Encrypted Watermarks and Linux Laptop Security

Markku-Juhani O. Saarinen

Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O. Box 5400, FIN-02015 HUT, Finland
mjos@tcs.hut.fi

Abstract. The most common way to implement full-disk encryption (as opposed to encrypted file systems) in the GNU/Linux operating system is using the encrypted loop device, known as CryptoLoop. We demonstrate clear weaknesses in the current CBC-based implementation of CryptoLoop, perhaps the most surprising being a very simple attack which allows specially watermarked files to be identified on an encrypted hard disk without knowledge of the secret encryption key.

We take a look into the practical problems of securely booting, authenticating, and keying full-disk encryption. We propose simple improvements to the current CryptoLoop implementation based on the notions of tweakable encryption algorithms and enciphering modes. We also discuss sector-level authentication codes. The new methods have been implemented as a set of patches to the Linux Kernel series 2.6 and the relevant system tools.

1 Introduction

Perhaps the most typical approach for protecting the confidentiality and authenticity of files is to use PGP or other similar encryption tools which allow users to encrypt files for transmission and storage. Explicit decryption is required before a file can be modified. After plaintext file is no longer needed, it must be re-encrypted and securely deleted (wiped). This can be cumbersome, so more transparent methods have been devised. These can be categorized into encrypting file systems and sector-level encryption.

Encrypting file systems. These generally allow flexible control over which directories are encrypted and which are not. The main problem with encrypting file systems is that temporary files containing sensitive information are often stored on unencrypted portions of the disk, such as swap devices / page files, various caches, or “temp” directories. Examples of encrypting file systems are CFS [5] and TCFS [7] in the UNIX world, and Microsoft’s EFS.

Sector-level encryption. These systems implement encryption below the file system level, and allow entire hard disks to be encrypted. Sector-level encryption systems do not usually allow fine-grained access control for files. The whole volume is protected with a single master key (many options for managing and storing this key exist). File system ("upper layer" from our viewpoint) accesses data as *sectors*. In our terminology a sector is a logical unit consisting of 512 bytes (4096 bits); larger physical sectors must be split into 512-byte pieces. We use this convention regardless of the actual sector size used by the file system (typically 1024, 2048 or 4096 bytes in the case of EXT2 [6]).

Since the file system must be able to perform quick single sector random access reads and writes on the disk, each sector must be "independent" of others; no other data is needed for encryption and decryption than the sector itself, sector identifier, and the secret key.

Sector-level encryption is often the preferred choice in cases where the hard disk has little physical protection (e.g. with personal portable computers). The Linux CryptoLoop implements sector-level encryption, as does SFS [12] and many commercial systems for the Microsoft OS platforms.

Linux CryptoLoop. CryptoLoop is a facility provided by the Linux Kernel to easily integrate encryption below the file system level. It works regardless of the file system used. With CryptoLoop, a physical disk drive, a disk drive partition, or a container file is "looped" as a loop device (`/dev/loop0`, `/dev/loop1`, ...). After a key is provided to the Kernel using the `losetup` utility, the loop device driver transparently takes care of encryption and decryption whenever the loop device is accessed. The loop device can then be initialized and mounted using any file system. Figure 1 illustrates the call dependencies of CryptoLoop.

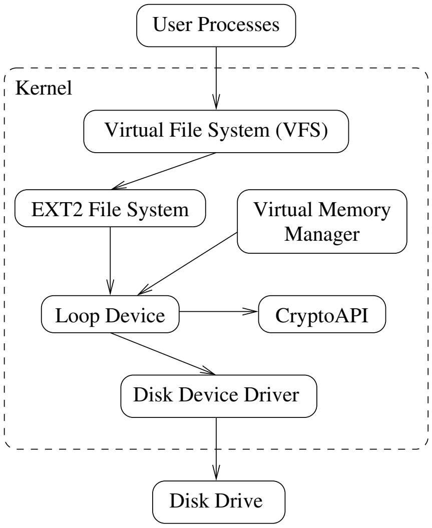

Fig. 1. CryptoLoop call structure

Since encryption and decryption is always done on independent 512-byte sectors, we call such a transformation *Sector Enciphering Operation* (SEO). SEO and its inverse can be characterized as follows:

$$C = \text{SEO}(P, K, T)$$
$$P = \text{SEO}^{-1}(C, K, T)$$

Here P is the plaintext sector of 512 bytes (4096 bits), K is the secret key, T is the tweak (sector number), and C is the corresponding ciphertext sector.

Efficiency considerations usually dictate that SEO is length-preserving. Information-theoretic arguments can be used to show the impossibility of proper message authentication in such a case. We will return to these issues in section 5.

Structure of this paper. In section 2 we illustrate some attacks against the present Linux implementation of CryptoLoop. We believe the "watermarking" technique in section 3 to be novel. In section 4 we introduce a practical security model for sector level-encryption (this is fundamentally the same as that proposed by Halevi and Rogaway in [14, 13] and implicitly by others before them [8]), and discuss various ways to achieve the goal set by the model. Section 5 discusses sector-level message authentication and makes an argument for not having sector-level authentication but rather authenticating at file system level. Section 6 contains some of our experiences and practical thoughts about implementation of sector-level encryption. Section 7 contains performance analysis of the implementation, and is followed by concluding remarks in section 8.

2 Attacks

Currently the Linux 2.6 series offers a selection of SEOs constructed from various well-known block cipher algorithms. Mode of operation can be chosen to be either ECB or CBC. We ignore the obviously weak ECB mode and concentrate on CBC, which is absolutely the most commonly used choice.

CBC is currently initialized for each sector by using the sector number directly as the initialization vector. This convention is also used at least by 2.2, 2.4 series of Linux kernels, and by versions of Jari Ruusu's loop-AES package that were published prior to December 2003 [23].

The 512-byte plaintext sector P is split into blocks $P = p_1 \mid p_2 \mid p_3 \cdots$. The corresponding ciphertext is $C = c_1 \mid c_2 \mid c_3 \cdots$. Encryption of plaintext block x with a secret key K is denoted by $E_k(x)$. Hence the SEO becomes:

$$\begin{aligned}
c_1 &= E_k(p_1 \oplus T) \\
c_2 &= E_k(p_2 \oplus c_1) \\
c_3 &= E_k(p_3 \oplus c_2) \\
&\cdots \\
c_{32} &= E_k(p_{32} \oplus c_{31})
\end{aligned}$$

Here we assume that a 128-bit block cipher is used. In the case of 64-bit block ciphers, the last plaintext and ciphertext blocks would naturally be p_{64} and c_{64}, correspondingly. Here T is the Tweak and is equal to the logical sector number.

Standard attacks. In some conditions it is conceivable that an encrypted disk will be subject to repeated scans and even active manipulation based on such scans (typical scenario for such scans is during international travel – customs and baggage checks).

Replay attacks on CBC mode in CryptoLoop have been known for years (many were explicitly pointed out by J. Etienne [10]), yet it has persisted as the dominant disk-encryption system for Linux. We give some well-known examples:

1. **Corrupt.** Corruption of chosen data blocks is difficult to detect. As CBC decryption has little error propagation, modifying a ciphertext block within sector will only corrupt the corresponding plaintext block (8 or 16 bytes) and cause chosen bit changes the one block immediately following it.
2. **Move.** It is easy to shift multiple ciphertext blocks anywhere within the hard disk. Only the first plaintext block will be corrupted. This opens a big toolbox of “cut & paste” attacks.
3. **Revert.** It is possible to revert chosen sectors to their previous values without detection. An attacker can from two ciphertext images detect where the changes lie and choose the sectors to be reverted accordingly.

Together these options (possibly with the aid of watermarkings – see next section – to provide “location data”) allow subtle and powerful manipulation of the hard disk on ciphertext level.

3 Encrypted watermarks

We introduce the concept of *encrypted watermarks*, which are subtle markings on arbitrary plaintext files. The existence of these markings (and hence the existence of files that contain them) can be proven on the corresponding ciphertext without knowledge of the secret key. [1] Encrypted watermarks provide a powerful investigative technique for a forensic analyst who wishes to identify files such as restricted documents, pornography or “warez” on encrypted hard drives.

We recall a couple of basic facts that apply to most UNIX disk file systems, including EXT2, EXT3, ReiserFS, UFS, XFS, etc.

a) File data is mostly ($P > 0.99$) stored on consecutive sectors on the disk.[2]
b) File data starts at an even multiple of the sector length.

Simple watermarks. In CBC Linux CryptoLoop, we may create simplest kinds of watermarks in files by making the first block of two consecutive sectors differ only in the least significant bit. There is a significant probability that the corresponding ciphertext blocks will be equal, and hence can be identified with high certainty from the ciphertext alone (without knowledge of the secret key).

[1] The concept of *encrypted watermark* is an extension of watermarking as understood in the context of Digital Rights Management.

[2] This is not necessarily true for FAT file systems, which are much more prone to fragmentation.

Advanced watermarks. This idea can be extended in many ways, as long as the XOR -difference between the plaintext sectors' first plaintext blocks can be matched with the XOR -difference of their sector numbers. The probability is affected by the actual sector size and fragmentation properties of the file system above the loop device, but experiments have shown the probability to be very high with typical EXT2-based setup.

It is possible to devise more reliable and elaborate watermarking schemes by basing the markings on particular patterns of first-block collisions within a file. A file can contain multiple complementary watermarks. Such a marking (and file) can be located on an encrypted device with high certainty. Since the majority of the file is unchanged, such markings are stealthy and easy to insert into jpeg pictures, mp3 music files, ps/pdf files, program binaries etc.

Example (with a 64-bit block cipher):

```
0x00000  Any data ..
0x13000  Eight bytes 0000000000000000
0x13008  Any data ..
0x13200  Eight bytes 0000000000000001
0x13208  Any data ..
0x13400  Eight bytes 0000000000000000
0x13408  Any data ..
```

This file contains a watermark that will be visible in the encrypted device, as long as data at positions 0x13000 .. 0x135ff are contained in three consecutive sectors on the physical disk.

Proof. Let n, $n+1$, and $n+2$ be the sectors containing this section of the file. We note that for any n either $n \oplus (n+1) = 1$ or $(n+1) \oplus (n+2) = 1$. Hence either $E_K(0 \oplus n) = E_K(1 \oplus (n+1))$ or $E_K(1 \oplus (n+1)) = E_K(0 \oplus (n+2))$. The watermark can thus be detected as the same ciphertext block can be found at the beginning of two consecutive ciphertext sectors (regardless of location).

4 A security model

Following the language of [13, 14, 18, 19], we want SEO to be a *strong*, *tweakable*, *pseudorandom permutation* (PRP); for a random key we wish SEO (and its inverse) to be indistinguishable from a random permutation.

Furthermore, we wish SEO to be resistant to various attacks based on key scheduling and tweaks. Equivalent keys, related keys, and other “weak key” classes (if they exist) should be computationally difficult to find. There should be no shortcut attacks based on chosen, weak, or related tweaks.

In short, there should be no analytic attack which is computationally cheaper than exhaustive search through the keyspace, regardless of the amount of chosen plaintext and/or ciphertext (with associated tweak values) available. If these conditions are violated by an attack, SEO can be considered broken.

Encryption Modes. We have found certain commercial disk encryption systems to utilize CTR and even ECB modes for encryption, both of which offer clearly unsatisfactory resistance to attacks. ECB for obvious reasons, and CTR in the case that multiple scans are made (all plaintext changes made between the scans will become visible when the two ciphertext disk images are "xorred" together).

Recently Halevi and Rogaway proposed the CMC and EME modes for this purpose [13, 14]. These remain unbroken. We observe that EME, CMC, and all other satisfactory modes for sector-level encryption are in fact *double-encryption modes*, and hence offer roughly half the speed of conventional modes such as CBC.

Special tweakable block ciphers. As the use of provably secure sector encipherment modes appears to result a significant performance penalty (due to beforementioned double-encryption), crafting a custom 4096-bit block cipher seems justified.

This motivated us to design a prototype cipher named Herring [22]. Herring was designed with the explicit purpose of satisfying all the security requirements outlined above, while also maintaining encryption speed comparable to AES in "single - encryption" mode. Herring accepts a 128-bit key and a 128-bit tweak. Conclusions about the security of Herring can be drawn only after years of public analysis. We wish to make Herring public soon, although the findings of this paper are valid for any 4096-bit tweakable block cipher.

Some other 4096-bit block ciphers have been constructed explicitly for the purpose of sector level encryption. One proposal was the Mercy block cipher [8], which was broken by Fluhrer [11]. Another 4096-bit block cipher present in the literature is the unnamed proposal of Kaliski and Robshaw [17], which was found to be weak by Saarinen [21].

Wide-block block ciphers have also been constructed from other primitives. An interesting approach was taken by Anderson and Biham with the BEAR and LION ciphers, which combine a hash function with a stream cipher to produce a block cipher [2].

5 Why no authentication at sector level?

Even if SEO satisfies the criteria given in the previous section, it is still prone to some replay attacks (e.g. reverting and corruption of sectors). We have considered a number of approaches for the sector-level authentication problem but we have not found satisfactory solutions. Different options include:

a) Check everything at mount time using a signature algorithm. This is clearly preventive, since it would involve both reading the entire hard disk during mount time and correspondingly signing it when unmounting (assuming that it was mounted read-write).

b) Incorporating MACs with the sectors. By including the sector number in the authenticated data, this makes other modifications other than reversion attack (where sector n is reverted to its previous contents) detectable.

   If implemented on loop device level, this would make the boundaries between physical sectors and logical sectors incompatible. A read operation on a single isolated sector would always imply two physical sector reads. A write operation would imply two sector reads and two sector writes. The performance drop would therefore be significant.

c) Maintain a table of MACs in memory, which is loaded during mount time and stored (and signed) in a file when the disk is unmounted. This is a reasonably implementable approach, especially if we use the actual file system sector size and a truncated MAC. Using HMAC-SHA1-64 [4] with sector size 4096 bytes corresponds to two megabytes of MACs for each gigabyte of disk (0.195%), which is manageable.

d) Dynamically maintain a Hash Tree of the sectors of the disk. The memory requirements are similar to the previous option, but an advantage exists in that the hash of the whole disk can be maintained at all times, thus perhaps allowing better recovery from crashes. We do not see this as a very effective solution.

e) Dynamically maintain a "sum" of MACs of each sector:

$$S = \bigoplus_{i=1}^{n} \mathrm{MAC}_K(i \mid P_i)$$

Here MAC_K is a keyed MAC with secret K and P_i is the sector with logical number i. It is plain that a sector write becomes a sector read-write in this option, but that performance penalty is acceptable. Even though read, write, mount, and unmount operations are quick and memory usage is acceptable, the drawback is that in order to *detect* changes (i.e. to find out that the sum doesn't match), the whole disk needs to be read and this doesn't even tell us where the change occurred!

Of these approaches, maintaining a table of MACs in memory (option C) appears to be the most feasible one, but it alone doesn't really provide a satisfactory solution for error recovery. It is nice to know that an error has occurred but what can you do with this information? In cryptographic communication protocols the connection is typically simply shut down if "MACs don't match", but making the whole disk unusable after a single bit error or operating system crash is not acceptable.

Therefore perhaps the most important question is to decide on how to respond to an anomalous situation, i.e. when the authentication code doesn't match. Mechanisms do not exist in the current Linux architecture for the loop device to communicate to a file system layer that the error may be a security issue rather than a simple disk error (such sectors will simply be marked "bad"). And how could such a determination be reached?

If an error is found (after a crash or unclean shutdown) and the encrypted disk is `fsck`'ed, the recovered file system may be perfectly healthy and the ill effects cancelled. Therefore it is reasonable to incorporate authentication at file system level.

Practical approach. Since CryptoLoop cannot really protect against all attacks that modify ciphertext in the disk, we recommend regular use (e.g. by a `cron` mechanism) of systems such as Tripwire (see `www.tripwire.org` or `www.tripwire.com`), which can detect malicious changes to files. It is noteworthy that these tools will not only detect physical attacks when a computer is at unauthorized hands, but also many network-based attack vectors (esp. "backdooring") while the computer is being used by its authorized user.

6 Implementation

We can see three different methods of implementation for a SEO.

a) A software implementation as a part of the operating system kernel. This is the easiest option, but will cause a performance penalty in disk-intensive applications. Such an implementation may also utilize cryptographic hardware speedups via DMA.
b) Hardware support in the disk controller or the hard drive itself. After keying the disk controller or the drive itself will transparently encrypt and decrypt everything. This would essentially make the encryption process independent of the operating system itself.
c) Implementation as a "bump in the cable" on the (IDE/ATA or SCSI) cable. This method of implementation would be easy to integrate into existing systems, but it may end up being more costly than hardware support in the controller.

We have only experimented with the first option, but plans exist for a FPGA-based hardware implementations as well.

Our current implementation. The required modifications to the present Linux 2.6 series kernel source were minor. The patch contains about 3000 lines of code, majority of it being the Herring cipher. The patch also adds support for a new mode of operation and "tweak" keying required for sector-level encryption. The needed changes can therefore be implemented in the framework of current Linux cryptographic support.

Perhaps surprisingly, no changes were required for `losetup` or other v.2.12 utilities that are used in setting up and keying the loop devices. A script was written which requests a passphrase from the user, hashes it (with salting), and passes it on to `losetup`. More elaborate key management and authentication methods would be easy to add. We had also to create small statically linked versions of these utilities so that they fit into the initial ram disk, discussed below.

Booting. One of the trickiest things about encrypted Linux laptops is to come up with a reasonably secure boot procedure. Since we're using standard hardware, it is almost impossible to come up with a "bullet-proof" solution, as the hardware itself can be modified to include key loggers or similar intercept devices. It is not within the scope of this paper to discuss methods for preventing hardware modification in any detail, but some level of inexpensive protection can be achieved by simple seals and manual inspection of hardware after suspected modification.

As a rule of thumb, one would like to have as little as possible of the computer hard disk to be in plaintext. Some practioners have decided to keep the "static" parts of the hard disk unencrypted (e.g. `/bin`, `/sbin`, `/usr`, but not `/etc`, `/var` or `/home`). It is relatively easy to backdoor such systems with standard rootkit tools.

Currently our best practice for booting is as follows:

1. **Firmware setup.** We are in practically forced to use standard BIOS firmware. We enable the available security features (BIOS setup passwords etc.), and disable booting from other devices than the main hard disk.

2. **Boot loader.** The boot loader resides on the Master Boot Record on the hard disk. We use the LILO boot loader in "simple mode", so that it will directly load up the compressed kernel image and initrd (initial ram disk) using BIOS routines.
3. **Boot partition.** Since we saw little purpose in incorporating encryption into the LILO itself, we are forced to keep the kernel image and the initrd unencrypted in a separate partition. These require a total of 2.5 MB of space.
4. **Initrd.** The initial ram disk contains a small file system that is first mounted as root. A passphrase prompt becomes visible within 5 seconds after power-on. Other authentication mechanisms can also be easily incorporated into the initrd environment (which roughly corresponds to single-user mode). A master key is derived from the passphrase and used to set up the loop device. The encrypted loop device is then mounted as the new root.

   At this stage a "rescue system" can also be activated, which contains the following items (all in ram disk):

   a) **Diskwipe.** A program for quickly wiping the contents of entire hard disk. The wipe procedure exceeds the requirements set in DoD 5220.22-M standard ("sanitize" method D for non-removable rigid disks, p. 8-3-5 [9]).
   b) **Convert.** A tool for re-encrypting the hard disk using an alternative master key or cipher algorithm.
   c) **Busybox.** A small shell-like environment for rescue purposes.[3]

   There is no access control to reach the rescue system.
5. **Boot verification.** In the first stages of (encrypted) boot, digital signatures of the kernel image and the initrd image are verified at the boot partition. The corresponding signatures and public keys reside on the encrypted partition of the disk are thus difficult to forge (in fact it would be sufficient to check their message digests against known values).

   Also an attempt is made to verify the integrity of firmware by comparing it to known digest values with physical memory image accessible through `/dev/ram`. However, not all of the firmware binary appears to be visible at this point.

We acknowledge that this boot procedure is not wholly secure against all software-based attacks, although such attacks would seem to require non-trivial human effort. One possible attack would involve crafting modifications to the kernel so that it is able to maintain the appearance that everything is going smoothly (by returning false values to system calls) while also containing a trojan horse for capturing and transmitting the master encryption key. A reasonable amount of obscurity and variation in the implementation details guards against such attacks (obscurity is unfortunately the only method available to resist against this class of attacks). [4]

Alternative boot and keying methods. We have also experimented with other boot sequence options. Small, lightweight, and inexpensive USB solid state memory devices have become available in recent years. Many BIOS Firmware vendors allow booting

[3] Busybox was not developed as a part of this research effort. See `www.busybox.net`

[4] Trusted Computing Group is an emerging industry standard that will provide mechanisms to detect BIOS and MBR manipulation (among other things) and therefore to improve the security of early steps of booting. See `www.trustedcomputinggroup.org`

from these devices. It is relatively straight-forward to include the boot loader, kernel image, and initial ram disk into such a device, and thus allow 100 % of the hard disk to be encrypted, the unencrypted portion being part of one's keychain! Similar approach can be taken with CD-ROMs, albeit they are not as easily transportable. USB tokens and various smartcard systems are easily incorporated into the initial ram disk phase of booting. There is no need to store the key on the token itself, so even national ID cards which support public key decryption may be used (e.g. FINEID in case of Finland). For yet another approach see AEGIS [3] and its smartcard - based extension sAEGIS [15].

Encrypting swap. Linux has full support for encrypted swap devices. Even easier solution is to use a swap file which resides on the same encrypted partition with the main file system. [5]

Other implementations. To our knowledge, tweakable modes have not been previously implemented in the Linux kernel.

The BestCrypt for Linux 1.5.1 [16] package from Jetico Inc. uses IV with CBC mode in a similar (although not entirely compatible) fashion as current CryptoLoop, and hence is vulnerable to the attacks described in this paper. [6]

TCFS [7] appears to either use ECB mode (before version 3) or CBC mode with zero IV (after version 3), and is vulnerable to similar attacks.

Matt Blaze's Cryptographic File System [5] utilizes a combination of OFB and ECB modes, but is also vulnerable to attack. Peter Guttman's Secure File System (SFS) used MDC/SHS, a special construction which turns the SHA hash function into a block cipher in CFB mode. This does not satisfy our security requirements either. The construction based on MD5 (MDC/MD5) was shown to be weak by Saarinen [12, 21].

We have also evaluated several commercial disk encryption systems for which source code or full specification is not publicly available. Generally speaking, most commercial sector encryption products do not appear to offer effective protection against watermarking / multiple scanning attacks. [7]

7 Performance

We measured the speed of read and write operations on a typical modern PC laptop, Acer TravelMate 420, which has a 2 gHz Pentium 4 CPU and ATA disks.

The performance was measured with one gigabyte (2^{30} bytes) continuous reads and writes in single-user mode using `dd` (i.e. data transfers from the loop device to `/dev/null` and from `/dev/zero` to the loop device). No special optimization was

[5] It appears that at the time of writing all mainline Linux versions may have deadlock problems with encrypted swap. We hope that this will be fixed soon.

[6] Since the Linux version is compatible with the Windows versions of BestCrypt from the same vendor, we believe these to be also vulnerable.

[7] Since some of our security analysis methods may be interpreted as reverse engineering or disclosure of trade secrets and thus violation of certain local laws, we are restricted in discussing these results.

used, and caches were disabled. The operating system was our custom modified Linux 2.6.1.

The following table summarizes our measurements.

Encryption algorithm	Read MB/s	Write MB /s
None	14.8	14.7
AES-128 CBC	13.0	14.6
Herring	11.0	10.2
AES-128 EME	7.3	9.7

Implementation of AES and CBC were the the standard ones in Linux 2.6 kernel. The AES implementation is based on the work by Brian Gladman and is reasonably optimized. The implementation of EME mode [14] was by the author, with sector size 512 bytes (hence 65 AES block operations per sector, compared to 32 required by CBC). The implementation of the preliminary version of Herring was also by the author. All the implementations are in portable C language without assembly optimizations. There is room for performance improvement.

It should also be noted that laptop hard drives are somewhat slower than those drives now common on desktop machines. However, we feel confident in concluding that full sector-level encryption does not present a significant performance bottleneck for day-to-day computer use.

8 Concluding remarks

Most government agencies and many large corporations have security policies in place which make encryption of hard disks mandatory for laptops. Easiest and most transparent method of achieving such protection is by using sector-level encryption, which leaves as little as possible of the disk unencrypted.

Disk encryption is not a performance bottleneck nor does it significantly decrease the usability of the system. Therefore there are few excuses for not deploying it where ever possible.

Careful analysis has shown that many products and techniques widely used for sector-level encryption are vulnerable to active manipulation and even watermarking; the presence of (planted) restricted data can be detected without breaking the encryption key.

However, good solutions can be reached with limited resources and open software. Sector-level encryption also offers a good motivation for research into very wide-block tweakable block cipher designs and tweakable enciphering modes.

9 Acknowledgements

The author would like to thank the anonymous program committee members, Jari Ruusu, and various active members of the Linux Kernel Mailing List for valuable comments. This work was supported by Finnish Defence Forces Research Institute of Technology.

References

1. *Specification for the Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication 197, 2001.
2. R. Anderson and E. Biham. *Two Practical and Provably Secure Block Ciphers: BEAR and LION*. Proc. Fast Software Encryption '96, LNCS 1039, Springer-Verlag, 1996. pp. 113–120.
3. W. A. Arbaugh, D. J. Farber, and J. M. Smith. *A Secure and Reliable Bootstrap Architecture* Proc. 1997 IEEE Symposium on Security and Privacy, ACM Press, 1997. pp. 65–72.
4. M. Bellare, R. Canetti, and H. Krawczyk. *Keying Hash Functions for Message Authentication*. Proc. Crypto 1996, LNCS 1109, Springer-Verlag, 1996.
5. M. Blaze. *A Cryptographic File System for Unix*. Proc. First ACM Conference on Computer and Communications Security, Fairfax, VA, 1993.
6. R. Card, T. Ts'o, Stephen Tweedie. *Design and implementation of the Second Extended Filesystem*. In Frank B. Brokken et al, editor, Proc. of the First Dutch International Symposium on Linux, Amsterdam, December 1994.
7. G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. *The Design and Implementation of a Transparent Cryptographic File System for UNIX*. USENIX Annual Technical Conference '01, Freenix Track. 2001.
8. P. Crowley. *Mercy: A Fast Large Block Cipher for Disk Sector Encryption*. Proc. Fast Software Encryption 2000, LNCS 1978, Springer-Verlag, 2000, pp. 49–63.
9. Department of Defense. *Industrial Security Manual for Safeguarding Classified Information,* Department of Defense Manual, DoD 5220.22-M, June 1987.
10. J. Etienne, *Vulnerability in encrypted loop device for Linux*, Manuscript available from `http://www.off.net/~jme/`, 2002.
11. S. R. Fluhrer. *Cryptanalysis of the Mercy Block Cipher*. Proc. Fast Software Encryption 2001, LNCS 2355, Springer-Verlag, 2002, pp. 28–36.
12. P. C. Gutmann. *SFS Version Documentation.* `http://www.cs.auckland.ac.nz/~pgut001/sfs/`
13. S. Halevi and P. Rogaway. *A Tweakable Enciphering Mode.* Proc. Crypto '03, LNCS 2729, Springer-Verlag, 2003.
14. S. Halevi and P. Rogaway. *A Parallelizable Enciphering Mode.* To appear in Proc. RSA Conference 2004 – Cryptographer's Track. Springer-Verlag, 2004.
15. N. Itoi, W. A. Arbaugh, S: J. Pollack, and D. M. Reeves. *Personal Secure Booting.* Proc. ACISP 2001, LNCS 2119, Springer-Verlag, 2001, pp. 130–144.
16. Jetico Inc. *BestCrypt for Linux v.1.5.1 with Linux 2.6 support*. available from `http://www.jetico.com`, 2004.
17. B. S. Kaliski and M. J. B. Robshaw. *Fast Block Cipher Proposal*. Proc. Fast Software Encryption 1993, LNCS 0809. Springer-Verlag, 1994, pp. 33 – 40.
18. M. Liskov, R. L. Rivest, and D. Wagner. *Tweakable Block Ciphers*. Proc. CRYPTO 2002, LNCS 2442, Springer-Verlag, 2002, pp. 31–46.
19. M. Luby and C. Rackoff. *How to construct Pseudorandom Permutations from Pseudorandom Functions*. SIAM J. of Computation, 17(2), April 1988.
20. R. L. Rivest. *All-Or-Nothing Encryption and The Package Transform*. Proc. Fast Software Encryption '97, LNCS 1267, Springer-Verlag, 1997, pp. 210–218.
21. M.-J. O. Saarinen. *Cryptanalysis of block ciphers based on SHA-1 and MD5*. Proc. Fast Software Encryption 2003. LNCS. Springer-Verlag, 2004.
22. M.-J. O. Saarinen. *Herring: A Tweakable Block Cipher for Sector level Encryption*. Manuscript (to be submitted for publication), 2004.
23. J. Ruusu, *Loop-AES Source and Documentation*. `http://loop-aes.sourceforge.net/`