CamouflageFS: Increasing the Effective Key Length in Cryptographic Filesystems on the Cheap

Michael E. Locasto and Angelos D. Keromytis

Department of Computer Science
Columbia University in the City of New York
{locasto,angelos}@cs.columbia.edu

Abstract. One of the few quantitative metrics used to evaluate the security of a cryptographic file system is the key length of the encryption algorithm; larger key lengths correspond to higher resistance to brute force and other types of attacks. Since accepted cryptographic design principles dictate that larger key lengths also impose higher processing costs, increasing the security of a cryptographic file system also increases the overhead of the underlying cipher.
We present a general approach to effectively extend the key length without imposing the concomitant processing overhead. Our scheme is to spread the ciphertext inside an artificially large file that is seemingly filled with random bits according to a key-driven spreading sequence. Our prototype implementation, *CamouflageFS,* offers improved performance relative to a cipher with a larger key-schedule, while providing the same security properties. We discuss our implementation (based on the Linux Ext2 file system) and present some preliminary performance results. While CamouflageFS is implemented as a stand-alone file system, its primary mechanisms can easily be integrated into existing cryptographic file systems.

"*Why couldn't I fill my hard drive with random bytes, so that individual files would not be discernible? Their very existence would be hidden in the noise, like a striped tiger in tall grass.*" –Cryptonomicon, by Neal Stephenson [17]

1 Introduction

Cryptographic file systems provide data confidentiality by employing encryption to protect files against unauthorized access. Since encryption is an expensive operation, there is a trade-off between performance and security that a system designer must take into consideration. One factor that affects this balance is the key length of the underlying cipher: larger key lengths imply higher resistance against specific types of attacks, while at the same time requiring more rounds of processing to spread the influence of the key across all plaintext bit ("avalanche effect"). This is by no means a clear-cut comparison, however: different ciphers can exhibit radically different performance characteristics (*e.g.,* AES with 128 bit keys is faster than DES with 56 bit keys), and the security of a cipher is not simply encapsulated by its key length. However, given a well designed variable-key length cryptographic cipher, such as AES, the system designer or administrator is faced with the balance of performance *vs.* key length.

M. Jakobsson, M. Yung, J. Zhou (Eds.): ACNS 2004, LNCS 3089, pp. 1–15, 2004.

We are interested in reducing the performance penalty associated with using larger key sizes without decreasing the level of security. This goal is accomplished with a technique that is steganographic in nature; we camouflage the parts of the file that contain the encrypted data. Specifically, we use a spread-spectrum code to distribute the pointers in the file index block. We alter the operating system to intercept file requests made without an appropriate key and return data that is consistently random (*i.e.,* reading the same block will return the same "garbage"), without requiring that such data be stored on disk. This random data is indistinguishable from encrypted data. In this way, each file appears to be an opaque block of bits on the order of a terabyte. There is no need to actually fill the disk with random data, as done in [13], because the OS is responsible for generating this fake data on the fly. An attacker must mount a brute force attack not only against the underlying cipher, but also against the spreading sequence. In our prototype, this can increase an attacker's work factor by 2^{28} without noticeable performance loss for legitimate users.

1.1 Paper Organization

The remainder of this paper is organized as follows. In Section 2, we discuss our approach to the problem, examine the threat model, and provide a security analysis. In Section 3 we discuss in detail the implementation of CamouflageFS as a variant of the Linux Ext2fs, and Section 4 presents some preliminary performance measurements of the system. We give an overview of the related work on cryptographic and steganographic file systems in Section 5. We discuss our plans for future work in Section 6, and conclude the paper in Section 7.

2 Our Approach

Our primary insight is that a user may decrease the performance penalty they pay for employing a cryptographic file system by using only part of the key for cryptographic operations. The rest of the key may be used to unpredictably spread the data into the file's address space. Note that we are not necessarily fragmenting the placement of the data on disk, but rather mixing the placement of the data within the file.

2.1 Key Composition: Maintaining Confidentiality

While our goal is to mitigate the performance penalty paid for using a cryptographic file system, it is not advisable to trade confidentiality for performance. Instead, we argue that keys can be made effectively longer *without* incurring the usual performance penalty. One obvious method of reducing the performance penalty for encrypting files is to utilize a cipher with a shorter key length; however, there is a corresponding loss of confidentiality with a shorter key length. We address the tradeoff between key length and performance by extending the key with "spreading bits," and exploiting the properties of an indexed allocation file system.

A file system employing indexed allocation can efficiently address disk blocks for files approaching terabyte size. In practice, most files are much smaller than this and do

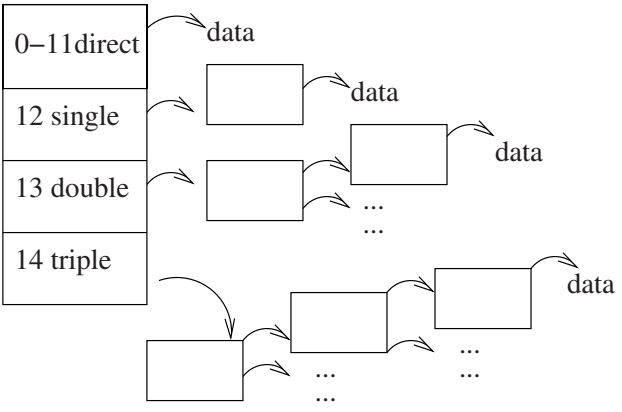

Fig. 1. Outline of a multi-level index scheme with triple-indirect addressing. The first 12 index entries point directly to 12 data blocks. The next three index entries are single, double, and triple indirect. Each indirect block contains 1024 entries: the first level can point to 1024 data blocks, the second level can point to 1024^2, and the third level points to 1024^3 data blocks.

not use their full "address space." The Linux Ext2fs on 32-bit architectures commonly provides an address range of a few gigabytes to just short of two terabytes, depending on the block size, although accessing files larger than two gigabytes requires setting a flag when opening the file [4].

We use the extra bits of the cryptographic key to spread the file data throughout its address space and use the primary key material to encrypt that data. By combining this spreading function with random data for unallocated blocks, we prevent an attacker from knowing which blocks to perform a brute force search on. To maintain this illusion of a larger file without actually allocating it on disk, we return consistently random data on *read()* operations that are not accompanied by the proper cryptographic key.

2.2 Indexed Allocation

In a multi-level indexed allocation scheme, the operating system maintains an index of entries per file that can quickly address any given block of that file. In the Ext2 file system, this index contains fifteen entries (see Figure 1). The first twelve entries point directly to the first twelve blocks of the file. Assuming a block size of 4096 bytes, the first twelve entries of this index map to the first 48Kb of a file. The next three entries are all indirect pointers to sub-indices, with one layer of indirection, two layers of indirection, and three layers of indirection, respectively [4].

Figure 2 shows a somewhat simplified example of a single-level direct-mapped index. The file index points directly to blocks with plaintext data. Holes in the file may exist; reading data from such holes returns zeroed-out blocks, while writing in the holes causes a physical disk block to be allocated. Cryptographic file systems encrypt the stored data, which leaves the index structure identical but protects the contents of the data blocks, as shown in Figure 3.

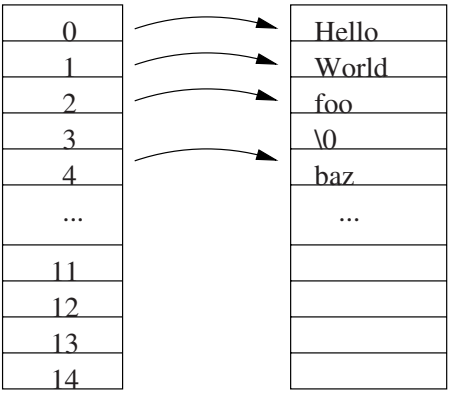

Fig. 2. File index for a normal data file. Pointers to plaintext data blocks are stored sequentially at the beginning of the index. Files may already contain *file holes* – this index has a hole at the third block position.

Usually, most files are small and do not need to expand beyond the first twelve direct mapped entries. This design allows the data in a small file to be retrieved in two disk accesses. However, retrieving data pointed to by entries of the sub-indices is not prohibitively expensive, especially in the presence of disk caches [4].

Therefore, instead of clustering the pointers to file data in the beginning entries of the index, we can distribute them throughout the index. In order for the operating system to reliably access the data in the file, we need some sequence of numbers to provide the *spreading schedule*, or which index entries point to the different blocks of the file. Figure 4 shows encrypted data that has been spread throughout the file's address space.

2.3 Spreading Schedule

The purpose of the *spreading schedule* is to randomly distribute the real file data throughout a large address space so that an attacker would have to first guess the spreading schedule before he attempts a brute force search on the rest of the key.

Normally, the number of the index entry is calculated by taking the floor of the current file position "pos" divided by the block size.

$$index = pos/blocksize$$

This index number is then used to derive the *logical block number* (the block on disk) where the data at "pos" resides.

$$lbn = get\_from\_index(index)$$

This procedure is altered to employ the spreading schedule. The initial calculation of the index is performed, but before the logical block number is derived, a pseudo-random permutation (PRP) function takes the calculated index and the bits of the spreading seed

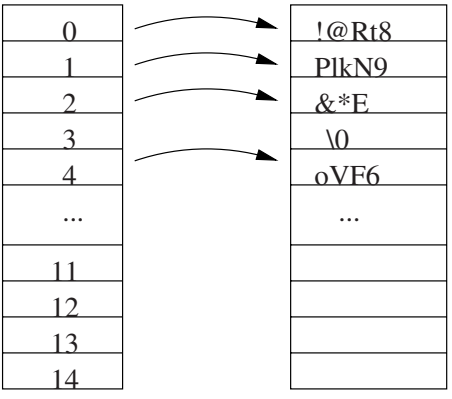

Fig. 3. Index for an encrypted file. The indexing has not changed, merely the contents of the data blocks. Again, the file hole at block three is present.

to return a new index value, without producing collisions. The logical block number is then derived from this new index.

$$index = pos/blocksize$$

$$index = map(index, spread\_seed)$$

$$lbn = get\_from\_index(index)$$

Note that the actual disk block is irrelevant; we are only interested in calculating a new entry in the file index, rather than using the strictly sequential ordering. Given the secret spreading seed bits of the key, this procedure will return consistent results. Therefore, using the same key will produce a consistent spreading schedule, and a legitimate user can easily retrieve and decrypt their data.

2.4 Consistent Garbage

The spreading schedule is useless without some mechanism to make the real encrypted data appear indistinguishable from unallocated data blocks. To accomplish this blending, camouflage data is generated by the operating system whenever a request is made on an index entry that points to unallocated disk space (essentially a file hole). Each CamouflageFS file will contain a number of file holes. Without the key, a request on any index entry will return random data. There is no way to determine if this data is encrypted without knowing the spreading schedule, because data encrypted by a strong cipher should appear to be random in its ciphertext form. We employ a linear congruential generator [11] (LCG) to provide pseudo-random data based on a secret random quantity known only to the operating system. This final touch camouflages the actual encrypted data, and the file index is logically similar to Figure 5. Note that camouflage data is only needed (and created on the fly) when the system is under attack; it has no impact on performance or disk capacity under regular system operation.

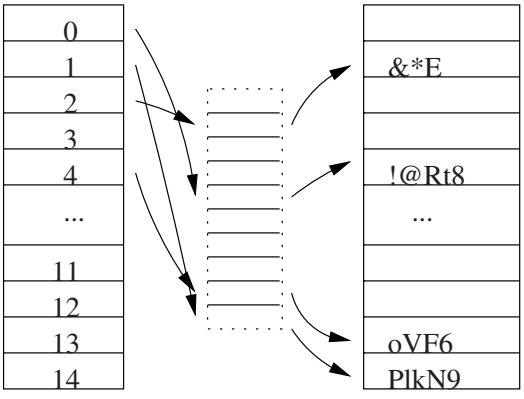

Fig. 4. Index where the entries for the data blocks have been spread. We have created an implicit *virtual index* to spread the file data blocks throughout the file's address space. The file address space is now replete with file holes. Note that it is simple to distinguish the encrypted data from the file holes because the operating system will happily return zeroed data in place of a hole.

2.5 Security Analysis

Threat Model. The threat model is based on two classes of attacker. The first has physical access to the disk (*e.g.,* by stealing the user's laptop). The second has read and write access to the file, perhaps because they have usurped the privileges of the file owner or because the file owner inadvertently provided a set of permission bits that was too liberal. The attacker does not know the secret key (including the spreading bits).

The attacker can observe the entire file, asking the operating system to provide every block. The attacker has access to the full range of Unix user-level tools, as well as the CamouflageFS tool set. The attacker could potentially corrupt the contents of the file, but our primary concern is maintaining the data's confidentiality. Integrity protection can be accomplished via other means.

Mechanism. For the purposes of this analysis, we assume that data would normally be enciphered with a 128 bit key. We also assume that 32 "spreading bits" are logically appended to the key, making an effective key of length 160 bits. Finally, we assume that the cipher used does not have any weakness that can be exploited to allow the attacker a less-than-brute-force search of the key space. Since only the operating system and the user know the 160 bits of the key, anyone trying to guess the spreading schedule would have to generate and test 2^{32} runs of the schedule generator even before they attempt any decryption. Note that if the operating system did not generate camouflage data, the attacker could easily ignore the spreading schedule function and simply grab disk blocks in the file that did not return null data. At this point, the attacker would still have to perform a 2^{128} brute force search on the key space.

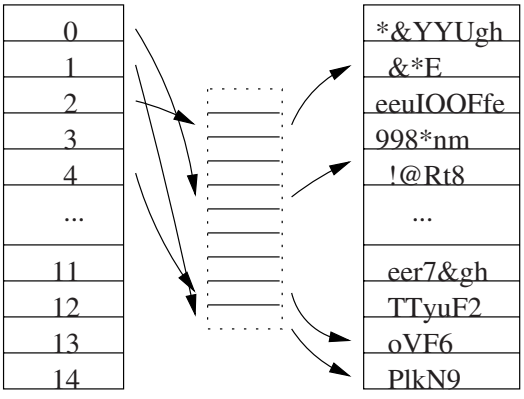

Fig. 5. Index where the data has been spread and camouflaged. Instructing the operating system to return consistent random data instead of zero-filled blocks for file holes effectively camouflages the encrypted data.

Camouflage Synchronization. There are some important issues that must be resolved in order for the generated camouflage data to actually protect the encrypted data. Most importantly, we do not want the attacker to be able to distinguish between the generated camouflage and the real encrypted data. Both sets should appear uniformly random. We assume that the attacker is free to make requests to the operating system to read the entire file. There are two instances of the problem of the camouflage data being "out of sync" with the real file data.

The first instance is that if the same camouflage data is returned consistently over a long period of time, the attacker could surmise that only the parts of the file that actually *do* change are being encrypted and thus correspond to the actual data in the file. This kind of de-synchronization could happen with a frequently edited file.

On the other hand, if the file data remains stable for a long period of time, and we repeatedly update the camouflage data, the attacker could conjecture that the parts of the file that *do not* change are the real data. This type of file could be a configuration file for a stable or long–running service.

These kinds of de-synchronization eliminate most of the benefits of the spreading schedule, because the attacker only has to rearrange a much smaller number of blocks and then move on to performing a search of the key space. In some cases, it may be reasonable to assume that these blocks are only a subset of the file data, but as a general rule, these "hotspots" (or "deadspots") of data (in)activity will stick out from the camouflage.

A mechanism should be provided for updating the composition of the camouflage data at a rate that approximates the change of the real file data. Since we do not actually store the camouflage data on disk, this requirement amounts to providing a mechanism for altering the generation of the camouflage data in some unpredictable manner.

Attacks. First, note that most attacks on the system still leave the attacker with a significant brute force search. Second, we are primarily concerned (as per the threat

model described above) with data confidentiality, including attacks where an intruder has access to the raw disk.

1. An attacker could request the entire file contents and perform a brute force search for the key. This attack is the least rewarding.
2. An attacker may discover the camouflage magic value by reading the i-node information. This would allow the attacker to identify camouflage data. The solution is to encrypt the index portion of the i-nodes with the user's full key, or with a file system-wide key. In either case, the performance penalty would be minimal, due to the small size of the encrypted data.
   Alternatively, we can use a smart card during a user session to allow the OS to decrypt the i-nodes. Recent work on disk encryption techniques [9] discusses various ways to accomplish this goal.
3. An attacker could use a bad key to write into the file, corrupting the data. Two possible solutions are to use an integrity protection mechanism or to store some redundancy in the i-node to check if the provided key correctly decrypts the redundancy. However, these measures act like an oracle to the attacker; failing writes indicate that the provided key was not correct.
4. The attacker could observe the file over a period of time and conjecture that certain parts of the file are camouflage because they do not change or change too often. A mechanism would need to be implemented to change the camouflage seed at the same rate other file data changes.

3 Implementation

CamouflageFS is a rather straightforward extension to the standard Ext2 file system for the Linux 2.4.19 kernel. The current implementation can coexist with normal file operations and does not require any extra work to use regular Ext2 files.

CamouflageFS consists of two major components. The first is a set of *ioctl()*'s through which the user can provide a key that controls how the kernel locates and decrypts camouflaged files. The second component is the set of read and write operations that implement the basic functionality of the system. In addition, a set of user-level tools was developed for simple file read and write operations (similar to *cat* and *cp*) that encapsulate the key handling and *ioctl()* mechanisms.

3.1 LFS: Large File Support

Employing the entire available address range for files is implied in the operation of CamouflageFS. Large File Support [8] for Linux is available in the kernel version of our implementation and requires that our user level utilities be compiled with this support.

The thirty-two bit architecture implementation of Ext2 with LFS and a block size of 4096 bytes imposes a twenty-eight bit limit on our "extension" of a key. This limitation exists because of the structure of the multi-level index (see Figure 1) and the blocksize of 4096 bytes. Since the index works at the block, rather than byte, granularity, the 2^{40} bytes in the file are addressed by blocks of 4096 (2^{12}) bytes, with 4 bytes per index entry.

This relationship dictates a selection of roughly 2^{28} index blocks (so that we do not run into the Ext2 file size limitation of just under 2 terabytes).

The O_LARGEFILE flag is needed when opening a file greater than two gigabytes; this flag and the 64-bit versions of various file handling functions are made available by defining _LARGEFILE_SOURCE and _LARGEFILE64_SOURCE in the source code of the utilities. The utilities are then compiled with the _LARGEFILE_SOURCE and _FILE_OFFSET_BITS flags.

3.2 Data Structures

The first changes to be made were the addition of the data structures that would support the CamouflageFS operations. In order to simplify the implementation, no changes were made to the structure of the Ext2 i-node on disk, so CamouflageFS can peacefully co-exist with and operate on Ext2 formatted partitions.

An unsigned thirty-two bit quantity (i_camouflaged) was added to the in-memory structure for an Ext2 i-node. This quantity served as a flag, where a zero value indicated that the file was not a CamouflageFS file. Any non-zero value indicated otherwise. Once a file was marked as a CamouflageFS file, a secret random value was stored in this field for use in producing the camouflage for the file holes. This field is initialized to zero when the i-node is allocated. A structure was defined for the cryptographic key and added to the file handle structure.

Other changes include the addition of various header files for the encryption and hash algorithms, our LCG operations, additional *ioctl()* commands, and our index entry spreading functions. The actual operation and implementation of these functions are described below.

3.3 Cryptographic Support

CamouflageFS uses the Blowfish encryption algorithm [15] to encrypt each block of data, and can use either SHA-1 or an adaptation of RC6 during the calculation of the spread index entries. Code for these algorithms is publicly available and most was adapted for use from the versions found in the Linux 2.5.49 kernel.

3.4 Command and Control

The *ioctl()* implementation for Ext2 was altered to interpret five new commands for controlling files that belong to CamouflageFS. The two most important commands are:

1. EXT2_IOC_ENABLE_CAMOUFLAGE is a command that marks a file as being used by CamouflageFS. When a file is marked as part of the CamouflageFS, a random number is extracted from the kernel entropy pool and stored in the i_camouflaged field of the i-node. This has the dual effect of marking the file and preparing the system to return random camouflage data in place of file holes.
2. EXT2_IOC_SHOW_KEY_MATERIAL is the primary command for interacting with the file once it has been marked as a CamouflageFS file. This command is accompanied by a key structure matching the one described above and is used during subsequent read or write operations on the file handle. Note that the supplied key could be incorrect; at no time is the genuine key stored on disk.

3.5 User Tools and Cryptographic Support

Several user-level tools were developed to aid in the use of the system. These tools primarily wrap the ioctl() commands and other routine work of supplying a key and reading from or writing to a file. A userland header file (*cmgfs.h*) is provided to define the *ioctl()* commands and the file key structure.

The *read()* and *write()* operations for Ext2 were augmented to use the provided key if necessary to decrypt or encrypt the file data, respectively. Each page was encrypted or decrypted as a whole. Before a write could succeed, the page needed to be decrypted, the plaintext added at the appropriate position, and then the altered page data encrypted and written to disk.

3.6 Index Mapping

A variable length block cipher is utilized as a pseudo-random permutation (PRP) to map sequential block indices to ostensibly random indices. The underlying concept and justification for the variable length block cipher construction of which the implementation in CamouflageFS is a particular instance is beyond the scope of this paper. While only the 28-bit PRP implemented for CamouflageFS is briefly described here, it should be noted the variable length block cipher can be built upon any existing block cipher and stream cipher. RC6 was chosen for this implementation because its construction makes it applicable to small block sizes and RC4 was utilized due to its simplicity.

The PRP is an unbalanced Feistel network consisting of the RC6 round function combined with initial and end of round whitening. RC4 is used to create the expanded key. The PRP operates on a 28-bit block split into left and right segments consisting of 16 bits and 12 bits, respectively. The RC6 round function is applied to the 16-bit segment using a word size of 4 bits. The number of rounds and specific words swapped after each round were chosen such that each word was active in 20 rounds, equally in each of the first four word positions.

While the current mapping of block indices cannot be considered pseudo-random in theory, because the maximum length of an index is restricted to 28 bits in the file system and thus an exhaustive search is feasible, the use of a variable length block cipher will allow support for longer indices when needed.

3.7 Producing Camouflage Data

Camouflage data is produced whenever an unallocated data block is pointed to by the file index. If the block is part of a hole and the file is camouflaged, then our LCG is invoked to provide the appropriate data.

In order to avoid timing attacks, whereby an attacker can determine whether a block contains real (encrypted) or camouflaged data based on the time it took for a request to be completed, we read a block from the disk before we generate the camouflage data. The disk block is placed on the file cache, so subsequent reads for the same block will simulate the effect of a cache, even though the data returned is camouflage and independent of the contents of the block that was read from disk.

Finally, notice that camouflage data is only produced when an attacker (or curious user) is probing the protected file — under regular use, no camouflaged data would be produced.

4 Performance Evaluation

To test the performance of the system, we compared three implementations of Ext2. The first implementation was the standard Ext2. The second implementation modified Ext2 to use the Blowfish algorithm to encrypt data inside the kernel. The third implementation was CamouflageFS and incorporated our techniques along with encryption under Blowfish. In all cases, performance (measured by the amount of time to read or write a file) is largely dependent on file size. Execution time was measured with the Unix *time(1)* utility; all file sizes were measured for ten runs and the average is recorded in the presented tables.

The primary goal of our performance measurements on the CamouflageFS prototype is to show that the work necessary for a brute force attack can be exponentially increased without a legitimate user having to significantly increase the amount of time it takes to read and write data files, which is shown in Figure 6.

file size (kb)	ext2 R	ext2 W	BF R	BF W	cmgfs R	cmgfs W
1	0.002	0.001	0.003	0.001	0.003	0.001
21	0.01	0.001	0.010	0.002	0.010	0.002
42	0.02	0.001	0.020	0.003	0.003	0.004
63	0.03	0.001	0.030	0.004	0.004	0.005
210	0.09	0.002	0.094	0.012	0.206	0.148
2107	0.8395	0.008	0.930	1.096	1.319	1.105
21070	8.371	0.071	9.305	11.019	9.851	11.047
84280	33.5	55.17	37.180	65.416	37.756	67.493

Fig. 6. Time to read and write various size files in our various ext2 file system implementations. All times are in seconds (s).

Using a longer key contributes to the performance penalty. Most notably, a longer key length is achieved in 3DES by performing multiple encrypt and decrypt operations on the input. This approach is understandably quite costly. A second approach, used in AES-128, simply uses a number of extra rounds (based on the keysize choice) and not entire re-runs of the algorithm, as with 3DES. Blowfish takes another approach, by effectively expanding its key material to 448 bits, regardless of the original key length. The performance impact of encryption (using Blowfish) on ext2fs is shown in the second set of columns in Figure 6.

Therefore, we want to show that CamouflageFS performs nearly as well as ext2 *read()* and *write()* operations that use Blowfish alone. Using our prototype implementation, the performance is very close to that of a simple encrypting file system, as shown in

Figure 6. However, we have increased the effective cryptographic key length by 28 bits, correspondingly increasing an attacker's work factor by 2^{28}.

The CamouflageFS numbers closely match the performance numbers for a pure kernel-level Blowfish encryption mechanism, suggesting that the calculation of a new index has a negligible impact on performance. For example, the performance overhead (calculated as an average over time from Figure 7) of Blowfish is 11% for *read()* operations and 17% for *write()* operations. CamouflageFS exhibits essentially the same performance for these operations: 12% for *read()*'s and 22% for *write()*'s.

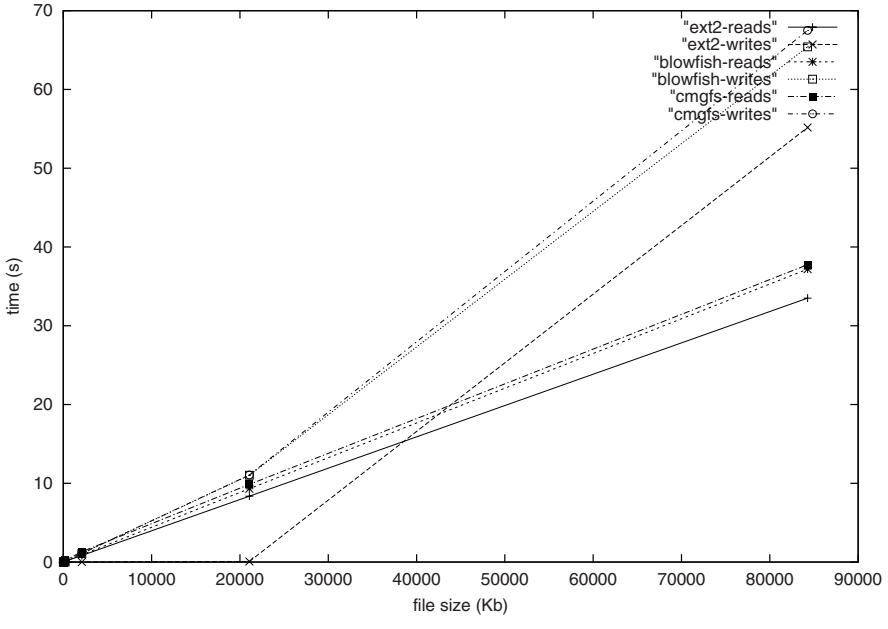

Fig. 7. Comparison of ext2 reads and writes versus CamouflageFS. CamouflageFS closely matches a file system that only performs encryption.

5 Related Work

The work presented in this paper draws on a number of research areas. Most notably, the recent work in information hiding and steganographic file systems serves the similar goal of hiding sensitive data. Our technique, on the other hand, combines steganography with the encryption mechanisms used by traditional cryptographic file systems to improve performance without the related cost.

5.1 Cryptographic File Systems

Most related efforts on secure file systems have concentrated on providing strong data integrity and confidentiality. Further work concentrates on making the process transparent or adjusting it for network and distributed environments. The original Cryptographic File System (CFS) [3] pointed out the need to embed file crypto services in the file system because it was too easy to misuse at the user or application layers.

Cryptfs [18] is an attempt to address the shortcomings of both CFS and TCFS [5] by providing greater transparency and performance. GBDE [9] discusses practical encryption at the disk level to provide long-term cryptographic protection to sensitive data.

FSFS [12] is designed to deal with the complexities of access control in a cryptographic file system. While the primary concern of CamouflageFS is the speedup of data file encryption, file system access control mechanisms are another related area that benefits from applied cryptography.

The Cooperative File System [6], like the Eliot [16] system are examples of file systems that attempt to provide anonymity and file survivability in a large network of peers. The Mnemosyne [7] file system takes this cause a step further, based on the work presented in [1], to provide a distributed steganographic file system.

5.2 Information Hiding

Information hiding, or steganography, has a broad range of application and a long history of use, mainly in the military or political sphere. Steganographic methods and tactics are currently being applied to a host of problems, including copyright and watermarking technology [14]. The survey by Petitcolas, Anderson, and Kuhn [14] presents an excellent overview of the field. Anderson [2] constructs a background for steganographic theory as well as examining core issues in developing steganographic systems.

Recently, the principles of information hiding have been applied to creating steganographic file systems that provide mechanisms for hiding the existence of data.

5.3 Steganographic File Systems

Steganographic file systems aim to hide the presence of sensitive data. While some implementations merely hide the data inside other files (like the low–order bits of images), other systems use encryption to not only hide the data, but protect it from access attempts even if discovered. This hybrid approach is similar to CamouflageFS.

StegFS [13,1] is one such steganographic file system. The primary goal of StegFS is to provide (and in some sense define) legal plausible deniability of sensitive data on the protected disk, as proposed and outlined by Anderson *et al* [1]. Unfortunately, using StegFS's strong security results in a major performance hit [13]. StegFS is concerned with concealing the location of the disk blocks that contain sensitive data. In short, StegFS acts as if two file systems were present: one file system for allocating disk blocks for normal files, and one file system for allocating blocks to hidden files using a 15 level access scheme. The multiple levels allow lower or less-sensitive levels to be revealed under duress without compromising the existence of more sensitive files.

Each of these two file systems uses the same collection of disk blocks. Normal files are allowed to overwrite the blocks used for hidden file data; in order to protect the hidden files, each block of a hidden file is mapped to a semi-random set of physical blocks. Since each disk block is initialized with random data, the replication makes the sensitive data appear no different than a normal unallocated disk block while ensuring that the hidden data will survive allocation for normal files.

6 Future Work

The work presented here can be extended to other operating systems and file systems. For example, OpenBSD provides a wide array of cryptographic support [10]. Further work includes performing standard file system benchmarks and implementing AES as a choice of cipher.

Beyond this work, there are two primary issues to be addressed: preventing both collisions in the spreading schedule and an attacker's discernment of camouflage data.

The use of a variable length block cipher to calculate the virtual index should address the possibility of collisions; however, as noted previously, the length should be increased to lessen the possibility of a brute force attack. The length of 28 bits in our implementation is an architecture and operating system limitation.

To prevent an attacker from knowing which data was actually camouflage, we would have to create some mechanism whereby the i_camouflaged field is updated at some rate to "stir" the entropy source of the camouflage data.

Further work includes both examining the feasibility of various attack strategies against the system and discovering what effect (if any) the spreading schedule has on the placement of data on disk. There should be little impact on performance here; the virtual index is relatively independent of what disk blocks contain the data.

7 Conclusions

CamouflageFS is a simple, portable, and effective approach to improving data confidentiality in cryptographic file systems. The approach taken is to hide the encrypted data in an artificially large file, using a key-driven spread-spectrum sequence. Attackers must guess both the cryptographic key and the spreading key, effectively increasing their work factor. Appropriate measures are taken to prevent an attacker from determining which disk blocks contains encrypted data. The performance impact of the technique to legitimate users is negligible.

We intend to investigate further applications of this *practical* combination of steganographic and cryptographic techniques for improving security in other areas.

References

1. R. Anderson, R. Needham, and A. Shamir. The Steganographic File System. In *Information Hiding, Second International Workshop IH '98*, pages 73–82, 1998.
2. R. J. Anderson. Stretching The Limits of Steganography. In *Information Hiding, Springer Lecture Notes in Computer Science*, volume 1174, pages 39–48, 1996.

3. M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, November 1993.
4. D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Management*. O'Reilly, second edition, 2003.
5. G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic File System For Unix. Technical report, July 1997.
6. F. Dabek, F. Kaashoek, R. Morris, D. Karger, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of ACM SOSP*, Banff, Canada, October 2001.
7. S. Hand and T. Roscoe. Mnemosyne: Peer-to-Peer Steganographic Storage. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.
8. A. Jaeger. Large File Support in Linux, July 2003.
9. P.-H. Kamp. GBDE - GEOM Based Disk Encryption. In *BSDCon 2003*, September 2003.
10. A. D. Keromytis, J. L. Wright, and T. de Raadt. The Design of the OpenBSD Cryptographic Framework. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
11. D. Lehmer. Mathematical Methods in Large-scale Computing Units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery*, pages 141–146. Harvard University Press, 1949.
12. S. Ludwig and W. Kalfa. File System Encryption with Integrated User Management. In *Operating Systems Review*, volume 35, October 2001.
13. A. D. McDonald and M. G. Kuhn. Stegfs: A Stegonographic File System for Linux. In *Information Hiding, Third International Workshop IH '99*, pages 463–477, 2000.
14. F. A. Petitcolas, R. Anderson, and M. G. Kuhn. Information Hiding–A Survey. In *Proceedings of the IEEE, special issue on protection of multimedia content*, volume 87, pages 1062–1078, July 1999.
15. B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.
16. C. Stein, M. Tucker, and M. Seltzer. Building a Reliable Mutable File System on Peer-to-peer Storage.
17. N. Stephenson. *Cryptonomicon*. Avon Books, 1999.
18. E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.