work. Banachowski et. al. were the first to look at intra file encryption to the best of our knowledge. They made the case for it by studying several commercial and scientific workloads. They modified an existing object based file system to support intra file encryption. They created a new structure called an s-node that stores a list of fields/data ranges in the file object that are encrypted. As compared to their approach, our solution, UsiFe, considers generic filesystems, and works in user space. [7, 8] look at fine grained access control in XML documents. The authors propose to lock nodes, and subtrees, and allow only a certain set of users to access them.They further extend this idea to DTD specifications and schemas. [8] proposes to lock XML subtrees on the basis of structure. If each node satisfies a certain set of rules related to the structure of the tree, then the node may be locked for a certain user. [6] further extends this idea to lock a set of nodes based on the data it contains. Please note that both of these features can be implemented by UsiFe for XML documents.

5. CONCLUSION

This paper presented the design of a novel user space file system that has support for intra-file encryption called *UsiFe*. UsiFe has been designed to be a fast, easy to use, user space file system that uses standard components like the FUSE library (part of the Linux kernel since version 2.6.14), and the ext2 file system. We showed that with existing features of the ext2 file system namely file attributes, we can implement fine grained encryption. Furthermore, we described two file access semantics: *display* and *stealth*. In one mode, the user is aware of the underlying encryption, whereas in the second mode the user is oblivious to it. For each access type, we showed a practical example of using this technology with pdf files. We evaluated the performance of our file system in Section 3. We observed that with less than 15% overhead, we can effectively support fine grained encryption. Moreover, using hardware support like SSE extensions, we showed that the performance overhead of the *display* mode is minimal as compared to *native* or *stealth*.

REFERENCES

[1] Ext2fs libraries, http://e2fsprogs.sourceforge.net

[2] Fuse library, http://fuse.sourceforge.net.

[3] W3C xml security specifications, http://www.w3.org/TR/xml-encryption-req.

[4] S. A. Banachowski, Z. N. J. Peterson, E. L. Miller, and S. A. Brandt, "Intra-file security for a distributed file system," MSST, pages 153–163, 2002.

[5] M. Ben, "Iozone benchmark http://www.iozone.org"

[6] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Sarnarati, "Securing xml documents," Intl. Conf. on Extending Database Technology, pages 121–135, 2000

[7] E. Damiani, S. C. Vimercati, S. Paraboschi, and P. Samarati, "A fine-grained access control system for xml documents," ACM Trans. Inf. Syst. Secur., 5:169–202, May 2002

[8] N. Qi, M. Kudo, J. Myllymaki, and H. Pirahesh, "A function-based access control model for xml databases, International conference on Information and knowledge management, CIKM'05, pages 115–122, 2005.