When Cryptography Meets Storage

Sarah M. Diesburg
Florida State University
105-A Love Bldg
Tallahassee, FL 32306
diesburg@cs.fsu.edu

Christopher R. Meyers
Florida State University
105-A Love Bldg
Tallahassee, FL 32306
meyers@cs.fsu.edu

David M. Lary
Florida State University
105-A Love Bldg
Tallahassee, FL 32306
lary@cs.fsu.edu

An-I Andy Wang
Florida State University
265 Love Bldg
Tallahassee, FL 32306
awang@cs.fsu.edu

ABSTRACT

Confidential data storage through encryption is becoming increasingly important. Designers and implementers of encryption methods of storage media must be aware that storage has different usage patterns and properties compared to securing other information media such as networks. In this paper, we empirically demonstrate two-time pad vulnerabilities in storage that are exposed via shifting file contents, in-place file updates, storage mechanisms hidden by layers of abstractions, inconsistencies between memory and disk content, and backups. We also demonstrate how a simple application of Bloom filters can automatically extract plaintexts from two-time pads. Further, our experience sheds light on system research directions to better support cryptographic assumptions and guarantees.

Categories and Subject Descriptors

E.3 [**Data Encryption**]: Code Breaking, Standards; K.4.1 [**Computers and Society**]: Public Policy Issues – *Privacy;* K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Security.

Keywords

Block ciphers, bloom filters, modes of encryption, storage, two-time pads.

1. INTRODUCTION

As the cost of storage rapidly declines, more and more sensitive data are stored on media such as hard disks, CDs, and flash drives. Inevitably, confidentially plays an increasingly important role in protecting sensitive data from theft and leakage due to unauthorized access, viruses, system penetration, physical loss [33], and improper disposal [15, 2, 41].

1.1 General Encryption

The most widely-used mechanism to achieve confidentiality (of data in general) is through encryption. Ideally, each message is encrypted with a random unique key to achieve perfect secrecy; real-life implementations are far from perfect. Therefore, to avoid identical messages encrypted using the same key resulting in the same encrypted message, initialization vectors (IVs) are introduced to seed the encryption process.

Generally, applying encryption to communication shares the following characteristics:

- Short-lived data streams (e.g., messages)
- Write-once content (e.g., transactions)

Given the short-lived and write-once nature of communication, the uniqueness of keys and IVs can be probabilistically achieved by first cycling through a very large IV space before changing to a new key. (Of course, the assumption here is that the communication infrastructure is largely stateless.)

1.2 Implications of Applying Encryption in Storage

At a quick glance, storage is analogous to a communication channel through time, in the sense that the sender sends the message to a persistent storage medium, and the receiver can later retrieve the message from the medium. Therefore, the same cryptographic mechanism should be applicable. Unfortunately, the usage patterns of storage are different from those of communications in fundamental ways.

- **In-place updates**: Unlike ephemeral communication messages, a file is persistent. Thus, an update to a file can be performed in-place (i.e., old content is overwritten with the new content at the same file location). Therefore, if keys and IVs are generated as a function of data positions within a file or storage medium, the uniqueness of keys and IVs relative to data content is compromised.
- **Content shifting**: In addition to in-place updates, content can be inserted into a file, resulting in the shifting of original content. Therefore, potentially a large quantity of original plaintext is encrypted via reusing the keys and IVs defined as a function of file and disk locations.
- **Backups**: Backups are often considered a problem domain orthogonal to confidential storage. Unfortunately, naïve users may rely on encryption without using proper secure backup schemes. For example, byte-to-byte image dumps of storage enable a passive form of "dead forensic" attack, where an attacker can simply collect different versions of backups, which violates the uniqueness of IVs and keys. This form of attack can be formidable, since an attacker at the archival site can potentially extract the plaintexts with neither access to keys nor user account information.

  Another problem occurs when rolling back more than one version of backup. Should the generation process of unique

International Workshop on Storage Security and Survivability, October 27-31, 2008, Alexandra, Virginia, USA.

IVs and keys be deterministic after the restore point, all subsequent updates with different content will reuse the same IVs and keys used for content stored in previous backups.

Additionally, the storage data path, which bridges user-space applications and the physical storage media, contains many states, with legacy design choices that are incompatible with the goal of data confidentiality.

- **Single generic data type**: A storage data path generally does not discern encrypted data from non-encrypted data. Therefore, sensitive data may not be handled properly (e.g., cached in plaintext). This is commonly referred to as the data lifetime problem [16, 8, 7].
- **Poor consistency guarantees**: Even if encrypted at all times, a newer, uncommitted version of an encrypted file may reside in memory while an older version of the file may reside on a disk. Therefore, operating system mechanisms such as swapping and hibernation can lead to different versions of encrypted data stored on disks, potentially reusing the keys and IVs.
- **Information hiding**: Many storage data path components only provide logical views of the underlying storage and may not honor the intent of the security measures imposed above. For example, old and newly encrypted data may coexist, even though a user application can only see the latest version. Should an attacker have access to the raw storage device, the attacker may find blocks pertaining to older file versions, potentially encrypted with reused keys and IVs.

With these different usage patterns and infrastructures, we can identify encryption methods that are used in communications but vulnerable in storage.

1.3 Contributions and Non-contributions

The two-time-pad problem [27] describes how a *key* is reused to encrypt two plaintexts *P* and *P'*, where $P \oplus key$ and $P' \oplus key$ can be XORed to recover $P \oplus P'$. Although XOR-based stream-cipher attacks are well-known, these weaknesses have not stopped storage designers from using fast encryption modes to support efficient random in-place updates without re-encrypting the remaining file after the updated file location. The fundamental assumption is that the randomness and uniqueness of keys and IVs, relative to encrypted content, can be largely achieved, but that, as we will demonstrate, can be very difficult in modern storage systems.

Another reason for recurring two-time-pad storage solutions is that the ease with which automation can extract plaintexts is doubtful without advanced linguistic expertise and the general availability of such tools. We demonstrate that English plaintexts can be automatically extracted from two-time pads via the simple application of a Bloom filter [3].

This paper is not about rediscovering the XOR-based stream-cipher attacks, nor is it meant to criticize particular storage encryption systems. The fact that systems with such problems exist shows a lack of cross-dissemination between the cryptography and storage arenas. Therefore, we hope that this paper can empirically show what can go wrong when cryptography and storage constraints collide. In particular, we will demonstrate various attacks that leverage vulnerabilities exposed via shifting file contents, in-place file updates, backups, storage mechanisms hidden by layers of abstractions, and inconsistencies between memory and disk content.

The paper is written to be as self-contained as possible, so that both storage and security researchers can be exposed to the primitives and constraints from the other field.

2. BACKGROUND

2.1 Block Cipher Modes of Operation

The operating mode of an encryption algorithm allows block ciphers to output messages of arbitrary length or turns block ciphers into self-synchronizing stream ciphers, which generate a continuous key stream to produce ciphertexts of arbitrary length. For example, using AES alone, one may only input and output blocks of 128 bits. Using AES with a mode of operation for a block cipher, one may input and output data of any length.

The most common modes of operation for block ciphers include electronic codebook (ECB) mode, cipher-feedback (CFB) mode, cipher-block-chaining (CBC) mode, output-feedback (OFB) mode, and counter (CTR) mode. Table 1 lists the various modes of operation, along with the corresponding encryption and decryption specifications. *E* and *D* stand for encryption and decryption respectively. *C* is the ciphertext; *P*, the plaintext; and *O*, the temporary output.

Table 1. Block cipher modes of operations.

Mode of operation	Encryption/decryption
ECB	$C_i = E_{key}(P_i); P_i = D_{key}(C_i)$
CFB	$C_i = E_{key}(C_{i-1}) \oplus P_i, C_0 = IV$ $P_i = E_{key}(C_{i-1}) \oplus C_i, C_0 = IV$
CBC	$C_i = E_{key}(P_i \oplus C_{i-1}), C_0 = IV$ $P_i = D_{key}(C_i) \oplus C_{i-1}, C_0 = IV$
OFB	$C_i = P_i \oplus O_i; O_i = E_{key}(O_{i-1}), O_0 = IV$ $P_i = C_i \oplus O_i; O_i = E_{key}(O_{i-1}), O_0 = IV$
CTR	$C_i = E_{key}(IV \oplus CTR_i) \oplus P_i$ $P_i = E_{key}(IV \oplus CTR_i) \oplus C_i$

2.2 Vulnerabilities of Certain Block Cipher Modes of Operation

A common problem with stream ciphers is that generating two ciphertexts with the same key and IV can leak information about both original plaintexts. A stream cipher operates by producing a key stream based on a key and IV. The plaintext is then XORed to the stream to produce the ciphertext. When two ciphertexts created with the same key and IV are XORed together, the key stream is canceled out, and the result is the XOR of the two original plaintexts.

Some block cipher operation modes can behave similarly to a stream cipher. For example, the CFB, OFB, and CTR modes all create a mask based on a key and IV, and the ciphertext block in question is created by XORing the mask together with the plaintext. If two versions of the ciphertext are found to have been created by the same key and IV, the mask can be canceled out to generate the XOR of the two plaintexts.

We formally demonstrate a vulnerability, which involves two known versions of blocks of ciphertext (C and C') that share identical keys and IVs in certain block cipher modes of operation. The vulnerability occurs when $C \oplus C' = P \oplus P'$, where P and P' are the plaintext versions of the block. Once the attacker has $P \oplus P'$ of the block in question, the attacker may employ various methods to extract plaintext, as discussed later in the paper.

The following sub-sections demonstrate the vulnerability with the following modes of operation: CFB, CTR, and OFB. Our examples may be used with many common block ciphers as the encryption algorithm used to create the mask (such as DES, 3-DES, and AES). These examples also explore a single encrypted block of text, but could easily be extended to address entire files.

2.2.1 CFB

In cipher feedback mode, a ciphertext block is encrypted by XORing the current plaintext block with the previous ciphertext block (i.e., $C_i = E_{key}(C_{i-1}) \oplus P_i$, where $C_0 = IV$). If we refer $E_{key}(C_{i-1})$ as the key mask, M, then we have $C_i = M \oplus P_i$. Similarly, by updating P_i to P_i' in-place, we have $C_i' = E_{key}(C_{i-1}) \oplus P_i'$, or $C_i' = M \oplus P_i'$, assuming that the key and IV remain unchanged. Thus, by XORing C_i and C_i' on the left-hand side, we have the following right-hand side: $M \oplus P_i \oplus M \oplus P_i'$, or $P_i \oplus P_i'$. With CFB, the scope of vulnerability is limited to the current in-place updated block, since the key masks for subsequent blocks differ (i.e., $C_{i+1} = E_{key}(C_i) \oplus P_{i+1}$, while $C_{i+1}' = E_{key}(C_I') \oplus P_{i+1}$). Another implication is that an in-place update can cause the remaining file to be re-encrypted.

2.2.2 CTR

In counter mode, a block is encrypted via XORing a plaintext block with a key mask produced by encrypting content based on a per-block unique counter (i.e., $C_i = E_{key}(IV \oplus CTR_i) \oplus P_i$). If we refer $E_{key}(IV \oplus CTR_i)$ as the key mask, M, we can see that $C_i = M \oplus P_i$, similar to the CFB case. An in-place update from P_i to P_i' yields $C_i' = E_{key}(IV \oplus CTR_i) \oplus P_i'$, or $C_i' = M \oplus P_i'$. With the IV and counter unchanged by the in-place update, it is easy to see that $C_i \oplus C_i' = P_i \oplus P_i'$.

Unlike CFB, the counter mode supports random access, where an in-place update does not require the remaining file to be re-encrypted. However, since the key stream for the entire file can be reused, different versions of the file can be XORed to reveal changed information. This vulnerability is particularly pronounced, given that a common file operation is content insertion, which can cause a significant portion of the original content to be shifted. In this situation, CTR mode leaks information about the plaintext beginning with the first changed block and potentially ending with the last block in the file or extent. While it can be challenging to extract original information from XORed plaintexts, the knowledge of content shifting can significantly improve the chance of extracting the plaintexts.

2.2.3 OFB

In output-feedback mode, a ciphertext block is generated via XORing the plaintext with the encryption of the previous key mask (i.e., $C_i = E_{key}(O_{i-1}) \oplus P_i$, $O_0 = IV$). In this case, the key mask is $E_{key}(O_{i-1})$, or M; therefore, $C_i = M \oplus P_i$. An in-place update from P to P' yields $C_i' = M \oplus P_i'$, and one can see that again $C_i \oplus C_i' = P_i \oplus P_i'$.

Both the support for random access and vulnerability characteristics of the OFB mode is the similar to the counter mode in the storage context, since the key stream is based on only the original key and IV. This key stream can be pre-generated and can stay the same, even if a block is modified. Like CTR mode, the OFB mode leaks information about the plaintext from the first changed block to potentially the last block in the file or extent.

3. CRYPTANALYSIS METHODS

With different versions of data encrypted with the same key and IV under CFB, CTR, or OFB mode, we may perform cryptanalysis on the resulting $C_i \oplus C_i'$, which is just $P_i \oplus P_i'$ with an entropy that is likely to be significantly lower than that of a random data stream. As proof of this concept, we prototyped a utility to extract XORed English texts.

3.1 Common Methods

Many methods exist to separate plaintexts from $P_i \oplus P_i'$. For example, if one of the plaintexts is known apriori, the other plaintext is easily extractable. Often times, an attacker may guess a probable plaintext string in P_i and use a "dragging crib" method [40] to XOR the probable plaintext against every position in P_i' to detect intelligible text. An attacker could also use language-specific heuristics (e.g., average word length and word frequency) to solve to $P_i \oplus P_i'$ [9].

Another method to extract XORed plaintexts is based on letter frequency analysis. Depending on the file type (e.g., English plain text, source code, or word processing document), a frequency table of each character's occurrence in a representative training set can be tallied. For example, several such tabulations exist for letters in the English language [1, 12, 28, 39]. With these statistical distributions, emphasis can be given to those XORed character pairs that contain high frequency characters, while enumerating all possibilities.

N-gram analysis [18, 27] can be used in conjunction with frequency analysis or on its own. Like frequency analysis, lists of n-character strings, or n-grams, can be tallied from a representative training set. Then, while enumerating all possibilities, emphasis may be given to texts that appear frequently in the tally.

3.2 Automated Extraction of XORed Plaintexts

We prototyped a DecodeXOR utility to extract XORed English plaintexts based on the concept of N-grams. However, our simple prototype extracted plaintexts only based on constraints imposed by XORs and texts seen in the training set. We did not track the frequency of N-gram occurrences, build Markov and hidden Markov models, or apply dynamic programming methods such as the Viterbi algorithm [42]. The predominant algorithm used was hashing. Unlike many studies, we made no restrictions on the absence of punctuations, capital and small case letters, numbers, and extended ASCII characters.

DecodeXOR consists of three design components: (1) n-gram table representation and construction, (2) solving plaintext substrings under various constraints, and (3) data structures for tracking and assembling candidate plaintext substrings.

3.2.1 N-gram Table Representation and Construction

The program takes in a training file, in our case, `enwik8` [24], which consists of 100MB of (mostly) English content from random web pages. To capture all 2-grams in the file with characters $\{p_0, p_1, \dots p_{100 \times 2^{20} -- 1}\}$, we hashed all two consecutive characters into a bitmap, where the $hash(p_i, p_{i+1})^{\text{th}}$ bit is set to 1 to indicate the possible transition from p_i to p_{i+1}; 0, otherwise.

Collisions are possible and allowed. Therefore, it is possible to explore letter transitions not present in the training set. However, out of 256 x 256, or 64K possible extended ASCII character transitions, only 18% of edges are used in our training set. Therefore, we can reduce the collision rate to an arbitrary threshold (in our case < 1%) by increasing the number of hash bins.

The method used to capture 2-grams was extended to capture 3- to 6-grams. Collisions are also allowed. To verify that two 6-letter strings are identical, all possible substrings of 2- to 5-grams also need to be checked. Fortunately, knowing that the training file is processed sequentially, an optimization can be applied to check only the 2- to 5-grams containing the last letter of a 6-letter string.

One concern is the memory size requirement. For the 6-gram case, 2^{23} edges (out of 2^{40} possible) are present in our training set, and we allocated 2^{28} hash bins, which can be represented with 32MB of memory if each bin is represented by one bit. Also, hash tables for 2- to 6-grams can be collapsed into a single table, where various hashing mechanisms can be consolidated into a Bloom filter [3], where five hash functions are based on different n-gram lengths.

3.2.2 Solving Plaintext Substrings

Without the frequency information, DecodeXOR had to solve XORed plaintexts based on the constraints of individual letters. Formally, given a stream $\{c_0, c_1, \dots\}$ created by XORing the corresponding letters in plaintext stream1 $\{p_0, p_1, \dots\}$, and plaintext stream2 $\{p_0', p_1',\dots\}$, the candidate plaintexts need to conform to the following constraints for each 7-character XORed substring $\{c_i, c_{i+1}, \dots, c_{i+6}\}$:

1. $p_j \oplus p_j' = c_i$, for all $i \leq j \leq j+6$
2. $\{p_i, p_{i+1}, \dots, p_{i+5}\}$, $\{p_i', p_{i+1}', \dots, p_{i+5}'\}$, $\{p_{i+1}, p_{i+2}, \dots, p_{i+6}\}$, and $\{p_{i+1}', p_{i+2}', \dots, p_{i+6}'\}$ are legitimate 6-grams.
3. The last five characters of $\{p_i, p_{i+1}, \dots, p_{i+5}\}$ need to match the first five characters of $\{p_{i+1}, p_{i+2}, p_{i+3}, \dots, p_{i+6}\}$, same for p' substrings.

Interestingly, we allowed the analysis for the entire extended ASCII character set due to the second constraint. If we disallow certain characters, the second constraint may not be satisfied.

3.2.3 Tracking and Assembling Candidate Plaintext Substrings

For each plaintext solving window of 7 characters (XORed substring), DecodeXOR can identify many 6-character candidate plaintext substrings that satisfy constraints listed in Section 3.2.2. To track and eventually assemble the final plaintexts, we need to have a dedicated data structure.

DecodeXOR uses a hash table to track 6-character candidate substrings. Its key design is the hash function. For each candidate 6-character substring, only the last 5 characters are used for hashing, to determine the storage location of the 6-character substring. In other words, for a given 6-character substring, the hash of the first 5 characters points to the hash bin location of the previous candidate substring with the last 5 characters matched. (Hash collisions are resolved via a linked list.) Therefore, when the decoding process reaches the last 6 characters, a series of hash operations will connect various candidate substrings to form the final plaintext string.

3.2.4 Observations and Limitations

Although this decoder is simple, written in C, with only 363 semicolons, it is sufficient to be used to demonstrate various two-time-pad-related vulnerabilities in storage. DecodeXOR can process the 100MB training file in 5 minutes (a single pass) on a 3Ghz Pentium® D with 2GB of RAM, and decode short XORed strings in seconds.

Of course, this naïve decoder has ample room for enhancements. First, as with any decoder, our ability to decode relies heavily on the training data set. Second, the XOR of two lower-case letters are the same as capital letters. Therefore, in certain cases, determining the capitalization at the beginning of a sentence requires a higher level of language processing [21]. Third, decoding XORed numbers is problematic. Again, a higher level semantic process is required. Fourth, we did not take advantage of possible content shifting, which can further enhance the decoding ability due to additional decoding constraints.

4. REAL STORAGE EXAMPLES

This section demonstrates how real storage mechanisms can turn the original intended one-time pads into two-time pads. We need to reiterate that the intent of this paper is not to criticize particular implementation; rather, it aims to show that (1) although not necessarily straightforward, these attacks can be materialized, and (2) the scope of the problem proliferates throughout the storage data path, ranging from high-level file systems and memory management to low-level device management. First, we use a widely used file system to demonstrate the problems with in-place updates and shifting content. We then explore the issue of inconsistency in memory and disk content via hibernation and demonstrate how, through entropy analysis and the DecodeXOR tool, we can extract newer versions of encrypted data from swap. Additionally, we illustrate how storage layer abstractions, such as wear-leveling applied to flash storage can lead to this vulnerability. Finally, we show how backups can cause reuses of keys and IVs in an all-or-nothing secure deletion system.

4.1 File System Encryption

CryptoFS [20] is a file system that takes advantage of the Linux Userland FileSystem (LUFS) kernel module [26], which allows a file system to be built in user space without having to write any kernel-level code. CryptoFS adds cryptographic functionality in a layer above an underlying file system (e.g., ext3). Encrypted file names and data are stored in a regular directory. This directory becomes accessible as plaintext by mounting to a special unencrypted directory after a user provides the password.

CryptoFS uses `libgcrypt`, a general-purpose cryptographic library based on GnuPG [17], and supports the symmetric ciphers AES, DES, Blowfish, CAST5, Twofish, and Arcfour. The message digest algorithm, the granularity of encryption per IV (e.g., extent size), and number of unique IVs (i.e., default 256) are user-configurable. The message digest algorithm produces the encryption key from a passphrase. Files are divided into extents, and CFB mode is used within each extent to support faster random access times. The extent size is configurable, although it is recommended to be the disk block size (usually 4096 bytes in Linux). Initialization vectors are generated as *the disk block number % number of IVs*. Therefore, the ciphertext will repeat after *number of IVs x extent size* bytes if the same data is being encrypted. The CFB mode is hardcoded and cannot be changed without changing the source code.

Our test system is a Debian Sarge VMware virtual machine running a 2.4.27-3-386 kernel. We installed the `lufs-cryptofs-0.3.1-1.1` package from the main stable Debain repository [10]. We also downloaded and installed `lufs-source-0.9.7-6` to make the `lufs` kernel module.

We began by creating a mount point `/crypt` and a regular ext3 directory `/root/secrets`, which is later designated to store encrypted content. We then used a special mount command to mount `/root/secrets` over `/crypt`:

```
>lufsmount cryptofs:///root/secrets /crypt
Enter password:
>
```

This special mount operation allows us to create encrypted content under `/root/secrets`. However, while the directory is mounted, we can access the content in plaintexts via the `/crypt` path. Next, we created a file under `/crypt` called `secret.txt`, containing the string: "*Now is the time for all good people to worry about their privacy.*" The directory `/root/secrets` now holds a corresponding file called `TdedtcxtXL5j5g==`, which is the encrypted name for `secret.txt`.

After we unmounted CryptoFS, the plaintext is no longer accessible, and `/crypt` appears to be empty. However, suppose the owner of the encrypted file makes a backup copy of `/root/secrets/TdedtcxtXL5j5g==` to a removal medium but leaves the copy sitting around due to the confidence of encryption. Then, an attacker can just secretly make copies of different versions of the encrypted file over time. With two different versions of the ciphertext, the attacker can perform $C_i \oplus C_i'$ and run DecodeXOR to extract plaintexts.

To illustrate information leaks due to both in-place updates and content shifting, we inserted a space at the beginning of the file to generate the second version of the encrypted file. Due to unchanged IVs, in-place updates imply that we can extract 128 bits (or 16 bytes) of information from the updated block (Figure 1).

Additionally, due to the way CryptoFS handles encryption in extents, every extent is associated with a predictable, unchanging, per-extent IV. Therefore, once the content starts to shift, an attacker can decrypt the first 128 bits (or 16 bytes) of subsequent extents after the content insertion point. Thus, as an attacker accumulates different versions of the same encrypted file over time, more information can be revealed due to both in-place updates and content shifting.

$C_i \oplus C_i'$

```
6e 21 18 57 49 1a 53 54 1c 0d 45
54 1d 04 08 45 ea 71 93 52 8a 7d
```

$P_i \oplus P_i'$

```
6e 21 18 57 49 1a 53 54 1c 0d 45
54 1d 04 08 45 46 09 1d 52 41 0d
```

Figure 1. XOR of ciphertext compared to XOR of plaintext in hexadecimal from file `secret.txt`.

With our simple DecodeXOR, we were able to extract five possible XOR pairings of 16-byte English plaintexts that can generate the same ciphertext (Table 2). Should DecodeXOR leverage additional information such as content shifting, the fourth pair will be favored over others.

Table 2. Possible pairings of 16-byte English plaintexts that generate the same XORed ciphertext.

1st XOR pair	`Now is the tied`
	` Now is the tale`
2nd XOR pair	`Now is the tied.`
	` Now is the talk`
3rd XOR pair	`Now is the time`
	` Now is the timb`
4th XOR pair	`Now is the time'`
	` Now is the time`
5th XOR pair	`Now is the time,`
	` Now is the timi`

We also note that this type of attack may be possible without either physical access to the machine or root privileges on a shared network folder or server if the owner of the encrypted files does not properly protect the files from unauthorized read access.

4.2 Swap via Hibernation

Two storage-related components can interact in ways that introduce the two-time-pad vulnerability. One component is memory caching, which may keep recently modified encrypted data around, in hope to consolidate multiple updates to the same disk location into a single write to disk. The other component is hibernation, which allows a system to save current memory states to non-volatile storage and power down. The system can then later resume execution from the state saved before hibernation by restoring power. Although the cached data can be encrypted at all times, the memory version is not always consistent with the disk version due to the write-back policy. A trigger of hibernation will lead to two versions of encrypted data to be stored on disk, with a potential reuse of the same key and IV.

We demonstrate this vulnerability under Linux 2.6.22.6, in which `swsusp` implements a software suspend mechanism that allows for non-volatile hibernation. To suspend, `swsusp` creates a list of active pages excluding bus addresses used between peripheral buses and memory. Next, `swsusp` writes both the user memory and kernel memory to the swap device. Then, `swsusp` follows the three-level page cache, starting from the page global directory, and writes the pages in sequence onto the disk. Note that the kernel memory always gets written to the beginning of the swap partition, and only active pages are written during suspend We can use this knowledge to reduce the search space significantly when examining a large swap device.

To verify that kernel memory is written to the swap partition during suspend (instead of having memory retaining the content in a low-power mode), we conducted the following experiment. We modified the kernel `init()` function to `kmalloc` 512 bytes 20 times. Each time we filled the 512 bytes with the same predetermined content. Before each test, we verified that the predetermined string was not present in memory initially, and zeroed out the swap partition. We then ran `mkswap` to create a clean swap device. The machine was then rebooted, followed by a suspend, resume, and another reboot. Next, we searched the swap partition for the predetermined 512 byte string, which appeared 20 times. This experiment told us that the suspend mechanism does write kernel memory to the swap partition when invoked.

Up to now, we have shown that known plaintext can be found at unknown locations on a suspend partition. We then need to illustrate how to find encrypted data at unknown locations on a swap partition. In this experiment we (1) injected encrypted data into memory that is then propagated to swap via the `swsusp` mechanism, (2) identified swap candidate blocks with high entropy, (3) created XOR blocks by XORing encrypted file blocks with the candidate swap blocks, and (4) used DecodeXOR to analyze XOR blocks that exhibit low, but non-zero entropy.

We began with two versions of a file. The first was an 8-KB file with the repeating string "This is a test" encrypted with AES using CTR mode. The second was the same file with a space character inserted somewhere in the second 4-KB block of the file, and the modified file was truncated to 8 KB. The file was then encrypted with AES using CTR mode. Both versions used the same IV and key. The modified version was loaded into page-aligned kernel memory and the machine is suspended using `swsusp`. This simulated a request to modify an encrypted file that did not immediately reach the disk.

At this point we assume that an attacker has access to the entire disk image. The unmodified 8-KB encrypted file can be identified and retrieved if the file system does not hide its directory structure. The entropy was then computed via the `ent` tool [43] on each 4-KB block on the swap partition. Blocks with high entropy were marked as candidate blocks. This step filters out half of the swap partition blocks. Each candidate swap block was XORed with each 4-KB block in the unmodified encrypted data file, and the entropy was again calculated.

The resulting entropy placed the XORed block into one of three categories. When the entropy is high, either one of the two XORed blocks is encrypted, or both blocks are encrypted with different key masks. Therefore, the XORed block is not considered further. This step reduces the number of candidate swap blocks down to two. When the entropy is zero, the candidate swap block is the same as the unmodified file block. In our case, only one block was XORed to zero. Knowing how the Linux memory allocator strives to allocate memory contiguously whenever possible, the zero-block location can serve as a valuable reference point to speed up searches. The remaining non-zero entropy candidate blocks are then examined further to determine the position of the modification within the 4-KB block. Specifically, we took the XORed 4-KB block and found the first non-zero byte at the starting position. The end of the modified encrypted data is found by traversing from the last byte of the block and back to the beginning of the block until a non-zero byte is reached. The identified XORed string between the beginning and the end were then analyzed by DecodeXOR.

Although the above experiment is a minimalist example, we illustrated key steps to exploit the vulnerability. We also tried a larger example with 500 MB of swap pair-wise XORed with every encrypted block in a 2-GB partition. The search time only took about 30 seconds on a 2.8 GHz Pentium® D machine with 1 GB of RAM. A more realistic setup would include a system that has been in use for some time, which would result in versions of random content in the memory swap overlapped at times from various hibernation sessions, making it more difficult to identify candidate blocks. With the knowledge that `swsusp` writes the content of the kernel memory to the beginning of the swap partition and that memory is divided into regions and allocated consecutively whenever possible, together with the result of unmodified encrypted data XORed into zeros, we can reduce our search space drastically when analyzing the swap partition.

4.3 Flash Storage

The use of flash-based memory storage is now ubiquitous due to its low cost, the lack of moving parts (when compared to hard disks), low levels of energy consumption, and fast read times. Flash comes in two forms: NAND and NOR. NOR flash memory allows applications to execute in-place and has been traditionally used in embedded computing devices, such as cell phones and PDAs. NAND flash is less expensive, is accessed on a page basis (typically 512 bytes), and is typically used in digital cameras, flash drives, USB thumb drives, solid state hard disks, and mp3 players. In this paper, we are only concerned with NAND flash.

Although popular, NAND flash has a number of physical limitations [13]: (1) Each memory location can only be written from 10,000 to 1,000,000 times before they become unreliable. (2) The erasure time of a memory location is orders of magnitude longer than reads. (3) Overwriting a memory location with existing data involves first erasing the memory location before writing new data. To overcome the limited number of erasure cycles for a given memory location, a technique called wear-leveling [6, 22] rotates the usage, or wear and tear, of memory locations evenly to prolong the life of the device. To avoid slow erasures and overwrites, many storage optimizations, such as flash translation, allow new updates to be stored in empty memory locations, while the locations with old content are erased in the background.

Various optimizations are problematic when encryption methods are applied to common NAND flash devices. Old versions of ciphertext blocks may be frequently left on the device due to the lack of provision of in-place updates. To demonstrate this, we used the file system jffs2, which is the second version of the Journaling Flash File System [44]. Flash file systems like jffs2 are typically log-structure-based [36], tailored to provide wear-leveling and performance optimizations for flash devices, and they operate directly on the flash chips. Since these mechanisms are performed at the file-system level and are not sensitive to the underlying medium, we used a 256MB on-disk partition to emulate flash storage. The main reason for emulation is to isolate the optimizations performed by jffs2 from built-in wear-leveling and translation mechanisms on flash, which are not always well documented or exposed for direct manipulations and observations. The emulation was done via the following steps.

1. We loaded the emulation module `block2mtd,` which came with the jffs2 source [45]. We also loaded modules `mtdblock` and jffs2 for our test system running Linux 2.6.18-5-686.
2. We issued the command `mkfs.jffs2 -o /dev/sdb4` to create the file system, where `sdb4` is our emulated flash partition.
3. We issued the command `mount -t jffs2 /dev/mtdblock0 /dev/sdb4` to mount the file system.

We then wanted to simulate a user making a supposed "in-place" update to an encrypted file stored on our emulated flash partition. We began with two versions of a file encrypted with 128-bit AES in OFB mode. The file was an 8-KB file with the repeating string "This is a test", and the second was the same encrypted file with a space character inserted somewhere in the second 4-KB block of the file, with the file truncated to 8KB. Both versions used the same random IV and key. The original version was placed on the emulated flash partition. The modified file was stored under a different file system. We copied the modified 4-KB block to overwrite the second block of the original file.

Using a hex editor on the raw emulated flash partition, we were able to verify large portions of the second 4 KB of the original file. An attacker could similarly use tools based on entropy, such as the tools we used in Section 4.2, to discover probable old ciphertext blocks. By XORing two versions of ciphertexts and feeding the result to DecodeXOR, we were able to reconstruct plaintexts from the last 4KB of each ciphertext file.

4.4 Secure Storage and Deletion using the AON Transform

All-or-nothing (AON) [35, 5] is defined as a cryptographic transform that, given only partial output, reveals nothing about its input. In other words, no one block of ciphertext can be decrypted without obtaining all blocks of ciphertext. The original intention of AON was to increase the difficulty of brute-force attacks on the key.

This concept was adapted by a versioning file system [32], which was based on ext3cow [31], a copy-on-write file system. This system used authenticated encryption to store data confidentially, provide file integrity, and delete unwanted versions of files on versioning file systems. Specifically, versions of files are deleted by overwriting a small 128-bit stub. Once a stub is overwritten, the corresponding version of a file cannot be recovered.

The encryption algorithm takes the following as inputs:

- One plaintext data block divided into 128-bit plaintext blocks $\{d_1, d_2, \ldots, d_m\}$.
- A unique identifier *id* for the block (block's physical address).
- A unique global counter x (a system-wide epoch currently stored in the superblock).
- An encryption key K.
- A message authentication (MAC) key M.

To encrypt, as shown in Figure 2, the algorithm generates a unique encryption counter ctr_1 (step 1) by concatenating the unique block identifier *id* with the unique global counter x, padded with zeros. The algorithm then performs an AES encryption in CTR mode (step 2) using the XOR of the encryption key K and ctr_1, resulting in the encrypted ciphertext blocks $\{c_1, c_2, \ldots, c_m\}$.

```
Input: Data Block $\{d_1, d_2, \ldots, d_m\}$, Block ID $id$,
Counter $x$, Encryption key $K$, MAC key $M$
1: $ctr_1 \leftarrow id \| x \| 1 \| 0^{128-|x|-|id|-1}$
2: $\{c_1, c_2, \ldots, c_m\} \leftarrow$ AES-CTR$[ctr_1, K, \{d_1, d_2, \ldots, d_m\}]$
3: $t \leftarrow$ HMAC-SHA-1$[M, \{c_1, c_2, \ldots, c_m\}]$
4: $ctr_2 \leftarrow id \| x \| 0 \| 0^{128-|x|-|id|-1}$
5: $\{x_1, x_2, \ldots, x_m\} \leftarrow$ AES-CTR$[ctr_2, t, \{c_1, c_2, \ldots, c_m\}]$
6: $x_0 \leftarrow x_1 \oplus \ldots \oplus x_m \oplus t$
Output: Stub $x_0$, Ciphertext $\{x_1, x_2, \ldots, x_m\}$
```

Figure 2. The adopted AON transform encryption operation.

The encrypted blocks are authenticated (Step 3) using SHA-1 and MAC key M as a keyed-hash for message authentication codes (HMAC) to produce the authenticator t. A second unique encryption counter ctr_2 is created (step 4), and t and ctr_2 are used to re-encrypt the data via the AES-CTR mode to produce double-ciphertext blocks $\{x_1, x_2, \ldots, x_m\}$. The stub x_0 is generated (Step 6) by XORing all the double-ciphertext blocks $\{x_1, x_2, \ldots, x_m\}$ with the authenticator t. The resulting stub is an expansion of the encrypted data and is not secret. Decryption is detailed in [32].

A number of properties in this AON encryption make it an intriguing example. (1) The ciphertext is doubly encrypted. Therefore, at the first glance little information is available to extract versions of singly encrypted text. (2) The second encryption is in counter mode, but with a changing key, namely t the authenticator, which is a function of all singly encrypted text. (3) The counters are based on physical disk locations, which are unique, and the epoch number x (also unique). In addition, the copy-on-write semantics may avoid in-place updates, which further prevents different versions of data to reuse the same disk location and epoch number.

With a closer look, we have the access to ctr_1, ctr_2, x_0, $\{x_1, x_2, \ldots, x_m\}$ as public knowledge. Therefore, the authenticator t can be retrieved, reflecting that the second round of encryption is only a transformation to achieve the all-or-nothing property. Therefore, we once again have the access to singly encrypted $\{c_1, c_2, \ldots, c_m\}$ for cryptanalysis. Since the two counters are based on the disk location and the epoch number, we have to check and verify the circumstances where in-place updates are allowed and how the epoch number is incremented. The ext3cow design conserves storage by allowing in-place updates for the same file blocks within the same epoch. The epoch number is incremented when a snapshot is taken. Therefore, as long as snapshots are not taken frequently, an attacker can use disk backup images to locate different file versions within the same epoch for cryptanalysis.

To demonstrate this weakness, we created a scenario with the following steps:

- Create file 1 with two encrypted 4-KB blocks, C1 and C2
- Create a backup B1
- Update C2 with C2'
- An attacker can take a disk image dump B2 and analyze B1 and B2

We gained access to the original source of AON-ext3cow. However, we were unable to retrofit the environment to conduct our experiments on the actual system. Instead, we duplicated the AON-ext3cow encryption scheme via a user-level program using the `nettle` encryption library [29] (`libnettle2` and `libnettle-dev` packages under Ubuntu Linux), so that we could create the same files that would be generated in the above scenario. Through static code reviews of AON-ext3cow, we were glad to find out that this particular vulnerability is actually fixed, via replacing the CTR mode with CBC. However, our point is not about showing the flaw of a particular system. Rather, even the design of modern and sophisticated storage systems can still mismatch diverse storage usage patterns and become vulnerable. Therefore, we implemented our user-level encryption program according to the scheme described in the original paper.

Without a running instance of AON-ext3cow, we divided our demonstration into two parts: (1) showing that we can update a file in-place with the same epoch under ext3cow (2) showing that we can decrypt AON-encryption via two versions of ciphertext.

For the former, we downloaded and installed ext3cow-2.6.20.3 and an epoch query program called `tt` from the ext3cow-tools package. We then performed the following steps. (1) We created a plaintext file with two 4-KB blocks and noted their relative positions within the file system partition, which were 10,240 and 10,241. (2) We ran `tt` to check the epoch number, which was also the file creation time, represented in the number of seconds since 1/1/1970. This number was 1201314595. (3) We updated the file content in the second 4-KB block and searched for the original content in the first block and the modified content in the second block. Both blocks stayed at the same location, indicating in-place updates. (4) We ran `tt` again, and the epoch number remained the same.

For demonstrating the decryption capability, we first created two versions of an encrypted 8-KB file with the same content as the files in the flash example in Section 4.3. We assume the same user-provided encryption key K is used to encrypt both files, since the AON-ext3cow code suggests one key K is used to encrypt the entire file system. We used the hex string "9DFE54BFABA6A065FD1091F7B98524E4" as key K, the hex string "C27CB47DDAC849FCA4F90656E694CF90" as MAC key M, 41,943,040 and 41,947,136 (byte offsets) as block ID id, and 1201314595 as the global epoch number x. To extract the authenticator t for each version, we XORed all doubly-encrypted blocks, including the stub x_0. To extract the singly-encrypted ciphertext, reverse the second encryption operation by performing the AES-CTR operation with ctr_2, the authenticator t, and doubly-encrypted blocks $\{x_1, x_2, \ldots, x_m\}$. With singly-encrypted ciphertext, we were able to use DecodeXOR to extract the plaintext. If the entire disk images B1 and B2 were available, an attacker could use tools based on entropy, such as the tools we used in Section 4.2, to discover probable old ciphertext blocks.

5. DISCUSSION

Through the above demonstrations, we experience first-hand that applying cryptography to storage is different from applying cryptography to networks in both theory and in practice. The attacks are quite feasible with simple home-grown tools and the speed of modern computers.

One alternative is to use full-disk encryption, which can be provided by the new generation of hard disks that claim low performance overhead [38]. Since this solution defines the disk drive as the boundary for encrypted data and the uniqueness of keys and IVs, this makes it unclear whether different versions of encrypted disk images may reveal crucial information. Another alternative may lay with the narrow-block tweakable [25] encryption mode XTS. At the time of this writing, the IEEE Security in Storage Working Group (SISWG) has submitted an active, approved draft to the IEEE for narrow-block encryption P1619/D18 titled "Draft Standard for Cryptographic Protection of Data on Block-oriented Storage Devices" which details the use of XTS for data at rest. Further analysis of this new mode may be needed.

Other traditional modes of encryption, such as CBC and ECB, do not suffer from the vulnerabilities mentioned in this paper. However, they need to be combined with other mechanisms in order to overcome structural analysis. For example, Microsoft's BitLocker encrypts a specified volume sector-by-sector using AES in CBC mode with a diffuser called Elephant [11]. The diffuser is necessary due to a weakness in CBC mode, which allows an attacker to flip an *i*th bit in the next block's plaintext by flipping the *i*th bit in the current block's ciphertext. This is done at the risk of randomizing the current block's plaintext. This diffuser runs a series of XORs and rotations on words within a sector, which enables one flipped bit to cause more random bit flips within the same sector.

Perhaps the encryption scheme used in IBM's eCryptfs [19] is part of the answer. eCryptFS divides files into encryption extents to support fast random access. IVs are changed for each write, and the CBC mode is used within each extent. However, so far, we have found that design assumptions of various solutions are typically violated by implementations and unanticipated interactions between the encryption layer and the other storage layers. Therefore, unless the design makes no assumptions about other system components, it is likely that the weakest point is not the file system itself (e.g., RAID parities and backups).

One possible approach to a solution is to rethink the entire storage data path with a clean slate in the context of cryptography. The storage equivalence of a theoretical one-time pad is beyond the boundary and lifetime of the system itself. At one extreme, whenever a piece of encrypted data is updated, shifted, or even copied, the unique key or the IV of the encrypted data needs to be changed. The encryption component can make no assumptions about whether versions of encrypted data can be properly removed, once generated. The resulting characteristics of this system will be similar to write-once [37] or copy-on-write [31] storage systems. Unfortunately, such a design is likely to be cost-prohibitive in terms of performance, storage requirements, and key management complexity.

To relax the design constraints, the interface of storage layers needs to become more expressive for communicating and controlling ways of handling encrypted data. For example, a layer needs mechanisms to provide verifiable guarantees of removing encrypted data and not moving or copying encrypted data. However, given the backward compatibility and legacy constraints of storage mechanisms such as read-copy-update [14], versioning [31], journaling [44], RAIDs [30], and so on, achieving secure storage will remain a difficult research area, ripe with challenges and opportunities.

6. RELATED WORK

Much of the related work on algorithms [18, 27, 42] and implementations [40, 9] of cryptanalysis on two-time pads was already discussed in Sections 3.1 and 3.2. Most of the work reviewing two-time pad vulnerabilities is explored in the context of network communications. We discuss a similar weakness found in the Wired Equivalent Privacy (WEP) protocol, which was introduced by the 802.11 standard for wireless communications [23] and is based on the believed-to-be-secure RC4 stream cipher [34]. Borisov et. al. [4] discuss many security weaknesses of WEP, including implementation and architectural problems which causes the keystream to be repeated much more frequently than necessary.

WEP expands a secret key and a public per-packet IV into a keystream of pseudorandom bits which is XORed with the plaintext to produce ciphertext. Plaintext is recovered by producing an identical second keystream and XORing it with the ciphertext. The WEP standard recommends using different IVs for each packet, but some implementations use a somewhat less than random approach to changing IVs. For example, the particular PCMCIA cards that Borisov et. al. examined reset the IV to 0 every time they were re-initialized and incremented the IV by 1 for each packet. Those cards would similarly re-initialize every time they were plugged into the laptop. Thus, keystreams corresponding to low-valued IV's were found frequently. An architectural problem of a small IV field (24 bits) was also found in the WEP standard that nearly guarantees that the same IV will be used for multiple packets.

7. CONCLUSION

Through empirical demonstrations, we have shown that the characteristics of storage are fundamentally different from those of network, and cryptographic assumptions on the uniqueness of keys and IVs relative to content can be violated through unanticipated side effects from various storage mechanisms. In particular, we have demonstrated storage mechanisms that can convert intended one-time pads into two-time pads: in-place updates, backups, inconsistencies between memory and disk content, and wear-leveling for flash storage. Additionally, we illustrate the feasibility to exploit two-time pads to extract plaintexts efficiently with home-grown tools, bringing attacks from the theoretical realm to life.

While these examples cover a spectrum of storage mechanisms, they are still the tip of the iceberg. Legacy storage mechanisms and semantics such as RAIDs, versioning, etc. and their interactions with cryptography are yet to be examined. We believe that a clean-slate rethinking of storage requirements and usage patterns in the context of cryptography is one direction from which to envision a holistic solution. Expanding the interface of individual storage components to better support cryptographic assumptions and guarantees is another.

Given that two-time pads are just one of many security problems in the storage domain, a deeper problem lies in the lack of fostering of cross-understanding between the storage and security domains. Hopefully, this paper takes steps toward bridging the understanding of requirements and constraints between the two fields.

8. ACKNOWLEDGEMENTS

We thank Peter Reiher and Geoff Kuenning for reviewing an early draft of this paper. We also thank Zachary N. J. Peterson et. al for providing accesses to various to the AON secure deletion versioning file system code. Cory Fox, Ryan Fishel, Dragan Lojpur, Mark Stanovic, and Ted Baker also have contributed to this work. This work is sponsored in part by DoE grant number P200A060279. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the DoE, FSU, or the U.S. Government.

9. REFERENCES

[1] Baddeley AD, Conrad R, Thompson WE. Letter structure in the English language. *Nature*, 186, pp. 414-416, 1960.

[2] Bennison PF, Lasher PJ. Data security issues relating to end of life equipment. *Proceedings of the 2004 IEEE International Symposium on Electronics and the Environment*, May 2004.

[3] Bloom B, Space/time tradeoffs in hash coding with allowable errors, *Communications of the ACM*, 13(7), pp. 422–426, July 1970.

[4] Borisov N, Goldberg I, Wagner D. Intercepting mobile communications: The insecurity of 802.11. *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, 2001.

[5] Boyko V. On the security properties of OAEP as an all-or-nothing transform. In *Advances in Cryptology - Crypto'99 Proceedings,* Springer-Verlag, pp. 503–518, August 1999.

[6] Chang L. On efficient wear leveling for large-scale flash-memory storage systems. *Proceedings of the 2007 ACM Symposium on Applied Computing*, March 2007.

[7] Chow J, Pfaff B, Garfinkel T, Christopher K, Rosenblum M. Understanding data lifetime via whole system simulation, *Proceedings of the 12th USENIX Security Symposium*, 2004.

[8] Chow J, Pfaff B, Garfinkel T, Rosenblum M. Shredding your garbage: Reducing data lifetime through secure deallocation, *Proceedings of the USENIX Security Symposium*, August 2005.

[9] Dawson E, Nielsen L. Automated cryptanalysis of XOR plaintext strings, Cryptologia, 20(2):165-181,April 1996.

[10] Debian Source Repository, http://ftp.de.debian.org/debian, 2008.

[11] Ferguson, N. AES-CBC + Elephant diffuser: A Disk Encryption Algorithm for Windows Vista. Technical Report, August 2006. Available online at http://www.microsoft.com/downloads/details.aspx?FamilyID=131dae03-39ae-48be-a8d6-8b0034c92555&DisplayLang=en.

[12] Gaines HF, Cryptanalysis: A study of ciphers and their solutions. New York: Dover., 1939.

[13] Gal E, Toledo S, Mapping structures for flash memories: techniques and open problems, *Proceedings of the IEEE International Conference on Software - Science, Technology and Engineering,* February 2005.

[14] Ganger G, Patt Y. Metadata Update Performance in File Systems, *Proceedings of the First USENIX Conference on Operating Systems Design and Implementation,* 1994.

[15] Garfinkel SL, Shelat A. Remembrance of Data Passed: A Study of Disk Sanitization Practices, *IEEE Security & Privacy*, 1(1), pp. 17-28, 2003.

[16] Garfinkel T, Pfaff B, Chow J, Rosenblum M. Data Lifetime is a Systems Problem, *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, September 2004.

[17] The GNU Privacy Guard, http://gnupg.org, 2008.

[18] Griffing A. Solving XOR Plaintext Strings with the Viterbi Algorithm. Cryptologia, 30(3), pp. 258-265, 2006.

[19] Halcrow M. eCryptfs: a Stacked Cryptographic Filesystem. *Linux Journal.* April 2007.

[20] Hohmann C. CryptoFS. http://reboot.animeirc.de/cryptofs/. August 2007.

[21] Jones MN, Mewhort DJK. Case-sensitive letter and bigram frequency counts from large-scale English corpora. *Behavior Research Methods, Instruments, & Computers*, 36, pp. 388-396, 2004.

[22] Kawaguchi A, Nishioka S, Motoda H, A flash-memory based file system, *Proceedings of the USENIX 1995 Technical Conference*, January 1995.

[23] L. M. S. C. of the IEEE Computer Society. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. IEEE Standard 802.11, 1999 Edition, 1999.

[24] Large Text Compression Benchmark, http://www.cs.fit.edu/~mmahoney/compression/text.html, 2008.

[25] Liskov, M., Rivest, R., And Wagner, D. Tweakable block ciphers. In *Advances in Cryptology* (CRYPTO '02), Lecture Notes in Computer Science, Springer-Verlag.

[26] Malita F. LUFS Userland Filesystem - Default branch. http://freshmeat.net/projects/lufs/. October 2003.

[27] Mason J, Watkins K, Eisner J, Stubblefield A. A natural language approach to automated cryptanalysis of two-time pads. *Proceedings of the 13th ACM Conference on Computer and Communications Security*, October 2006.

[28] Mayzner MS, Tresselt ME. Tables of single-letter and digram frequency counts for various word-length and letter-position combinations. *Psychonomic Monograph Supplements*, 1(2), pp, 13-32, 1965.

[29] Nettle – A Low-Level Cryptographic Library, http://www.lysator.liu.se/~nisse/nettle/, 2008.

[30] Patterson DA, Gibson GA, Katz RH. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, September 1988.

[31] Peterson Z, Burns R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage* 1(2), pp. 190-212, 2005.

[32] Peterson ZNJ, Burns R, Herring J, Stubblefield A, Rubin AD. Secure Deletion for a Versioning File System. *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[33] Richardson, R. 2007. CSI Survey 2007: The 12th Annual Computer Crime and Security Survey. Computer Security Institute. http://www.gocsi.com/forms/csi_survey.jhtml.

[34] Rivest RL. *The RC4 Encryption Algorithm.* RSA Data Security, Inc., March 1992.

[35] Rivest RL. All-or-nothing encryption and the package transform. *Proceedings of the Fast Software Encryption Conference*, 1997.

[36] Rosenblum M, Ousterhout JK, The Design and Implementation of a Log-Structured File System, *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[37] Santry DJ, Feeley MJ, Hutchinson NC, Veitch AC. Elephant: The file system that never forgets. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, 1999.

[38] Seagate Momentus Hard Drive Family, http://www.seagate.com/www/en-us/products/laptops/momentus/, 2008.

[39] Solso RL, King JF. Frequency and versatility of letters in the English language. *Behavior Research Methods & Instrumentation*, 8, 283-286, 1976.

[40] Tutte W. FISH and I, A transcript of Tutte's lecture at the University of Waterloo, June 1998.

[41] Valli C. Throwing out the Enterprise with the Hard Disk, *Proceedings of the 2nd Australian Computer, Networks & Information Forensics Conference,* 2004.

[42] Viterbi AJ. Error Bounds for Convolutional Codes and Asymptotically Optimal Decoding Algorithm. *IEEE Transactions on Information Theory*, 13(2), pp. 260-267, 1967.

[43] Walker J. Ent – A Pseudorandom Number Sequence Test Program, http://www.fourmilab.ch/random/, 2008.

[44] Woodhouse D. JFFS: The Journaling Flash File System. *Proceedings of the Ottawa Linux Symposium.* RedHat Inc., 2001.

[45] Woodhouse D. JFFS2: The Journalling Flash File System, version 2. http://sourceware.org/jffs2/, 2008.