A Survey of Confidential Data Storage and Deletion Methods

SARAH M. DIESBURG
Florida State University
AND
AN-I ANDY WANG
Florida State University

As the amount of digital data grows, so does the theft of sensitive data through the loss or misplacement of laptops, thumb drives, external hard drives, and other electronic storage media. Sensitive data may also be leaked accidentally due to improper disposal or resale of storage media. To protect the secrecy of the entire data lifetime, we must have confidential ways to store and delete data. This survey summarizes and compares existing methods of providing confidential storage and deletion of data in personal computing environments.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection – *Cryptographic controls*; H.3.2 [**Information Storage and Retrieval**]: Information Storage; K.4.1 [**Computers and Society**]: Public Policy Issues – *Privacy*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Human Factors, Legal Aspects, Performance, Security

Additional Key Words and Phrases: Data overwriting techniques, data theft, personal privacy, secure deletion, secure erasure

---

1. INTRODUCTION

As the cost of electronic storage declines rapidly, more and more sensitive data are stored on media such as hard disks, CDs, and thumb drives. The trend of the paperless office also drives businesses toward converting sensitive documents, once stored in locked filing cabinets, into digital forms. Today, an insurance agent can carry a laptop that holds thousands of Social Security numbers, medical histories, and other confidential information.

As early as 2003, the U.S. Census Bureau reported that two-thirds of American households have at least one computer, with about one-third of adults using computers to manage household finances and make online purchases [U.S. Census Bureau 2005]. These statistics suggest that many computers store data on personal finances and online transactions, not to mention other confidential data such as tax records, passwords for bank accounts, and email. We can estimate that these figures have risen dramatically since the 2003 survey.

Sensitive information stored in an insecure manner is vulnerable to theft. According to the most recent CSI Computer Crime and Security Survey [Richardson 2007], 50 percent of the respondents have been victims of laptop and mobile theft in the last 12 months. These respondents include 494 security practitioners in U.S.

This research was supported in part by the U.S. DoE, grant number P200A060279.
Authors' addresses: Sarah Diesburg and An-I Andy Wang, 253 Love Building, Department of Computer Science, Florida State University, 32306.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

corporations, government agencies, financial institutions, medical institutions, and universities. The survey also shows that between the years 2001 and 2007, the theft of electronic storage occurred much more frequently than other forms of attack or misuse such as denial of service, telecom fraud, unauthorized access to information, financial fraud, abuse of wireless network, sabotage, and Web site defacement. Many incidences of theft continue to make headlines across many segments of the society, including the government [Hines 2007], academia [Square 2007], the health industry [McNevin 2007; Sullivan 2005], and large companies [WirelessWeek 2007].

Two major components exist to safeguard the privacy of data on electronic storage media. First, data must be stored in a confidential manner to prevent unauthorized access, and the solution should not impose significant inconvenience during normal use. Second, at the time of disposal, confidential data must be removed from the storage media as well as the overall computing environment in an irrecoverable manner. While the fundamental goals are clear, existing solutions tend to evolve independently around each goal. This survey summarizes the advantages and challenges of various confidential data storage and deletion techniques, with the aim of identifying underlying trends and lessons to arrive at an overarching solution.

Due to the size of the problem, the focus of this survey is on non-distributed, single-user computing environments (e.g., desktops and laptops). The user is assumed to have system administrative privileges to configure the confidentiality settings of storage and deletion methods. The confidentiality threat model assumes that attacks to recover sensitive data are staged after the computer has been powered off: in other words, the attacker uses "dead" forensic methods. We assume that an attacker can again access to the original storage media either by booting the original machine with a live CD or removing the storage media and placing it in another machine under the attacker's control. Other forms of attack (e.g., network-based attacks, memory-based attacks, or privilege escalation attacks) are beyond the scope of this survey. Thus, the strength of the confidentiality of the data storage or deletion is based directly upon the technique used to store or delete the data at rest (for example, strength of the encryption technique used).

2. SECURITY BACKGROUND

This section is designed for storage researchers and provides the relevant security concepts used when comparing storage designs.

The general concept of secure handling of data is composed of three aspects: confidentiality, integrity, and availability. Confidentiality involves ensuring that information is not read by unauthorized persons. Using encryption to store data or authenticating valid users are example means by which confidentiality is achieved. Integrity ensures that the information is not altered by unauthorized persons. Storing a message authentication code or a digital signature computed on encrypted data is a way to verify integrity. Finally, availability ensures that data is accessible when needed. Having multiple servers withstand a malicious shutdown of a server is one way to improve availability.

This survey compares various confidential storage and deletion approaches in terms of how each trades confidentiality with convenience (e.g., ease-of-use, performance, and flexibility of setting security policies). Both integrity and availability goals are assumed and are beyond the scope of this survey.

The strength of confidentiality is the result of how well secure storage and deletion mechanisms address the following questions:

- If encryption is used, how well are the keys protected?
- Do copies of sensitive data reside at multiple places in the system?

- If encryption is used, how strong is the encryption mechanism and mode of operation in terms of the computational efforts to subvert the encryption?
- Can deleted data be recovered? If so, what are the time and resource costs?
- Is the entire file securely deleted, or is some portion left behind (such as the file name or other metadata)?

In other words, the confidential data must not be accessed by unauthorized persons after it is properly stored or deleted.

The ease-of-use of an approach reflects the level of inconvenience imposed to end users. Methods of confidential storage and deletion that are too hard to use will either encourage users to circumvent the methods or discourage users from using them entirely [Whitten and Tygar 1999]. Some aspects of the user model examined include:

- The number of times a person must enter an encryption key per session
- The ease with which the method is invoked
- The number of encryption keys or passwords a person or a system must remember

Levels of inconvenience vary depending on the type of applications used. For example, entering a password to access each sensitive file in a text editor environment is less inconvenient than having to enter a password every time a compiler accesses a sensitive file, since a compiler must generally access many files in a short period of time. In the most ideal environment, users should behave similarly when accessing secure files versus normal files.

Performance is another form of convenience, as methods that either take too long or consume unreasonable amounts of system resources will not be used. For both confidential storage and deletion, performance can be measured by the latency and bandwidth of file access/erasure and overhead pertaining to the encryption algorithm and the mode of operation used. Additionally, both methods can be measured by the time taken per operation and total amount of system resources used.

Security policies are comprised of a set of rules, laws, and practices that regulate how an individual or organization manages, distributes, and protects secure information. A policy may be specific to a person or organization and may need to change frequently. This survey compares the flexibility of the method, or the ease of configurations, with regards to the implementation of various confidential storage and deletion policies. Some aspects that we examine include:

- Method compatibility with legacy applications and file systems
- The ease of key or password revocation
- How easily one may change the method's configuration to fulfill a security policy (e.g., encryption algorithm and key size)
- Whether one can control the granularity (e.g., file and disk partition) of confidential storage and deletion operations

Many techniques of confidential storage and deletion involve cryptography. The following subsection briefly introduces and compares commonly used encryption algorithms and their modes of operation.

2.1 Encryption Basics

Encryption is a procedure used in cryptography "to scramble information so that only someone knowing the appropriate secret can obtain the original information (through decryption) [Kaufman et al. 2002]." The secret is often a key of *n* random bits of zeros and ones, which can be derived through the use of a password or passphrase. A key's strength is often associated with the length of the key which, if it consists of truly random

bits, requires a brute-force enumeration of the key space to decrypt the original message. Another measure of a key's strength is its entropy, which measures the degree of randomness associated with the bits in the key.

An encryption algorithm, or cipher, takes an input (referred as plaintext) and produces encrypted output (i.e., ciphertext); similarly, a decryption algorithm takes a ciphertext as input and generates decrypted plaintext. Plaintext can be text files, videos, music, executables, entire disk partitions, or even encrypted files for nested encryptions. Thus, encrypting a music file is no different than encrypting a text file unless otherwise stated.

Encryption algorithms can be either symmetric or asymmetric. Symmetric algorithms use the same key for both encryption and decryption. Asymmetric algorithms use two keys: one for encryption and another for decryption. For example, public-key cryptography, which is a form of asymmetric encryption, uses two keys (public and private keys) and is often used to establish secure communication across a network where there is no way to exchange a symmetric key beforehand. Symmetric encryption schemes can be many times faster than comparable asymmetric schemes, and are therefore used more often in secure data storage, especially when the data in question does not traverse through an insecure network.

Common symmetric key encryption algorithms include the Data Encryption Standard (DES), Triple-DES (3DES), and the Advanced Encryption Standard (AES). These algorithms are block ciphers, meaning that they take a block of symbols of size *n* as input and output a block of symbols of size *n*. DES was published in 1975 and developed as the U.S. standard for unclassified applications in 1977 [Stinson 2002]. DES uses a key size of 56 bits and a block size of 64 bits. The main criticism of DES today is that the 56-bit key length is too short. With newer CPUs, the key space of 2^{56} can be enumerated. Even with machines in 1998, a machine called the "DES Cracker" could find a DES key in 56 hours.

Triple-DES was built to enlarge the DES key space without requiring users to switch to a new encryption algorithm. 3DES operates by performing three DES operations on the data with three keys: encryption with key one, decryption with key two, and encryption with key three. The three keys increase the key space to 2^{168}, but the strength of 3DES is only twice as strong as DES as demonstrated in the meet-in-the-middle attack [Chaum and Evertse 1985]. Unfortunately, performing three cryptographic operations for every data access imposes a high performance penalty.

DES was replaced with the Advanced Encryption Standard (AES) algorithm in 2001. AES has a block length of 128 bits and supports key lengths of 128, 192, and 256 bits. Among the five finalist algorithms to be chosen as AES (MARS, RC6, Rijndael, Serpent, Twofish), Rijndael was chosen "because its combination of security, performance, efficiency, implementability, and flexibility was judged to be superior to the other finalists [Stinson 2002]." The National Security Agency (NSA) has reviewed and concluded that all five finalists were secure enough for U.S. Government non-classified data. In 2003, the U.S. government announced that AES, with all key lengths, is sufficient to protect classified information up to the level of *secret*. Only AES key lengths of 192 or 256 bits can protect classified information at the *top secret* level [Ferguson et al. 2001].

Key revocation is the act of invalidating an encryption key so that it may no longer be used to decrypt data. This process often involves re-encrypting the data with a new encryption key. Three types of key re-encryption modes exist: aggressive, lazy, and periodic [Riedel 2002]. Aggressive re-encryption involves re-encrypting data directly after key revocation. Lazy re-encryption involves re-encrypting selected data once it is next modified or read. Periodic re-encryption involves changing keys and re-encrypting data periodically. All cryptographic systems discussed in-depth in this survey force the user to employ aggressive re-encryption, when key revocation is possible. Some out of

scope network-based cryptographic systems listed briefly in Section 3.1.3 employ lazy re-encryption (e.g. Cepheus [Fu 1999], Plutus [Kallahalla et al. 2003], and Secure Network-Attached Disks (SNAD) [Miller et al. 2002]).

2.2 Traditional 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 ciphertext of arbitrary length. For example, using AES alone, one may only input and output blocks of 128 bits each. Using AES with a mode of operation for a block cipher, one may input and output data of any length.

An initialization vector (IV) is commonly used with many block ciphers: it is a small, often random, but non-secret value used to help introduce randomness into the block cipher. The IV is often used at the beginning of the block cipher.

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 [Dworkin 2001].

2.2.1 Mode Examples and Specifications. ECB is the simplest mode of operation, and does not use an IV (Figure 1). With a *key*, P_i as the ith block of plaintext, and C_i as the ith block of ciphertext, the encryption is performed as $C_i = E_{key}(P_i)$, and decryption is performed as $P_i = D_{key}(C_i)$.

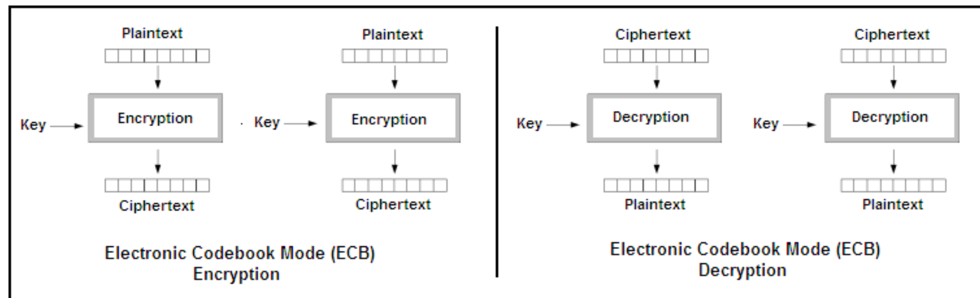

Figure 1. ECB-mode encryption and decryption.

The CBC mode is slightly more complicated and uses an IV (Figure 2). Each block of plaintext first is XORed with the previous block of ciphertext before being encrypted. Therefore, each block of ciphertext relies on its previous block of ciphertext. Encryption of the first block of plaintext is performed as $C_1 = E_{key}(P_1 \oplus \text{IV})$, where C_1 is the 1st block of ciphertext; IV is the random, non-secret initialization vector; and P_1 is the 1st block of plaintext. Subsequent blocks of plaintext are encrypted as $C_i = E_{key}(P_i \oplus C_{i-1})$. In the same manner, the first block of ciphertext is decrypted as $P_1 = D_{key}(C_1) \oplus \text{IV}$, and the subsequent blocks of ciphertext are decrypted as $P_i = D_{key}(C_i) \oplus C_{i-1}$.

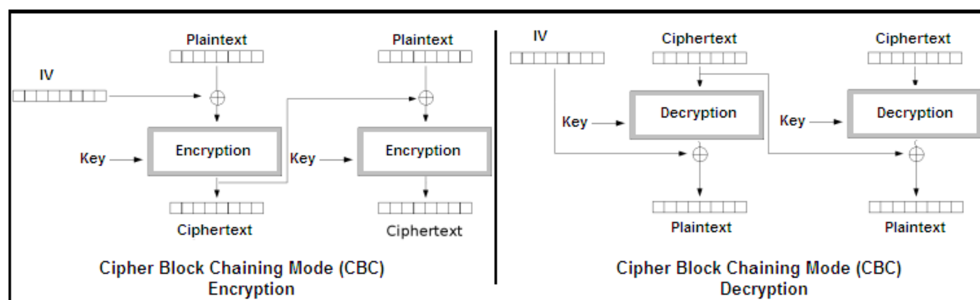

Figure 2. CBC-mode encryption and decryption.

Other block ciphers can be specified in a similar way. Table I lists the various modes of operation, along with the corresponding encryption and decryption specifications.

Table I. Encryption and Decryption Specifications for Various 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	$C_i = E_{key}(\mathrm{IV} \oplus \mathrm{CTR}_i) \oplus P_i$	$P_i = E_{key}(\mathrm{IV} \oplus \mathrm{CTR}_i) \oplus C_i$

2.2.2 Padding. Certain modes of operation, such as ECB, CFB, and CBC mode, require the plaintext to be partitioned into blocks of a static length. Therefore, whenever the last block of the plaintext is partially filled, it is "padded" or appended with predefined bits to form a full block. The padding is encrypted with the data and, after decryption, is automatically removed from the data.

NIST [Dworkin 2001] recommends appending a single '1' bit to the last block in need of padding and then adding as few '0' bits as possible (perhaps none) until the block is full. The padding can be removed unambiguously by the decryption function if either (1) every message is padded, even messages that have a full last block, or (2) the decryption function knows the length of the data.

Another commonly used padding scheme is to pad the data with a string of one to eight bytes to make the last block eight bytes [RSA Laboratories 1999, RSA Laboratories 1993, Housley 2004]. The value of each padding byte will be the number of bytes added to the data. For example, if two bytes must be added to the last block to make the last block eight bytes long, the padding bytes will consist of two 2s.

NIST has recently developed a proposal to extended CBC mode with ciphertext stealing [NIST 2007], which incorporates a padding mechanism that, unlike the two methods described above, does not increase the size of the resulting ciphertext. The mode uses the term ciphertext stealing because padding bits are taken from the penultimate ciphertext block whenever padding is necessary.

For example, consider an example of CBC mode extended with ciphertext stealing during encryption. Figure 3 illustrates what happens when the last plaintext block, P_n, fills only a partial block, where *000..* is a temporary pad, C_{n-1}' equals the leftmost bits in the penultimate ciphertext block (with a size equal to P_n), and C_{n-1}'' equals the rightmost bits in the penultimate ciphertext block. C_{n-1}'' is omitted from the final ciphertext, as it can be re-generated in the decryption process,

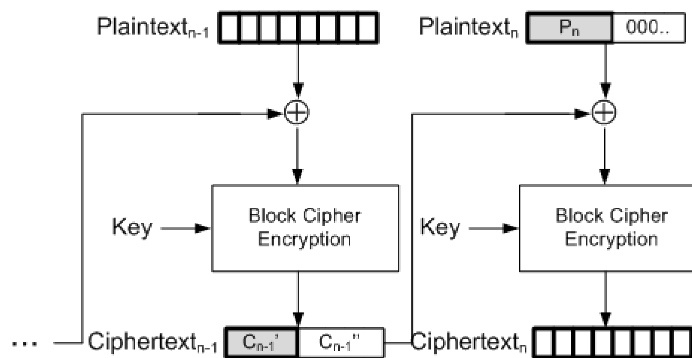

Figure 3. Encryption example using CBC and ciphertext stealing.

2.2.3 Performance Factors. Even when employing a fast block cipher (such as AES), certain modes of operation may not interact well with the file system usage patterns. Although most file accesses are sequential, where the accessed file locations are consecutive, a significant fraction of file references are not and are loosely defined as random file accesses. Therefore, some modes may require decrypting an entire file before a read can be performed at the very end of the file. Conversely, some modes may require re-encrypting an entire file after a write is performed at the beginning of the file. To reduce random access time, certain solutions divide files into extents (contiguous segments of blocks) for a finer encryption granularity. The main tradeoff is introducing complexity in key and IV management.

Knowing the performance characteristics of various modes of operation helps the storage designers understand how mode choices affect confidential storage and deletion later in this survey. Table II compares encryption and decryption performance of the discussed modes of operation. Since the random file access pattern is more general compared to the sequential one, the table only characterizes the performance of various operation modes under random accesses.

Table II. Random-access Performance for Various Modes of Operation

<table>
<tr><th>Mode of operation</th><th>Encryption performance</th><th>Decryption performance</th></tr>
<tr><td>ECB/CTR</td><td colspan="2">Good: ECB and CTR do not depend on previous blocks. Multiple blocks can be encrypted and decrypted in parallel.</td></tr>
<tr><td>CFB/CBC</td><td>Poor: Generating a CFB and CBC ciphertext requires the previous ciphertext block as input. In the case of updates, CFB and CBC require re-encrypting the remainder of a file, since all subsequent ciphertext blocks depend on the current ciphertext block. Thus, encryption is not parallelizable.</td><td>Good: CFB and CBC decryption of one block requires only one previous ciphertext block as input. Multiple blocks can be decrypted in parallel.</td></tr>
<tr><td>OFB</td><td colspan="2">Medium: The key stream (or encryption mask) can be pre-computed independently of the ciphertext. This pre-computation can speed up random file access, as the encryption and decryption of a block does not depend on previous blocks. Once the key stream is computed, encryption and decryption may be performed in parallel.</td></tr>
</table>

2.2.4 Security Caveats. Other considerations when choosing a mode of operation are error resiliency and the overall security of the mode (Table III). Error resiliency is concerned with the propagation of damage when errors occur in a ciphertext block (e.g., the result of a damaged sector on a hard disk). Overall security is concerned with the weaknesses for various modes of operation during attacks that attempt to recover the key, plaintext, or both.

2.2.5 A Side Note about CTR Mode and Key Management. While CTR mode may seem to excel in both security and performance, most encryption systems today use CBC mode. The reason lies in the way the counters are generated and the keys are managed. Each block encrypted or decrypted in CTR mode does not depend on previous blocks. Instead, counters are generated and fed into the encryption algorithm along with the file's key. A counter can be just the block index within a file.

To prevent direct comparisons of ciphered blocks from different files that share a key and the same counter indexing method, a per-file IV is often XORed with each counter. Unfortunately, the counter XORed with IV is not guaranteed to be unique (e.g., files with

the same initial file header), giving attackers multiple counter-IV pairs and ciphertext using the same key for analysis. Using a per-file unique key could be one solution, but also introduces the hard problem of key management in encryption systems, which is a large area of research beyond the scope of this survey.

Table III. Recoverability and Security Characteristics for Various Modes of Operation

Mode of operation	Error resiliency	Overall security
ECB	**Good:** Bit errors in a ciphertext block will only corrupt the corresponding plaintext block.	**Poor:** Two plaintext blocks with the same content yield the same encrypted ciphertext. Ciphertext blocks may also be reordered by an adversary.
CFB	**Good:** A bit error in a ciphertext block affects only two plaintext blocks—a one-bit change in the corresponding plaintext block, and the corruption of the following plaintext block. Remaining plaintext blocks are decrypted normally.	**Good:** As long as IVs are randomly generated and not reused.
CBC	**Good:** A bit error in the ciphertext will corrupt the corresponding plaintext block, and the corresponding bit in the next plaintext block will be flipped. Later plaintext blocks are decrypted normally.	**Good:** As long as IVs are randomly generated and not reused.
OFB/CTR	**Good:** Bit errors in a single ciphertext block will corrupt only the corresponding plaintext block.	**Good:** As long as IVs are randomly generated and not reused.

2.3 SISWG Cipher Modes

This section summarizes modes of confidential encryption designed for hard disk encryption and data at rest. The IEEE Security in Storage Working Group (SISWG) is currently looking to standardize narrow-block and wide-block encryption modes. A narrow-block cipher mode operates on the block size of the underlying cipher (e.g., 16 bytes for AES). A wide-block cipher mode operates on more cipher blocks equal to the size of an underlying disk sector (generally 512 bytes and above). At the time of this writing, SISWG has already gotten the narrow-block encryption specification P1619-2007 [SISWG 2008a] approved through the IEEE and is actively submitting the specification to NIST for it to become an approved mode of operation [NIST 2008]. SISWG is also working on a draft for wide-block encryption P1619.2/D7 [SISWG 2008b] titled "Draft Standard for Wide-Block Encryption for Shared Storage Media."

Both narrow-block encryption and wide-block encryption are tweakable block ciphers. A tweakable block cipher has similar inputs (plaintext and key) and outputs (ciphertext) as the standard encryption algorithms discussed in Section 2.1, with the addition of a non-secret input called a "tweak" [Liskov et al. 2002]. This tweak serves a similar randomization purpose as a nonce or IV in a mode of operation, except at the block cipher level. See Figure 4. Using traditional algorithms, the ciphertext output will not vary during multiple encryption operations using the same plaintext and key as inputs. With a tweakable block cipher, however, using a different tweak value during multiple encryptions with the same key and plaintext values will create different ciphertexts. The authors specify certain goals for tweakable block ciphers, namely that (1) any tweakable block ciphers designed should be just as efficient as non-tweakable

block ciphers, and (2) a tweakable block cipher should also be secure, even if an adversary has control of the tweak value.

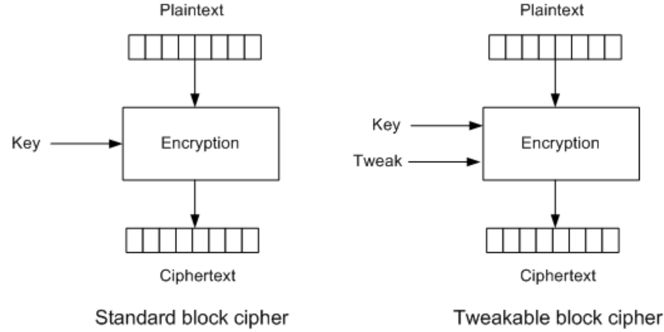

Figure 4. Standard block cipher contrasted with a tweakable block cipher.

2.3.1 Narrow-Block Encryption. The published P1619-2007 draft standard discusses the use of XTS-AES, which is a tweakable block cipher with ciphertext stealing. XTS-AES can be applied to data units of 128 bits or more and uses AES as a subroutine [Bohman 2007]. Data units are divided into blocks of 128 bits, but the last part of the data unit may be shorter than that. With a *key*, *P* as the 128-bit block of plaintext, and *C* as the resulting 128-bit block of ciphertext for block *i*, *t* as the 128-bit tweak value, and *j* as the sequential number of the 128-bit block inside the data unit, the encryption and decryption operations are

$$C_i = E_{key}(P_i,\ t_i,\ j_i),$$
$$P_i = D_{key}(C_i,\ t_i,\ j_i).$$

Specifically, the XTS-AES encryption and decryption operations involve multiplication of the 128-bit result of AES encryption or decryption by the jth power of a primitive element of GF(2^{128}), where GF stands for Galois Field [Menezes et al. 1997]. Unlike other modes of operation, such as CBC, CFB, and OFB, XTS-AES mode can operate in parallel, which may increase performance.

2.3.2 Wide-Block Encryption. The unpublished P1619.2/D7 draft currently specifies Encryption with Associated Data (EAD) methods, such as the EME2-AES transform and the XCB-AES transform. An EAD method consists of an encryption operation that accepts three inputs: a secret key, a plaintext, and the associated data. Associated data is data that characterizes the plaintext in some fine-grained way and does not need to be encrypted. In other words, the associated data is used as a tweak value in that it serves to randomize the result of multiple encryptions with the same keys and plaintexts. However, unlike normal tweakable encryption in which the tweak value has a fixed size, the associated data may have a variable size. The encryption operation returns a single ciphertext value of the same size as the original plaintext. An EAD method also consists of a decryption operation that accepts three inputs: a secret key, a ciphertext, and the associated data. The decryption operation returns a single plaintext value of the same size as the original ciphertext. Like XTS-AES, wide-block encryption operations can be performed in parallel. With a *key*, P_i as the ith block of plaintext, and C_i as the ith block of ciphertext, and where C_0 and P_0 are initial values, the encryption and decryption operations of an EAD method are

$$C_i = E_{key}(AD_i\ ,\ P_i),$$
$$P_i = D_{key}(AD_i\ ,\ C_i).$$

2.4 NIST-proposed Encryption Modes of Operation

This section summarizes other modes of encryption currently being proposed by the National Institute of Standards and Technology (NIST). These modes primarily deal with protecting data confidentiality. Other modes, such as authenticated encryption, are beyond the scope of this survey.

2.4.1 2DEM. The 2D-Encryption Mode (2DEM) encrypts and decrypts binary data in block structures with rows and columns [Belal and Abdel-Gawad 2001]. If DES is chosen as the underlying encryption algorithm, the number of bytes n in each row and column must be a multiple of 8; for AES, 16. 2DEM encrypts data by first performing ECB encryption operations on each row of the block structure (traveling from top to bottom) and producing an intermediary result. It then performs additional ECB encryption operations on each column of the intermediary result (traveling from left to right). The resulting output is the ciphtertext. Decryption is performed on the ciphertext by reversing the encryption operation.

2DEM seems naturally suited for encrypting data of a 2D nature (such as images and tables), but it can be applied to other forms of 1D data by simulating artificial rows and columns. Since each block of data in the row-encrypting phase or the column encrypting phase does not rely on any other block of data, these operations can be parallelized. Bit errors are limited to the encryption/decryption block.

2.4.2 ABC. Traditional modes of operations have good error resiliency (Table III) in that bit errors in the nth ciphertext block will affect at most up to the $(n+1)^{th}$ plaintext block. The Accumulated Block Chaining (ABC) mode, on the other hand, was designed to have infinite error propagation [Knudsen 2000]. As Knudsen states, error propagation modes are best suited "…for situations where errors in transmission are either unlikely to happen or taken care of by noncryptographic means like error-correcting codes, and/or situations where an erroneous data transmission is dealt with by a retransmission."

With a *key*, P_i as the ith block of plaintext, and C_i as the ith block of ciphertext, h as a mapping from n to n bits, and where H_0 and C_0 are initial values, the encryption operations is

$$H_i = P_i \oplus h(H_{i-1}),$$
$$C_i = E_{key}(H_i \oplus C_{i-1}) \oplus H_{i-1}.$$

The decryption operation is

$$H_i = D_{key}(C_i \oplus H_{i-1}) \oplus C_{i-1},$$
$$P_i = H_i \oplus h(H_{i-1}).$$

h may be chosen as a hash function, but Knudsen also argues that choosing $h(X) = X$ or $h(X) = X << 1$ (one-bit rotation) may be sufficient for most applications.

2.4.3 IGE. The Infinite Grappling Extension (IGE) is a special case of ABC where $h(X) = 0$. The IGE mode was first proposed by Campbell [1978] and further analyzed by Gligor and Donescu [2000]. IGE mode was conceived to prevent spoofing attacks in which an attacker would intercept, modify, and retransmit a cipher in such a way that the deception is not detected, even when the attacker does not know the secret key. Any change made to the ciphertext in IGE mode will garble all remaining plaintext during decryption. Campbell suggests placing an expected pattern at the end of the message. If the recipient finds the expected pattern after decryption at the end of the message, the recipient can be assured that the message has not been tampered with.

With a *key*, P_i as the *i*th block of plaintext, and C_i as the ith block of ciphertext, and where C_0 and P_0 are initial values, the encryption operation is

$$C_i = E_{key}(C_{i-1} \oplus P_i) \oplus P_{i-1}.$$

The decryption operation is

$$P_i = D_{key} (P_{i-1} \oplus C_i) \oplus C_{i-1}.$$

2.4.4 FFSEM. Some traditional modes of operation, such as ECB, CBC, and CTR, will only accept as input fixed size plaintext blocks and output fixed-size encryption blocks. The Feistel Finite Set Encryption Mode (FFSEM) [Spies 2008] was designed to encrypt arbitrarily sized data using two components: cycle following and the Feistel method. Cycle following uses a q-bit block cipher to encrypt and decrypt sets of size n where $n < 2^q$. The Feistel method uses a Luby-Rackoff construction [Luby and Rackoff 1988] to turn a fixed-width block cipher into an arbitrary-width block cipher.

An advantage is that FFSEM does not encrypt multiple blocks of data, but is designed to be used where data expansion is not acceptable. Some disadvantages are that FFSEM needs multiple invocations of the block cipher for a single encryption, and different data items can take different amounts of time to encrypt or decrypt due to FFSEM's cycling construction.

3. CONFIDENTIAL STORAGE

Many regulations and acts address the storage of sensitive data. The Gramm-Leach Bliley Act [Federal Trade Commission 1999] requires financial institutions to have a security plan for protecting the confidentiality and integrity of personal consumer data. The Federal Information Security Management Act addresses the minimum security requirements for information and systems within the federal government and affiliated parties. The Health Insurance Portability and Accountability Act mandates provisions to address the confidentiality and security of sensitive health data. These acts and regulations and the threat of possible storage media theft motivates the need for methods of secure data storage. For this survey, we focus on a major component of secure data storage, namely, protecting data confidentiality.

The concept of confidential storage of data may be easy to understand, yet difficult to implement. Achieving confidentiality means storing data in a way that it can be read or deciphered only by authorized persons. No unauthorized persons should be able to read or otherwise obtain meaningful information from this data, even with physical access to the storage media (e.g., a stolen laptop). To limit the scope of this paper, we do not cover cases where an attacker can infer information via indirect channels. For example, one could infer that valuable information exists on a stolen laptop if one finds ciphertext on the laptop's hard drive.

Confidential storage methods are difficult to implement for reasons including complexity of method setup, difficulty of conversion of prior methods to new secure methods, training, overhead and latency in everyday tasks (e.g., reading and writing to files), key management, and password management.

As a brief storage background, Figure 5 shows the storage data paths for popular Unix-based and Windows operating systems. For both platforms, applications reside in user space. When a Unix application makes a call to a file system, the call crosses the kernel boundary and is handled by the Virtual File System (VFS) Layer [Kleiman 1986]. VFS provides functions commonly used in various file systems to ease individual file system implementations, and allows different file systems to co-exist, including local file systems such as ext3 and network file systems such as NFS. Local file systems then proceed to read and write to the block layer, which provides a unified API to access block layer devices.

When a Windows application makes a file system call, that call gets passed to the I/O Manager. The I/O Manager translates application file system calls into I/O request packets, which it then translates into device-specific calls. The File System Drivers are high-level drivers such as FAT and NTFS. These drivers rely on the Storage Device

Drivers, which are lower-level drivers that directly access the storage media. Note that both UNIX and Windows storage data paths share almost one-to-one mapping in terms of their internal structures. Thus, a confidential storage solution designed for one can be generalized to both platforms.

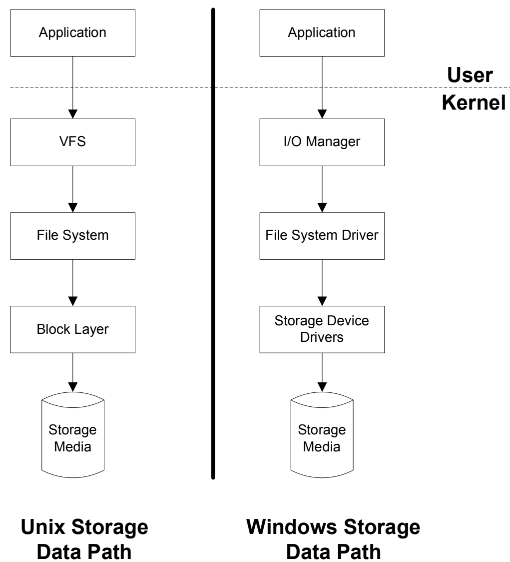

Figure 5. Unix and Windows storage data paths.

The following subsections discuss specific aspects of confidential storage. Section 3.1 demonstrates different types of software-based confidential storage techniques, and Section 3.2 delves into hardware-based confidential storage.

3.1 Software-based Confidential Storage

Software-based solutions to confidential storage require no specialized hardware and are widely available today. These solutions range from encryption programs to cryptographic file systems and block-based encryption systems. Encryption programs reside in user space and employ cryptographic operations at a file granularity. Cryptographic file systems include file systems that are either tailored for encryption or extended for cryptographic functionality. Block-based solutions are tailored to encrypt and decrypt large chunks of data at the granularity of a partition or disk. Each solution has its strengths and limitations with regard to the level of confidentiality, ease-of-use, performance, and the flexibility to set policies. Riedel et al. [2002] and Wright et al. [2003] include further references and discussion on performance and evaluation of software-based confidential storage.

3.1.1 Encryption Programs. Software encryption programs come in two flavors: generalized encryption programs and built-in encryption mechanisms in applications. Generalized encryption programs can encrypt and decrypt files using a variety of ciphers and encryption modes. Some examples are `mcrypt`, `openssl`, and `gpg`. `mcrypt` [Smith 2008] is a simple command-line program intended to replace the old Unix `crypt` program. `openssl` [Young and Hudson 2008] is an open source toolkit that implements the Secure Socket Layer and Transport Layer Security protocols as well as a general-purpose cryptography library. Through use of the library, one may encrypt and decrypt files through the command line. The GNU Privacy Guard, or GnuPG [Koch 2008], implements the OpenPGP standard, which features a versatile key management system

and allows the user to encrypt and sign data and communications using public and private keys.

Many applications also include cryptographic options to protect the confidentiality of files. Examples include the text editor `vim` [Moolenaar 2008] and Microsoft Office products such as Word and Excel [Microsoft Corporation 2003]. These applications either derive the key from the user's system information (such as a password) or prompt for a key or passphrase at the beginning of the session.

While easiest to deploy, application-level solutions have their limitations in regards to level of confidentiality. For example, temporary plaintext files may be created based on the files that are being accessed. Therefore, if temporary files are not deleted, an attacker can simply find them using dead forensic methods. If the temporary files have been deleted but not securely erased (see Section 4), an attacker may use forensic tools to recover the data. File names and metadata are also not encrypted, which may give an attacker information about the type of encrypted data (e.g. size of file, date last modified).

Encryption programs can vary widely in terms of the flexibility of security policies. For example, generalized encryption programs often offer a wide array of encryption algorithms and modes of operation. They can be used on general files and possibly used as a filter in a command pipeline. On the other hand, application-specific encryption programs tend to offer few ways to perform encryption, to limited files types, with limited compatibility with other applications. These characteristics could limit application-specific encryption in terms of the flexibility of changing security policies.

In terms of the user model, generalized encryption programs usually demand much user participation and may not be easy to use. For example, a user must invoke these programs along with the necessary key/password/passphrase every time encryption or decryption takes place. A simple mistake, such as using an incorrect passphrase to encrypt a file or accidental deletion of the private key, may render a file useless and irrecoverable. While this is true of any method using encryption, the chances for error tend to be higher when users can manipulate the encryption key directly. Conversely, application-specific solutions typically prompt for a key/password/passphrase once and tend to perform encryption in a "behind-the-scenes" manner.

In terms of performance, both generalized and application-specific solutions at the user level are slower than other solutions because they do not take full advantage of the VFS layer caching. To be specific, since encryption occurs at the user level, VFS has to cache encrypted data. Thus, unless a user space application caches plaintext itself, it needs to perform decryption and encryption functions on every read and write.

3.1.2 User-Space File Systems. Many user-space file systems take advantage of the Filesystem in Userspace (FUSE) module [Szeredi 2008], which is a Unix kernel module that allows a virtual file system to be built inside a user-space program without having to write any kernel-level code. FUSE intercepts VFS calls and directs them to a user-space file system with added security features before forwarding requests to an underlying legacy file system in the kernel space (Figure 6).

Two examples of FUSE-based secure storage file systems include EncFS [Gough 2008] and CryptoFS [Hohmann 2008]. Both systems are similar in (1) storing encrypted files and file names in encrypted directories, (2) requiring users to mount encrypted directories onto a special mount point with the correct key to see decrypted files and file names, (3) prompting users for a password to generate the encryption key, (4) typically supporting common encryption algorithms such as AES, DES, Blowfish, Twofish, based on what is available in external encryption libraries, and (5) encrypting files on a per-block basis. In cryptographic file systems, encrypted directories are similar to normal directories. They contain encrypted files and other encrypted directories, but generally only directory names are encrypted. Thus, it is possible to traverse an encrypted directory without mounting the encrypted directory by issuing encrypted path names

directly. File system blocks are encrypted with CBC mode in EncFS, and CryptoFS uses CFB mode within extents. Neither file system documents allows its mode of operation to be a configurable option. In [Oprea 2007], EncFS is extended to support the wide-block tweakable cipher CMC [Halevi and Rogaway 2003], as well as other integrity constructions for encrypted storage that are beyond the scope of this survey.

These file systems can provide strong confidentiality by employing good encryption mechanisms and placing temporary files in encrypted directories. One drawback is that user-space file systems still reveal the directory structure and file metadata information, which may help an attacker using dead forensic techniques gather information about the type of data that is encrypted. User-space file systems do allow for some security policy flexibility in that they generally allow for different encryption algorithms, but neither eases key revocation nor changing the mode of operation.

User-space file systems tend to be easier to use than user encryption applications because encryption and decryption happen transparently. Still, users must "mount" encrypted directories to a special mount point to manipulate them. These file systems tend to have higher performance overhead in that requests may need to travel across the kernel and user space multiple times. Since these file systems also rely on an underlying file system, the performance overhead of using user-space file systems is additional to the overhead of using the underlying file system.

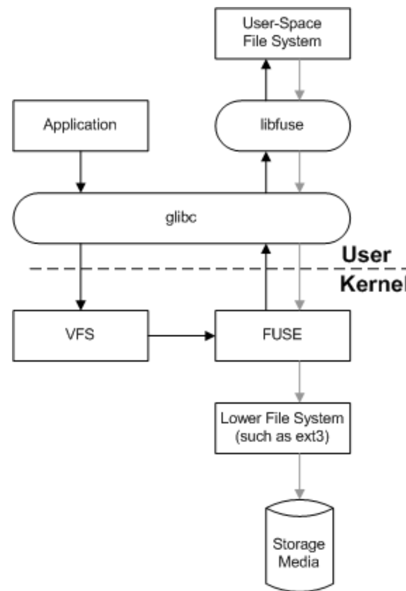

Figure 6. FUSE data path.

3.1.3 Local File Systems that leverage NFS Mechanisms. Some local file systems leverage the existing design of Network File System (NFS) [Sandberg et al. 1985] to ease development, since NFS redirects the operating-system-level traffic to the user space. Two examples that fit our non-distributed, single-user computing environment threat model are the Cryptographic File System (CFS) and the Transparent Cryptographic File System (TCFS). Both systems are described for use on a local machine, even though they could be mounted over a remote machine. However, both systems are somewhat dated. As NFS was designed more for network use between multiple machines, mechanisms such as tunneling NFS through SSH or SSL, or employing NFS over IPsec [Kent and Atkinson 1998] may be employed to provide confidentiality over an external network. However, these techniques will only protect data "on the wire", and not the

data at rest. For examples of network-based file systems beyond the assumed local computing environment of this paper, please see the Self-Certifying File System (SFS) [Mazières et al. 1999], Cepheus [Fu 1999], Secure Network-Attached Disks (SNAD) [Miller et al. 2002], Plutus [Kallahalla et al. 2003], SiRiUS [Goh et al. 2003], FARSITE [Adya et al. 2002], and the distributed IBM StorageTank file system SAN.FS [Pletka and Cachin 2007]. CFS [Blaze 1993] is implemented in the user space, which communicates with the Unix kernel via NFS. Specifically, it uses an unmodified NFS client to talk with a modified NFS server over the loopback network interface. Figure 7 shows the CFS data path.

CFS allows users to "attach" cryptographic keys to directories to make the content within the directory available to the user until the user "detaches" them. While directories are attached, files, filenames, and file paths inside those directories are transparently encrypted and decrypted. Files are encrypted using DES in a hybrid ECB+OFB mode by default, but multiple encryption algorithms are supported. In the standard encryption mode, no IV is used, and files are subject to an identical block analysis attack. In high security mode, the IV is stored in the group ID field of each file's i-node data structure. This mode improves security but precludes different group ownerships of files in the same directory, since the effective group ID is now inherited by the root encrypted directory. Users are responsible for remembering keys, and keys are not stored on disk.

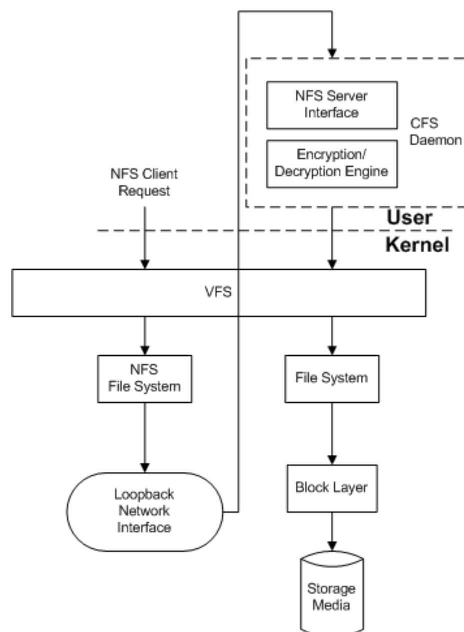

Figure 7. Data flow in CFS.

TCFS [Cattaneo et al. 2001] uses a modified NFS client (in kernel mode) and an unmodified NFS server, which makes it possible (but not necessary) to work with remote machines. TCFS encrypts files by employing a user-specified block encryption module (e.g., AES) in CBC mode, and also includes authentication tags per file to insure data integrity. Only file data and file names are encrypted; directory structures and other metadata are not encrypted. Instead of requiring passphrases, TCFS uses the UNIX authentication system. TCFS implements a threshold secret sharing scheme [Shamir 1979] for reconstructing a group key if a member leaves the group. TCFS is available

under Linux with a 2.2.17 kernel or earlier and stores its keys on disk, which may not be safe.

NFS-based local file systems can employ multiple confidential encryption mechanisms, confidential key solutions in which the decryption key is not stored on disk, and confidential temporary file solutions in which temporary files can be placed in encrypted directories. On the other hand, NFS-based local file systems reveal the directory structure and file metadata (except for the file name), and these file systems are subject to the security vulnerabilities of the underlying network protocols (e.g., an attack on NFS).

NFS-based local file systems enjoy certain flexibilities in the context of implementing and changing security policy settings, including the ability to operate on top of any file system and portability across different systems through the use of the network stack and various network interfaces. These characteristics may aid existing security policies by allowing the use of specific underlying file systems. Some limitations of NFS-based local file systems include not allowing easy key revocation and changes in the mode of operation.

NFS-based local file systems tend to be easier to use than user-space encryption programs because encryption and decryption happen transparently to the user. On the other hand, users must realize that they need to "mount" encrypted directories to a special mount point in order to manipulate them. Thus, the encryption is not entirely transparent.

Due to their need to cross the kernel boundary many times, NFS-based local file systems have perhaps the worst performance numbers compared to other forms of solutions in this survey.

3.1.4 Stackable File Systems. Stackable file systems use a stackable architecture to extend functionality (such as cryptographic functions) by intercepting system calls and routing them to an inserted file system layer. In other words, these file systems run inside the kernel and can operate on top of any other file system without requiring other user-level processes or daemons to run. Some examples of stackable file systems include Cryptfs, NCryptfs, and eCryptfs.

Cryptfs and NCryptfs are applications of FiST, a File System Translator language [Zadok and Nieh 2000]. FiST allows developers to describe stackable file systems at a high level and generates kernel file system modules for various platforms (e.g., Solaris, Linux, and FreeBSD).

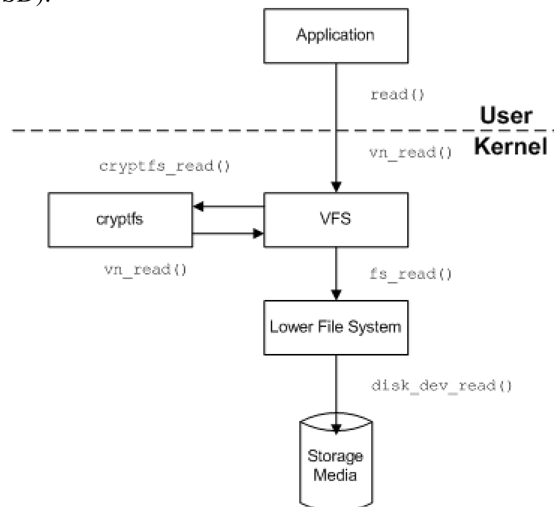

Figure 8. File system call path using Cryptfs.

Cryptfs is "a stackable v-node level encryption file system [Zadok et al. 1998]." The term *v-node* refers to a data structure used to represent a virtual i-node in the virtual file system layer. Cryptfs uses the Blowfish encryption algorithm with 128-bit keys in CBC mode. A tool prompts users for passphrases, which then form encryption keys. File names and directories are also encrypted. The key is not stored on disk. Figure 8 demonstrates how Cryptfs is layered between the user and the underlying file system.

NCryptfs is the successor to Cryptfs and improves on its base design in many ways [Wright 2003]. NCryptfs supports multiple users, keys, authentication modes, and any encryption algorithm that can encrypt arbitrary length buffers into a buffer of the same size (e.g., Blowfish or AES) in CFB mode. It extends key management to support ad-hoc groups, per-process and per-session keys, and key timeouts. NCryptfs uses "long-lived" keys to encrypt data to avoid the penalty of re-encrypting data when a key is changed. Once the key is entered (through a passphrase), it is kept in core memory by NCryptfs and is never revealed to other users. The key is not stored on disk.

One example of a non-FiST stackable file system is eCryptfs, which is a kernel-native stacked cryptographic file system for Linux [Halcrow 2007]. Similar to the FiST stackable file systems, eCryptfs intercepts calls to the existing mounted file system. While intercepting these calls, eCryptfs encrypts and decrypts file data. eCryptfs operates at a lower granularity by encrypting and decrypting individual data extents in each file using a uniquely generated File Encryption Key (FEK). That key is encrypted with the user-supplied File Encryption Key Encryption Key (FEKEK), and the result is stored inside the file's metadata. Users supply the FEKEK either by passphrase or a public key module.

eCryptfs supports multiple encryption algorithms and supports the CBC mode of operation. Each data extent also has a unique IV associated with it. When data is written to an extent, its corresponding IV is changed before the extent is re-encrypted.

Additional eCryptfs i-nodes are kept, which are mapped to the i-nodes in the underlying file system. eCryptfs i-nodes contain cryptographic contexts, including:

- The session key for the target file
- The size of the extents for the file
- A flag specifying whether the file is encrypted

Stackable file systems can employ confidential encryption mechanisms. The primary file encryption key does not have to be stored on disk, and temporary files from applications can be placed in encrypted directories. In terms of limitations, existing stackable file systems reveal the directory structure, as well as file metadata, and often do not allow easy key revocation or a change in mode of operation.

Stackable file systems tend to be easier to use than user-level encryption programs due to the transparency of cryptography to the user. Users must "mount" encrypted directories to a special mount point. Since stackable file systems insert functionality through a layer of indirection, the overhead can be higher than other methods (e.g., a single-layer file system tailored to offer confidentiality). On the other hand, since these file systems run in kernel space, they perform better than file systems that either run in user space or require crossing the kernel boundary multiple times.

3.1.5 Disk-based File Systems. Disk-based file systems operate at a lower level of abstraction than stackable file systems, software-based encryption programs, or NFS-based local file systems. A disk-based file system has full control over all of its directory and file metadata and operations. One example is Microsoft's Encryption File System (EFS).

EFS extends the journaling NTFS file system and utilizes Windows' authentication methods as well as access control lists [Microsoft Corporation 2002, 2008]. EFS is

supported by operating systems based on the Microsoft NT kernel, such as Windows 2000, XP, and Vista. Figure 9 below demonstrates how EFS extends NTFS inside and outside of kernel space.

EFS utilizes both public key and private key encryption techniques. When a file is encrypted for the first time, a unique symmetric per-file encryption key (FEK) is created and is used to encrypt the file. The FEK is embedded in the target file and is then encrypted with the user's public key. FEK can also be optionally encrypted with the private key of a user designated as the "recovery agent." Figure 10 shows an example of encrypted file structure. Decryption involves decrypting the FEK with the user's private key, which is stored in the user's Windows profile. One weakness of this method is that an attacker can recover the user's or recovery agent's private key if the attacker can gain access to the user's account [Microsoft Corporation 2007].

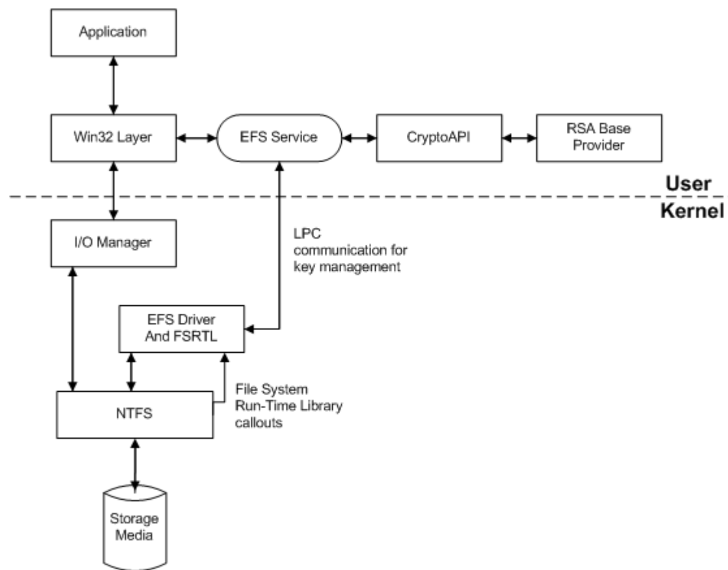

Figure 9. EFS data path.

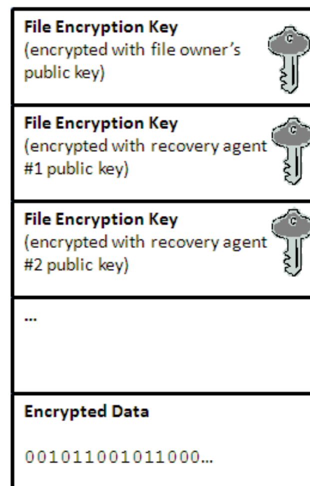

Figure 10. EFS file structure.

EFS encrypts files and FEKs using the DESX encryption algorithm, a variant of DES that increases brute-force attack complexity on the key [Kilian and Rogaway 1996]. Encrypted folders are set with an encryption attribute, and by default all files and folders inside an encrypted folder are also encrypted.

A disk-based file system, when configured correctly, can offer strong encryption mechanisms. In addition, since a disk-based file system has control over its primitives, such as metadata, it can associate a unique key (and possibly even an IV) to each file. Different encryption mechanisms are allowed for security policy creation and implementation, and multiple users may be specified to be able to decrypt certain files. However, a disk-based file system cannot easily change the mode of operation, and often times some file structures and metadata are still revealed (such as with EFS). The user experience is similar to NFS-based local and stackable file systems in that users must know enough of the encryption to place files into confidential directories (or designate directories as confidential directories). Disk-based file systems remove the level of indirection that other cryptographic file systems employ and perform encryption and decryption in the kernel, which utilizes lower level buffers and speed up performance.

3.1.5 Block-based Encryption Systems. Block-based encryption systems work at a lower layer of abstraction than file systems. In other words, these systems work transparently below file systems to encrypt data at the disk-block level. Examples of block-based encryption systems include `dm-crypt`, BestCrypt, the CryptoGraphic Disk driver, the Encrypted Volume and File System, and Microsoft BitLocker Drive Encryption.

`dm-crypt` [Peters 2004] is a replacement for the Linux `cryptoloop` system that works by using the Linux device mapper, an infrastructure introduced in Linux 2.6 to provide a generic way to create virtual layers of block devices on top of real block devices that interact with hardware directly. `dm-crypt` uses the Linux CryptoAPI and supports encrypting block devices such as disks, partitions, logical volumes, and RAID volumes. If one writes random blocks to a block device using `dm-crypt`, an attacker will not be able to discern the locations of encrypted files and the amount of free space left on the device. `dm-crypt` supports encryption algorithms and modes of operation present in the Linux CryptoAPI, which include AES, DES, Serpent, Twofish, Blowfish, ECB mode, and CBC mode. The IV is based on the sector number. An encrypted block device will wait to be mounted until a user passphrase is given. Similar to `cryptoloop` and `dm-crypt`, BestCrypt [Jetico, Inc. 2008] is a commercial example of a block-based encryption system. Other examples include the CryptoGraphic disk driver [Dowdeswell and Ioannidis 2003] for NetBSD and the Encrypted Volume and File System (EVFS) [Hewlett-Packard 2007] for HP-UX.

Microsoft BitLocker Drive Encryption provides encryption for hard disk volumes and is available with Vista Enterprise, Vista Ultimate, and the upcoming Windows Server operating systems [Microsoft Corporation 2006]. BitLocker drive encryption has two goals:

1. To encrypt the entire Windows operating system volume (and additional volumes in the future)
2. To verify the integrity of the boot process using a Trusted Platform Module (TPM)

We are only concerned with the first goal, as boot process integrity is beyond our scope. BitLocker encrypts the specified volume sector-by-sector using AES in CBC mode with a diffuser called Elephant [Ferguson 2006]. The diffuser is stated as necessary due to a weakness in CBC mode, which allows an attacker to flip an ith bit in the next block's plaintext by flipping the ith bit in the current block's ciphertext at the risk

of randomizing the current block's plaintext (Table III). This diffuser runs a series of XOR and rotations on 32-bit words in a block to cause one flipped bit to cause many more random bit flips in the same block. In fact, using this diffuser in a block with at least 128 words will cause full diffusion to occur within the active word with just one bit flip in about one third of a cycle. This means that all bits in the next block's plaintext will be randomly affected by the bit flip, and the attacker cannot selectively control any aspect of the plaintext if any ciphertext is modified. Figure 11 demonstrates Bitlocker encryption of one block, where the drive sector key is the IV and the block size can be any power of two within the range of 512-8,192 bytes, or 4,096-65,536 bits. Two separate diffusers (*A* and *B*) are used in Elepahant. This is because one diffuser has a property of diffusing well (quickly) during decryption, and the other has a property of diffusing well during encryption.

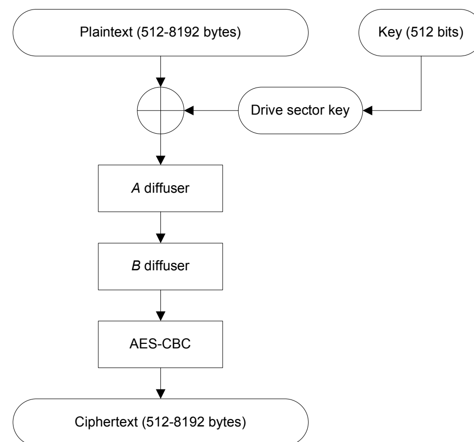

Figure 11: BitLocker encryption of a block.

A BitLocker encryption key can be retrieved either automatically from the TPM chip without the user entering a PIN or from a USB device. If the key is retrieved from the TPM chip, the device will boot up into the operating system.

Block-based encryption systems have many advantages. Once they are set up, they are transparent to the user (except when the user must enter the passphrase) and to the file system. Swap and temporary files are automatically encrypted if the entire block device or volume is encrypted, which is a large improvement in confidentiality over other methods.

Block-based encryption systems can employ confidential encryption algorithms and modes of encryption. Encryption must be performed over the entire disk on all types of data, which makes it very hard for an attacker using dead forensic techniques to discern file structures from data, An attacker may not even be able to discern partition size if the entire disk is first filled with random bits. Keys, encryption algorithms, and mode of operation generally cannot be changed. This characteristic makes it difficult to control the extra encryption granularity (e.g., everything must be encrypted), which may be a challenging obstacle for certain security policies. Performance is rated similarly to disk-based file systems, in that encryption and decryption operations take place in kernel-space and take advantage of lower-level disk buffering.

3.2 Hardware-based Confidential Storage

Hardware-based confidential storage mechanisms differ from software ones in that the cryptographic functionality is either hard-coded into the hardware or into an external specialty device. This method is more rigid in that a user cannot change authentication

mechanisms (such as an encryption key or smart card) or add extra functionality to the storage mechanism, yet it is often much faster than any software-based solution.

Some present-day examples of hardware-based secure storage include secure flash drives, extended cryptographic instruction sets used on specialty CPUs, and hard disk enclosures, as well as PCI/PCMCIA extension cards that act as cryptographic interceptors between the hard disk and the operating system. We discuss these mechanisms, as well as their strengths and limitations, in the following sections.

3.2.1 Secure Flash Drives. The secure flash drive is a relatively new phenomenon on the market today, apparently in response to data and identity theft. Some secure flash drives provide only software encryption using block-based encryption methods as mentioned. Other flash drives protect data through cryptographic mechanisms provided on the flash drive itself. This means that encryption and decryption are transparent to the host operating system, yet a mechanism must be provided (usually via operating system driver or application) to allow the user to input an encryption key (or a password that unlocks the encryption key or keys). Two example products are Ironkey and the Kingston Data Traveler Secure.

Ironkey [2007] uses hardware-based AES encryption in CBC mode with 128-bit randomly generated keys. These keys are generated inside the flash drive in a CryptoChip and are unlocked by a user password. Ten wrong attempts will trigger a “self-destruct” of the encryption keys. Two volumes become available when the Ironkey is inserted: one software volume and one encrypted volume where user data is stored. Password entering (or “unlocking”) software is installed on the software volume, and not on the host computer, yet it must be executed on the flash drive using the host operating system. The Kingston Data Traveler Secure [Kingston Technology 2008] is similar to Ironkey, except that it uses 256-bit keys with AES encryption and allows users to store data on either the encrypted or non-encrypted partition.

Hardware-based encryption flash drives can employ good encryption techniques and, similarly to software block-based encryption systems, directory structure is not revealed. Confidential policy changing is not supported, as encryption algorithms and mode of operation cannot be changed. Keys may only be changed by completely resetting the device. In other words, the hardware cannot be reconfigured to meet changes in confidential policy. Usability is good in that most of the password-entering software seems easy to use and requires the user to enter a password once per session. In terms of overhead, all encryption operations are performed on the flash drives themselves and do not consume CPU cycles and memory on the host machine. Therefore, the performance depends on the speeds of the on-flash cryptographic processing, flash data access times, and the interface used to access secure flash drives.

3.2.2 Enclosures and Extension Cards. Hard disk enclosures and extension cards (either PCI or PCMCIA) have been used for several years as a fast, transparent encryption mechanism for sensitive data. Examples include SecureDisk Hardware [SecureDisk 2008] and RocSecure Hard Drives [RocSecure 2008]. These solutions intercept and encrypt/decrypt data going to and from the hard drive real-time and use a specialized USB thumb drive as the encryption key. The encryption key is generated and placed on the thumb drive by the manufacturer, and often the method of key generation (i.e. randomness technique) is not disclosed.

Enclosures and extension cards can employ good encryption techniques and do not divulge any information about files or the structure of the file system on disk. Similarly to secure flash drives, policy regarding confidentiality changing is not supported since keys, encryption algorithms, and mode of operation cannot be changed. Secure storage is not performed on a per-file level, so the entire hard disk must be encrypted. This characteristic may not be flexible in regards to security policy. A USB thumb drive

acting as a key is simple to use. Similar to secure flash drives, enclosures and extension cards do not consume host CPU cycles and memory, as encryption is done in the enclosure or on the extension card.

3.2.3 Encrypted Hard Drives. Seagate [2006] is introducing "DriveTrust Technology" into their Momentus 5400 FDE series notebook hard drives, which implement full disk encryption. This technology is implemented in the hard drive firmware and provides encryption, decryption, hashing (for passwords), digital signature, and random-number generation functions. Extra-sensitive data, such as keys, can be stored in separate, secure partitions on the hard disk. Any time a user application attempts to access information on a secure partition, it must present its credentials to the "administrator function" on the hard drive, which "authenticates the application, activates the appropriate secure partition, and allows the application to interact with the secure partition through the trusted send/receive command set [Seagate 2006]." The current hard disk supports the AES encryption algorithm.

Once again, confidentiality is good if encryption keys are properly encrypted and stored in a secure manner, and the encryption algorithm, mode of operation, and key size are strong. Flexibility of policy settings is coarse-grained, as entire partitions must be encrypted, and specifications such as encryption algorithm, mode of operation, and key size cannot be changed. Keys also cannot be easily revoked. The user model is simple, in the sense that cryptographic functions occur transparently, and the user must enter a password/passphrase when mounting the hard drive. All encryption and decryption operations happen in the hard disk firmware with a claim of little to no performance penalty.

4. CONFIDENTIAL DATA ERASURE

When the time comes to remove confidential data, we must be sure that once deleted, the data can no longer be restored. A full secure data lifecycle implies that data is not only stored securely, but deleted in a secure manner as well. However, typical file deletion (encrypted or not) only removes a file name from its directory or folder, while a file's content is still stored on the physical media until the data blocks are overwritten. Unfortunately, average users believe that a file's content is erased with its name [Rosenbaum 2000].

Many forensic techniques are available to the determined (and well-funded) attacker to recover the data. CMRR scanning microscopes [Hughes 2004] can recover data on a piece of a destroyed disk if any remaining pieces are larger than a single 512-byte record block in size, which is about 1/125" on today's drives. Magnetic force microscopy and magnetic force scanning tunneling microscopy [Gomez et al. 1992] analyze the polarity of the magnetic domains of the electronic storage medium and can recover data in minutes. For example, when a zero overwrites a one, the actual value will become .95 and when a one overwrites a one it will be 1.05. Another approach is to use a spin-stand to collect several concentric and radial magnetic surface images, which can be processed to form a single surface image [Mayergoyz et al. 2000]. A less well-funded attacker can resort to many drive-independent data recovery techniques [Sobey et al. 2006], which may be used on most hard drives independently of their make. The existence of these recovery techniques makes it mandatory that sensitive data be securely deleted from its storage media.

Another issue is that true erasure may incur high overhead; therefore, security policy should have the flexibility to allow less-sensitive data to use conventional deletion techniques. For example, a user might want to securely delete tax information from his or her computer's hard drive, yet not mind if other files such as cookie recipes could be recovered.

Confidential data deletion can be accomplished in three ways: physical destruction of the storage medium, overwriting all of the sensitive data, and secure overwriting the key of encrypted sensitive data. Each method has its relative strengths, and will be addressed in the following subsections.

4.1 Physical Destruction

One way of deleting sensitive data is through physical destruction. For example, The Department of Defense government document DoD 522.22M [1995] states that classified material may be destroyed by numerous methods including smelting, shredding, sanding, pulverization, or acid bath. Needless to say, these methods will leave the storage medium unusable. The following definitions are supplied by P. Bennison and P. Lasher for destruction of hard drives [Bennison and Lasher 2004]:

- With smelting, the hard drive is melted down into liquid metal, effectively destroying any data contained therein.
- Shredding grinds the hard drive down into small pieces of scrap metal that cannot be reconstructed.
- The sanding process grinds the hard drive platter down with an emery wheel or disk sander until the recordable surface is removed completely.
- Pulverization is the act of pounding or crushing a hard drive into smaller pieces through a mechanical process.
- An acid bath can be used for destruction of data on hard drive platters. A 58% concentration of hydriodic acid will remove the recordable surface of the platter.

Magnetic degaussing is another option that erases data by exposing a hard drive platter to an inverted magnetic field, which leaves data unrecoverable by software or laboratory attempts [OSS-Spectrum Project 2008]. This method also renders the storage media unusable.

Physical destruction methods provide great confidentiality (the physical media is destroyed). On the other hand, the granularity of data destruction is the entire drive. For example, we cannot securely delete only one file using these methods. Therefore, this method does not support flexible security policies. Many of the discussed physical destruction methods require specialized equipment (which may not be easy to obtain) and potential physical removal of the storage media (which may not be easy to perform), so physical destruction may not be straightforward to perform. Conversely, since physical destruction can destroy large amounts of data in a relatively short amount of time, the performance in this sense is quite good (not including the time to acquire equipment for physical destruction).

4.2 Data Overwriting

Another way to remove confidential data is to overwrite the data. Several standards exist for overwriting data on electronic media. NIST recommends that magnetic media be degaussed or overwritten at least three times [Grance et al. 2003]. The Department of Defense document DoD 522.22M [1995] suggests an overwrite with a character, its compliment, then a random character, as well as other software-based, overwrite methods that refer to non-volatile electronic storage as listed below in Table IV.

Peter Gutmann [1996] developed a 35-pass data overwriting scheme to work on older disks that use error-correcting-encoding patterns referred to as run-length-limited encodings. The basic idea is to flip each magnetic domain on the disk back and forth as much as possible without writing the same pattern twice in a row and to saturate the disk surface to the greatest depth possible. Peter Gutmann's 35-pass overwrite technique is demonstrated in Table V.

Table IV. Software-based Methods of Erasing Data on Non-volatile Storage, Defined in the National Industrial Security Program Operating Manual

ID	Erasure method
C	Overwrite all addressable locations with a character.
D	Overwrite all addressable locations with a character, its complement, then a random character and verify.
E	Overwrite all addressable locations with a character, its complement, then a random character.
H	Overwrite all locations with a random pattern, with binary zeros, and then with binary ones.

Table V. Peter Gutmann's 35-pass Overwrite Technique [1996]

Pass number	Data written	Hex code
1	Random	Random
2	Random	Random
3	Random	Random
4	Random	Random
5	01010101 01010101 01010101	0x55 0x55 0x55
6	10101010 10101010 10101010	0xAA 0xAA 0xAA
7	10010010 01001001 00100100	0x92 0x49 0x24
8	01001001 00100100 10010010	0x49 0x24 0x92
9	00100100 10010010 01001001	0x24 0x92 0x49
10	00000000 00000000 00000000	0x00 0x00 0x00
11	00010001 00010001 00010001	0x11 0x11 0x11
12	00100010 00100010 00100010	0x22 0x22 0x22
13	00110011 00110011 00110011	0x33 0x33 0x33
14	01000100 01000100 01000100	0x44 0x44 0x44
15	01010101 01010101 01010101	0x55 0x55 0x55
16	01100110 01100110 01100110	0x66 0x66 0x66
17	01110111 01110111 01110111	0x77 0x77 0x77
18	10001000 10001000 10001000	0x88 0x88 0x88
19	10011001 10011001 10011001	0x99 0x99 0x99
20	10101010 10101010 10101010	0xAA 0xAA 0xAA
21	10111011 10111011 10111011	0xBB 0xBB 0xBB
22	11001100 11001100 11001100	0xCC 0xCC 0xCC
23	11011101 11011101 11011101	0xDD 0xDD 0xDD
24	11101110 11101110 11101110	0xEE 0xEE 0xEE
25	11111111 11111111 11111111	0xFF 0xFF 0xFF
26	10010010 01001001 00100100	0x92 0x49 0x24
27	01001001 00100100 10010010	0x49 0x24 0x92
28	00100100 10010010 01001001	0x24 0x92 0x49
29	01101101 10110110 11011011	0x6D 0xB6 0xDB
30	10110110 11011011 01101101	0xB6 0xDB 0x6D
31	11011011 01101101 10110110	0xDB 0x6D 0xB6
32	Random	Random
33	Random	Random
34	Random	Random
35	Random	Random

Modern hard drives use a different encoding scheme referred to as Partial-Response Maximum-Likelihood (PRML) encoding [Bauer and Priyantha 2001]. Specific overwrite patterns have not yet been developed for the newer PRML encoding.

The number of overwrite passes thought necessary to delete data securely is controversial. Some believe various governmental agencies can recover data that has been overwritten any number of times, but most data recovery companies say they cannot recover data that has been overwritten even once. It is probably safe to say, though, that the more times the data is overwritten, the more secure the deletion.

Three main methods exist to delete data securely from electronic storage media. These methods involve software applications, file systems, and hard disk mechanisms. Their characteristics and relative strengths are discussed in the following subsections.

4.2.1 Software Applications. Three main software methods exist for overwriting sensitive data:

1. Overwrite the contents of a file.
2. Delete the file normally, and then overwrite all free space in the partition.
3. Erase the entire partition or disk.

The first method is probably the quickest method if only a few small files are to be securely overwritten. Many utilities, both free and commercial, are available to perform this operation. Two common UNIX utilities are `shred`, made by the Free Software Foundation, Inc., and `wipe` [Nester 2008]. The `shred` utility will overwrite a file's content with random data for a configurable number of passes (default 25). However, `shred` will not work on file systems that do not overwrite data in place. This can include log-structured file systems [Rosenblum and Ousterhout 1991], journaling file systems (such as JFS, reiserFS, ext3), RAID-based file systems [Hartman and Ousterhout 1995], file systems that take snapshots [Peterson and Burns 2005], and compressed file systems [Woodhouse 2001; Dumesnil 2008]. The `shred` utility will not overwrite a file's metadata.

In contrast, the `wipe` utility will write over file data using the 35-bit patterns recommended by Peter Gutmann [1996]. It will also attempt to remove filenames by renaming them, although this does not guarantee that the old filename (or metadata) will be overwritten. The `wipe` utility has file system limitations similar to those of `shred`.

Overwriting all the free space in the partition is more of an afterthought method and might be employed after files have been deleted the normal way. One example is `scrub`, a Unix open-source utility [Garlick 2008], which erases free space in a partition by creating a file that extends to all the free space. A user needs to remember to remove the file after the application is done. The `scrub` utility implements user-selectable pattern algorithms compliant with the U.S. Department of Defense document 522.22M [1995].

Erasing the entire partition or disk will securely delete all confidential information on the partition or disk such as data, metadata, and directory structures. One such software utility is Darik's Boot and Nuke, or DBAN [2008], which is a self-contained boot floppy that wipes a hard drive by filling it with random data. Depending on the size of the drive, the erasure process can take a long time.

Neither the software file-erasure and free-space-erasure methods will write over previously deleted metadata. Therefore, these methods can still leak confidential information. On the other hand, partition overwriting software will erase all data and metadata, as well as the structure of the file system.

Flexibility of confidentiality policy settings varies among these methods due to different granularities of deletion. For example, it is possible to erase only sensitive files with software file erasure, while partition overwriting securely removes all files and metadata, regardless of their confidentiality requirements.

All three methods are relatively easy to use. The user needs only input a command in order for the secure erasure to take place. However, the user still needs to initiate secure erasure explicitly.

The level of performance can vary with software file erasure since the user has to wait for only chosen files (hopefully small) to be securely overwritten. The other two methods may incur a considerable wait time, depending on the size of the free space and storage partition.

4.2.2 File Systems. Two examples of data overwriting file systems are FoSgen [Joukov et al. 2006] and Purgefs [Joukov and Zadok 2005], which are stackable file systems built in FiST [Zadok and Nieh 2000]. Purgefs can overwrite file data and metadata when deleting or truncating a file. Alternatively, to increase efficiency, the purge delete option can be chosen using a special file attribute, for which only files with such an attribute will be purge deleted. Purgefs will delete data one or more times and supports the NIST standards and all NISPOM overwrite patterns (Table IV).

FoSgen consists of two components: a file system extension and the user mode `shred` tool (Section 4.2.1). FoSgen intercepts files that require overwriting and moves them to a special directory. The `shred` tool, invoked either manually or periodically, eventually writes over the data in the special directory.

The authors of FoSgen have also created patches to add secure deletion functionality to the ext3 file system. The first patch adds one-pass secure deletion functionality, and the second patch supports multiple overwrites and securely deletes a file's metadata. Both implementations work in all three of ext3's journaling modes and erase either a specially marked file's data or all files.

Overwriting file systems can confidentially erase files and metadata using a variety of methods and passes. Users can specify the files and the number of passes and writing patterns for security policies. These file systems are easy to use, because a user only needs to mount the file system with specific options. Unfortunately, depending on the file size, overwriting files may incur a heavy performance penalty.

4.2.3 Semantically-aware Hard Disks. A semantically-smart disk system (SDS) [Sivathanu et al. 2003] tries to understand how the file system uses the disk beyond the information made available by the interface between the two components. In other words, an SDS can have access to certain knowledge of the file system's on-disk data structures and policies and can make intelligent decisions based on this knowledge, such as pre-fetching blocks on a per-file basis. The authors describe a "secure-deleting" SDS that guarantees that deleted file data will be rendered unrecoverable by recognizing deleted blocks through "operation inferencing" and overwriting those blocks with different data patterns a specified number of times. Since it is possible that the file system might immediately reallocate those deleted blocks to another file, the SDS must track those deleted blocks and queue up pending write requests to those blocks until the secure overwrites have finished. The authors also make a note that the ext2 file system must be mounted synchronously to operate correctly.

Using implicit detection techniques, which enable the storage system to infer block liveness information without modification of the file system, Sivathanu et al. [2004] make a next-generation SDS prototype called FADED (A File-Aware Data-Erasing Disk) that operates under asynchronous file systems. Sometimes, though, this method of erasure can be incomplete. In [Arpaci-Dusseau 2006], the authors explain that due to the reordering and reuse inherent in the ext2 file system, the SDS cannot definitively know whether the current contents of a reallocated block are those of the new or old file. Here the authors use a conservative overwrite method to deal with this problem, which erases past layers of data on the block but leaves the current contents on the block. Because of this, the user can never be positive that all deleted data has been confidentially erased immediately after user file deletion.

SDS overwrites deleted files regardless of their need to be deleted securely, which could result in unnecessary performance overhead and limited flexibility with security policies. On the other hand, SDS operates transparently to the user (and operating system), which make them easy to use.

A type-safe disk [Sivathanu et al. 2006] is similar to a SDS system in that it too can perform semantically-aware optimizations by understanding the pointer relationships

between disk blocks that are imposed by high layers, such as the file system. One difference between a type-safe disks and semantically smart disk systems is that, when using a type-safe disk, the file system must be modified in multiple ways, including the use of the expanded type-safe disk API. The authors also describe a secure deletion type-safe disk, which tracks when a block is garbage-collected and overwrites it one or more times. It ensures that the block is not reallocated before it is overwritten by postponing associated updates to the free block bitmap until the overwriting is complete.

Since the file system is modified in [Sivathanu et al. 2006], the authors guarantee that all previously deleted data is overwritten. This characteristic also requires the use of a modified file system, which may not be conducive to use of policies in many environments. A type-safe disk's overwriting granularity characteristics also result in the same unnecessary performance overhead and limited flexibility with security policies as SDS due to the overwriting of all data (sensitive or not). Like SDS systems, though, type-safe disks are easy to use in that they operate in a completely transparent fashion to the user.

4.3 Encryption with Key Erasure

The third way to delete data securely is to encrypt the data and then securely erase the key. The encryption key is often securely deleted using overwriting methods. This combination allows for much faster secure deletion in that only a small key is overwritten instead of the entire file (which could be very large). The downside is the extra encryption/decryption overhead of regular file operations until the file is deleted. Not many specialized solutions exist. One solution [Peterson et al. 2005] is built on top of the versioning file system ext3cow [Peterson and Burns 2005], and is based on the all-or-nothing (AON) transform [Rivest 1997, Boyko 1999]. AON is defined as a cryptographic transform that, given only partial output, reveals nothing about its input. AON is leveraged in the secure versioning file system to make decryption impossible if one or more of the ciphertext blocks belonging to a file (or a file version) is deleted. No commonly used solution of encryption with key erasure that we are aware of exists for general-use file systems.

The policy and performance characteristics of any encryption method with the addition of key erasure are inherited from the base encryption method. The confidentiality characteristic is also inherited from the base encryption method with one caveat: the encrypted data may not stay "deleted" forever if the encryption method used to initially encrypt the data is ever broken. For example, this may occur if a weakness is ever found in the encryption method, or exhaustive search of the key space becomes possible. Also note that if the encryption key is protected by a password and the password is merely forgotten, the strength of the secure deletion is directly correlated to the strength of the password. It is best to delete the encryption key(s) securely through physical destruction or overwriting methods. The ease-of-use characteristic is degraded in that the user must destroy the key explicitly.

5. OTHER CHALLENGES

This section discusses other challenges for implementing confidential storage and deletion.

5.1 Hard-Disk Issues

Two hard-disk-specific issues we must consider in relation to confidential data deletion include bad sector forwarding and storage-persistent caches.

Bad sectors are disk locations that cannot be accessed consistently, developed during the normal use of a hard disk. Bad sector forwarding is performed transparently at the hardware level, in which the firmware identifies and remaps a bad sector to a reserved area hidden from the user through the hard-disk defects table (G-List) [Shirobokov 2006]. In other words, the defective sector is replaced with a sector on a different part of the

hard disk. The defective sector cannot be accessed again by the hard disk itself. Figure 12 demonstrates bad sector forwarding. Data sector 2 has gone bad and has been detected by the hard disk firmware. The hard disk firmware remaps sector 2 to the reserved area sector 0. Now whenever a read or write operation is performed on sector 2, the operation will be mapped to reserved area sector 0.

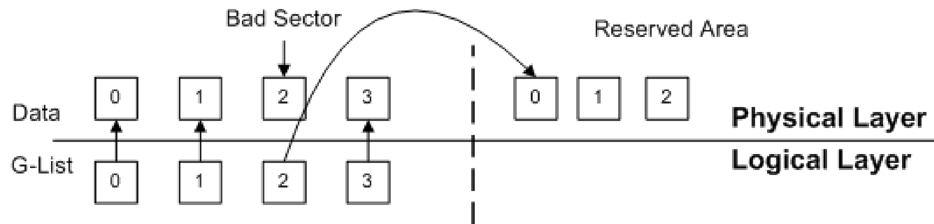

Figure 12: Demonstration of bad sector forwarding.

The problem with bad sector forwarding is that the sector might still be partially readable with only a small number of error bytes. This presents a problem if a bad sector contains a key or IV that could still be read using other forensic methods. SDS systems [Sivathanu et al. 2003; Sivathanu et al. 2004; Arpaci-Dusseau et al. 2006] and type-safe disks [Sivathanu et al. 2006] can address this problem (Section 4.2.3). Unfortunately the ATA specification does not have a command to turn off bad sector forwarding, so vendor-specific ATA commands must be used [Shirobokov 2006].

In addition to bad sector forwarding, persistent caches have been placed in disk-storage systems to improve performance [Joukov et al. 2006]. These caches may not only defer writing to the actual physical media, but may also aggregate multiple writes to the same location on the disk as a single write. In this case, the write cache of the disk must be disabled.

5.2 Data Lifetime Problem

The data lifetime problem addresses the phenomenon of various copies of sensitive data, such as passwords or encryption keys, being scattered all over a computer system during normal system operation [Garfinkel et al. 2004, Chow et al. 2005, Chow et al. 2004]. These locations include numerous buffers (such as string buffers, network buffers, or operating system input queues), core dumps of memory, virtual memory, swap, hibernation values, and unintended leakage through application logs or features. A strategy for secure deallocation of memory is presented in [Chow et al. 2005]. Garfinkel et al. [2004] and Chow et al. [2005] argue that data lifetime is a system-wide issue that must be addressed at *every* level of the software stack.

The attack model in this survey assumes that any attacks to recover sensitive data are staged after the computer has been powered off, so volatile leakage of data such as buffers, queues, and memory are beyond the scope of this survey. However, the recent cold boot suite of attacks [Halderman et al. 2008] demonstrate that encryption keys and other data can be recovered from DRAMs used in most modern computers in between cold reboots. The authors suggest four ways to partially mitigate the attack: continuously discarding or obscuring encryption keys in memory, preventing any sort of memory dumping software from being executed on the physical machine, physically protecting the DRAM chips, and making the rate of memory decay faster. Hibernation files and swap are generally stored on the hard disk and may not go away once the system is powered down. Some block-based encryption methods from Section 3.1.5 may be used to encrypt swap partitions and hibernation files. Hardware encryption enclosures and extensions from Section 3.2.2 and encrypted hard drives from Section 3.2.3 can protect both swap and hibernation as the data is decrypted upon load transparently from the operating system.

6. OVERALL TRENDS

Quite a few general trends emerge when comparing secure storage techniques. These techniques range from solutions closest to the user (application programs) to solutions farthest from the user (hardware-based encryption). We can observe that the level of confidentiality becomes higher as we move responsibility away from the user, which leads to a lower possibility of human mistakes. On the other hand, the flexibility of policy decreases as the solutions move away from the user. In the extreme, all policy decisions are hard-coded in hardware with no room for user configurations. Ease-of-use seems to be correlated to the degree of user involvement and therefore, indirectly, the confidence rating. Performance gains steadily as the method is moved toward the hardware, and then stabilizes when it is in hardware.

Table VI discusses observations of strengths and weaknesses of the confidential storage approaches discussed in this survey. Many of these observations are high-level generalizations, not absolutes. The purpose of the table is to help the reader become aware of the issues involved when designing solutions at different levels of the storage stack.

Table VI. Pros and Cons of Confidential Storage Approaches

Layer	Approach	Pros	Cons
Application	Generalized encryption programs	• Easy to deploy • Offer a wide array of encryption algorithms and modes of operation • Can be used on general files	• Must be careful about temporary files • May be difficult to use • Slower than lower-level approaches
	Application-specific encryption	• Easy to deploy • Easy to use through the application	• Must be careful about temporary files • Offer few ways to perform encryption • Limited compatibility with other applications • Often limited to certain file types • Slower than lower-level approaches
VFS/File system	User-space file systems	• Generally support multiple encryption algorithms • Easy to use but not completely transparent • Users may separate encrypted files and non-encrypted files via directories	• Reveal directory structure and file metadata information • No easy key revocation • No easy way to change encryption or mode of operation once started • Higher performance overhead due to kernel boundary crossings
	NFS-based local file systems	• Generally support multiple encryption algorithms • Operate on top of existing file system • Easily portable	• Reveal directory structure and file metadata information • Subject to vulnerabilities of underlying network protocol

		• Easy to use but not completely transparent • Users may separate encrypted files and non-encrypted files via mount points	• No easy key revocation • No easy way to change encryption or mode of operation once started • Higher performance overhead due to kernel boundary crossings
	Stackable file systems	• Generally support multiple encryption algorithms • Operate on top of existing file system. • Easy to use but not completely transparent • Users may separate encrypted files and non-encrypted files via mount points	• Reveal directory structure and file metadata information • No easy key revocation • No easy way to change encryption or mode of operation once started • Slight performance overhead due to layer of indirection
	Disk-based file systems	• Generally support multiple encryption algorithms • Easy to use but not completely transparent • Users may mix encrypted and non-encrypted files in the same directory • Good performance	• Reveal some directory-structure and file-metadata information • No easy key revocation • No easy way to change encryption or mode of operation once started
Block	Block-based encryption systems	• Generally support multiple encryption algorithms • Directory structure and metadata are not revealed. • Easy to use and transparent to the user • Good performance	• No easy key revocation • No easy way to change encryption or mode of operation once started • All files on the volume must be encrypted
Storage media	Secure flash drives, enclosures and extension cards, and encrypted hard drives	• Generally incorporate a strong confidential encryption algorithm • Directory structure and metadata are not revealed • Easy to use and transparent to the user • Performance not tied to host system	• No way (or very hard) to change encryption key • No way to change encryption or mode of operation • Generally all files on the volume must be encrypted

Confidential deletion techniques contain many tradeoffs. For example, data overwriting techniques have the potential to take a long time. Data encryption with key erasure solves this problem, but introduces cryptography overhead. Solutions that are farther away from user space and require little involvement from users once again tend to be easier to use if the necessary equipment is readily available.

Table VII discusses observations of strengths and weaknesses of the confidential erasure approaches discussed in this survey. Again, these observations are high-level generalizations shown to help the reader become aware of the issues involved when designing solutions at different levels of the storage stack.

Clearly, a combined solution that can store and remove confidential information should have the following ideal characteristics:

- High confidential storage and deletion granularity
- Acceptable performance overhead in terms of storage and deletion
- Enhanced security policy support to enable key revocation, encryption algorithm/mode of operation change and mitigation, and erasure technique
- Confidential storage and erasure of file and directory metadata
- Easy to use with minimal user awareness

While an overarching solution is currently elusive, hopefully, this survey sheds lights on the ground work and considerations to handle confidential data storage and deletion.

Table VII. Pros and Cons of Confidential Erasure Approaches

Layer	Approach	Pros	Cons
Application	Software file erasure	• Overwrite data using standards-compliant patterns • High level of deletion granularity • Easy to use, but user must initiate process	• Will not overwrite previously deleted metadata • Erasure wait time depends on size of file to erase and erasure pattern
	Software free-space erasure	• Overwrites data using standards-compliant patterns • May erase files after they have been normally deleted • Easy to use, but user must initiate process	• Will not overwrite previously deleted metadata • Erasure wait time depends on size of remaining free space on partition and erasure pattern
	Partition-overwriting software	• Will overwrite previously deleted metadata • Overwrites data using standards-compliant patterns • Easy to use, but user must initiate process	• Low level of deletion granularity • Erasure wait time depends on size of the partition and erasure pattern
VFS/File system	File systems	• Overwrites data and metadata using standards-compliant patterns • Easy to use, but not completely transparent	• Erasure wait time depends on size of file to erase and erasure pattern

Storage media	Semantically-aware hard disks	• Completely transparent • Can erase sectors that can only be accessed by the disk	• Confidentiality of erasure depends on modification of the file system and disk API • Erasure wait time depends on size of the partition and erasure pattern • Low erasure granularity
Any layer	Encryption with key erasure	• Fast confidential erasure of files	• Extra encryption/decryption overhead of regular file operations • No generalized solutions exist • Data could be recovered in the future if encryption method develops weakness or exhaustive key search becomes feasible

7. CONCLUSION

This survey took a look at the methods, advantages, and limitations of confidential storage and deletion methods for electronic media in a non-distributed, single-user environment, with a dead forensic attack model. We compared confidential data handling methods using characteristics associated with confidentiality, policy, ease-of-use, and performance. Additionally, we discussed challenges such as hard-disk issues and the data lifetime problem, as well as the overall trends of various approaches. By compiling experiences and constraints of various confidential storage and deletion techniques, we hope that knowledge from research areas that have been evolving independently can cross disseminate, to form solutions that are tolerant to a broader range of constraints.

ACKNOWLEDGMENTS

We thank Theodore Baker, Lois Hawkes, Steve Bellenot, and Mike Burmester for their advice and guidance during this process. Chris Meyers, Cory Fox, Ryan Fishel, Dragan Lojpur, and Mark Stanovich also have contributed to this work. We also thank Peter Reiher and Geoff Kuenning for reviewing an early draft of this survey. 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.

REFERENCES

ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002, 1-14.

ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU R. H., BAIRAVASUNDARAM, L. N., DENEHY, T. E., POPOVICI, F. I., PRABHAKARAN, V., AND SIVATHANU, M. 2006. Semantically-smart disk systems: past, present, and future. In *ACM SIGMETRICS Performance Evaluation Review*, vol 33 n.4.

BAUER, S., AND PRIYANTHA, N. B. 2001. Secure Data Deletion for Linux File Systems. In *The Tenth USENIX Security Symposium*,Washington, D.C., August 2001, 153-164.

BELAL, A., AND ABDEL-GAWAD, M. 2001. 2D-Encryption Mode. In *Second NIST Modes of Operation Workshop*, Goleta, California, USA, August 24, 2001.

BENNISON, P., AND LASHER, P. 2004. Data security issues relating to end of life equipment. In *IEEE International Symposium on Electronics and the Environment*, May 2004, 317- 320.

BLAZE, M. 1993. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Fairfax, Virginia, United States, November 03 - 05, 1993. CCS '93. ACM, New York, NY, 9-16.

BOHMAN, T. 2007. Critical recording benefits from cryptic measures. In *VME and Critical System*, December 2007, 26-28.

BOYKO, V. 1999. 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.

CAMPBELL, C. 1978. Design and specification of cryptographic capabilities. In *National Bureau of Standards Special Publications*, U.S. Department of Commerce, February 1978, D. BRANSTAD, Eds., 54-66.

CATTANEO G., CATUOGNO L., DEL SORBO A., AND PERSIANO, P. 2001. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 2001, 245-252.

CHAUM, D., AND EVERTSE, J.-H. 1985. Cryptanalysis of DES with a reduced number of rounds sequences of linear factors in block ciphers. In *Advances in Cryptology, Proceedings Crypto'85*, LNCS, vol 218, Springer-Verlag 192-211.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 12th USENIX Security Symposium*, 2004.

CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. 2005. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium,* August 2005, 331–346.

Darik's Boot and Nuke. 2008. Homepage. http://dban.sourceforge.net/.

DOWDESWELL, R. AND IOANNIDIS, J. 2003. The CryptoGraphic Disk Driver. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, June 2003, 179-186.

DUMESNIL, A. 2008. e2compr homepage. http://e2compr.sourceforge.net.

DWORKIN, M. 2001. Recommendation for Block Cipher Modes of Operation, Methods and Techniques. NIST Special Publication 800-38A 2001 Edition. National Institute of Standards and Technology, December 2001. Available at http://www.csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

Federal Trade Commission. 1999. Gramm-Leach-Bliley Financial Services Modernization Act, Pub. L. No. 106-102, 113 Stat. 1338 (November 12, 1999), *codified at* 15 U.S.C. §§6801-09.

FERGUSON, N., SCHROEPPEL, R., WHITING, D. 2001. A simple algebraic representation of Rijndael. In *Proceedings of Selected Areas in Cryptography, 2001*. LNCS, vol. 2259, Springer-Verlag, 103–111.

FERGUSON, N. 2006. 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.

FU, K. 1999. Group sharing and random access in cryptographic storage file systems. Master's thesis, Massachusetts Institute of Technology, June 1999.

GARFINKEL, T., PFAFF, B., CHOW, J., AND ROSENBLUM, M. 2004. Data Lifetime is a Systems Problem. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, Leuven, Belgium, September 2004.

GARLICK, J. 2008. Scrub utility project homepage. http://www.llnl.gov/linux/scrub/scrub.html.

GLIGOR, V. AND DONESCU, P. 2000. On message integrity in symmetric encryption. In *1st NIST Workshop on AES Modes of Operation,* October 20, 2000.

GOH, E., SHACHAM, H., MODADUGU, N., AND BONEH, D. 2003. SiRiUS: Securing remote untrusted storage. In *Proceedings of the ISOC Network and Distributed Systems Security (NDSS) Symposium*, February 2003, 131–145.

GOMEZ, R., ADLY, A., MAYERGOYZ, I., And BURKE., E. 1992. Magnetic Force Scanning Tunneling Microscope Imaging of Overwritten Data. *IEEE Transactions on Magnetics*, 28(5):3141–3143.

GOUGH, V. 2008. EncFS: Encrypted file system. http://arg0.net/wiki/encfs.

GRANCE, T., STEVENS, M., And MYERS, M. 2003. Guide to Selecting Information Security Products. NIST special publication 800-36, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland. http://csrc.nist.gov/publications/nistpubs/800-36/NIST-SP800-36.pdf.

GUTMANN, P. 1996. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, July 1996, 77-90.

HALCROW, M. 2007. eCryptfs: a stacked cryptographic filesystem. *Linux Journal,* 156, 2.

HARTMAN, J., AND OUSTERHOUT, J. 1995. The Zebra striped network file system. In *ACM Transactions on Computer Systems*, August 1995, vol. 13, n. 3, 274-310.

Hewlett-Packard. 2007. Encrypted Volume and File System v1.0.02 Release Notes. Hewlett-Packard Development Company, L.P., October 2007. Available online at http://docs.hp.com/en/5992-3353/5992-3353.pdf.

HALDERMAN, J, A,, SCHOEN, S. D., HENINGER, N, CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. 2008. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium* (Sec '08), San Jose, CA, July 2008.

HALEVI, S. AND ROGAWAY, P. 2003. A tweakable enciphering mode. In *Proc. Crypto 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 482–499. Springer-Verlag, 2003. 1.1, 2.1.1, 2.9, 3.5.

HINES, M. 2007. IRS still losing laptops. *InfoWorld*, April 5th, 2007. Available online at http://www.infoworld.com/article/07/04/05/HNirslostlaptops_1.html.

HOHMANN, C. 2008. CryptoFS. http://reboot.animeirc.de/cryptofs/.

HOUSLEY, R. 2004. Cryptographic Message Syntax. Network Working Group RFC 3825, Standards Track, July 2004. Available at http://www.ietf.org/rfc/rfc3852.txt.

HUGHES, G. 2004. CMRR Protocols for Disk Drive Secure Erase. Homepage at http://cmrr.ucsd.edu/people/Hughes/CmrrSecureEraseProtocols.pdf.

Ironkey. 2007. Benefits of Hardware-Based Encryption. Whitepaper, February 2007. Available online at https://learn.ironkey.com/docs/IronKey_Whitepaper-Benefits_of_Hardware_Encryption.pdf.

Jetico, Inc. 2008. BestCrypt software home page. http://www.jetico.com/.

JOUKOV N., AND ZADOK, E. 2005. Adding Secure Deletion to Your Favorite File System. In *Proceedings of the Third International IEEE Security In Storage Workshop*, San Francisco, CA, December 2005.

JOUKOV, N., PAPAXENOPOULOS, H., AND ZADOK, E. 2006. Secure deletion myths, issues, and solutions. In *Proceedings of the Second ACM Workshop on Storage Security and Survivability*, Alexandria, Virginia, USA, October 2006.

KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. 2003. Plutus : Scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, March 2003.

KAUFMAN, C., PERLMAN, R., AND SPECINER, M. 2002. *Network Security: Private Communication in a Public World, 2nd Ed.* Prentice Hall PTR, New Jersey, 2002.

KENT, S. AND ATKINSON, R. 1998. Security architecture for the Internet Protocol. Technical report, RFC 2401, November 1998. Available at http://www.ietf.org/rfc/rfc2401.txt.

KILIAN, J. AND ROGAWAY, P. 1996. How to protect DES against exhaustive key search. In *Proceedings of the 16th Annual International Conference on Advances in Cryptology* (CRYPTO '96, Santa Barbara, CA, Aug.), N. Koblitz, Ed. Springer-Verlag, New York, NY, 252–267.

Kingston Technology. 2008. Data Traveler Secure. http://www.kingston.com/flash/dt_secure.asp.

KLEIMAN, S. R. 1986. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the USENIX Annual Technical Conference.*, Summer 1986, 238–47.

KNUDSEN, L. 2000. Block chaining modes of operation. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, October 20, 2000.

KOCH, W. 2008. The GNU Privacy Guard. http://www.gnupg.org/.

LISKOV, M., RIVEST, R., AND WAGNER, D. 2002. Tweakable block ciphers. In *Advances in Cryptology* (CRYPTO '02), Lecture Notes in Computer Science, Springer-Verlag.

LUBY, M. AND RACKOFF, C. 1988. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386.

MAZIÈRES, D., KAMINSKY, M., KAASHOEK M., AND WITCHEL, E. 1999. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating System Principles,* December 12-15, 1999.

MAYERGOYZ, I., SEPRICO, C., KRAFFT, C., and TSE, C. 2000. Magnetic Imaging on a Spin-Stand. In *Journal of Applied Physics*, 87(9):6824–6826.

MCNEVIN, G. 2007. 2.9 Million US Residents at Risk of ID Theft. *Image and Data Manager*, April 12, 2007. Available online at http://www.idm.net.au/story.asp?id=8270.

MENEZES, A., OORSHOT, P., AND VANSTONE, S., 1997, *Handbook of Applied Cryptography,* CRC Press.

Microsoft Corporation. 2002. Encrypting File System in Windows XP and Windows Server 2003. Technical Report, August 2002. Available online at http://technet.microsoft.com/en-us/library/bb457065.aspx.

Microsoft Corporation. 2003. Microsoft Office 2003 Editions Security Whitepaper. Available online at http://www.microsoft.com/technet/prodtechnol/office/office2003/operate/o3secdet.mspx.

Microsoft Corporation. 2006. Bitlocker Drive Encryption: Executive Overview. Technical Report, August 2006. Available online at http://technet.microsoft.com/en-us/windowsvista/aa906018.aspx#EQB.

Microsoft Corporation. 2007. How to back up the recovery agent Encrypting File System (EFS) private key in Windows Server 2003, in Windows 2000, and in Windows XP. August 24, 2007. Available online at http://support.microsoft.com/kb/241201.

Microsoft Corporation. 2008. How EFS Works. Technical Report, 2008. Available online at http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/distrib/dsck_efs_duwf.mspx?mfr=true

MILLER, E., LONG, D., FREEMAN, W., AND REED, B. 2002. Strong Security for Network-Attached Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

MOOLENAAR, B. 2008. Vim the editor. http://www.vim.org/.

Nester. 2008. Wipe Homepage. http://wipe.sourceforge.net/.

NIST. 2007. Proposal To Extend CBC Mode By "Ciphertext Stealing". Available at http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/ciphertext%20stealing%20proposal.pdf.

NIST. 2008. Request for Public Comments on XTS. National Institute of Standards and Technology, 2008. Available at http://csrc.nist.gov/groups/ST/documents/Request-for-Public-Comment-on_XTS.pdf.

OPREA, A. 2007. Efficient Cryptographic Techniques for Securing Storage Systems. Ph.D. Thesis, Carnegie Mellon University, Technical Report CMU-CS-07-119, April 2007.

OSS-Spectrum Project. 2008. Disposition of Computer Hard Drives: Specifications for Sanitization of Hard Drives, Attachment 2. Available at http://www.spectrumwest.com/Attach2.htm.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks. In *International Conference on Management of Data (ACM SIGMOD)*, Chicago, IL, June 1988, 109-116.

PETERS, M. 2004. Encrypting partitions using dm-crypt and the 2.6 series kernel. *Linux.com*, June 8, 2004. Available online at http://www.linux.com/articles/36596.

PETERSON, Z., BURNS, R., HERRING, J., STUBBLEFIELD, A., AND RUBIN, A. 2005. Secure Deletion for a Versioning File System. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, December 2005,143-154.

PETERSON, Z., AND BURNS, R. 2005. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage 1*, 2 (2005),190–212.

PLETKA, R. AND CACHIN, C. 2007. Cryptographic security for a high-performance distributed file system. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, September 2007, 227–232.

RICHARDSON, R. 2007. CSI Survey 2007: The 12th Annual Computer Crime and Security Survey. Computer Security Institute. Available online at http://www.gocsi.com/forms/csi_survey.jhtml.

RIEDEL, E., KALLAHALLA, M., AND SWAMINATHAN, R. 2002. A Framework for Evaluating Storage System Security. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 15–30, Monterey, CA, January 2002.

RIVEST, R. L. 1997. All-or-nothing encryption and the package transform. In *Proceedings of the Fast Software Encryption Conference*, 1997.

RocSecure. 2008. RocSecure Security Encrypted Hard Drives. http://www.datamediastore.com/rocstor-rocsecure-security-encripted-hard-drives.html.

ROSENBAUM., J. 2000. In Defense of the DELETE Key. *The Green Bag*, 3(4). Available at www.greenbag.org/rosenbaum_deletekey.pdf.

ROSENBLUM, M., AND OUSTERHOUT, J. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

RSA Laboratories. 1993. Cryptographic Message Syntax Standard, Technical Note, PKCS #7, Version 1.5. Available at ftp://ftp.rsasecurity.com/pub/pkcs/ps/pkcs-7.ps.

RSA Laboratories. 1999. PKCS #5 v2.0: Password-Based Cryptography Standard. Available at ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and Implementation of the Sun Network File System. In *Proceedings of the USENIX Annual Technical Conference,* Summer 1985, 119-130.

Seagate. 2006. DriveTrust Technology: A Technical Overview, October 2006. Available online at http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf.

SecureDisk. 2008. Hardware SecureDisk Encryption. http://www.usbgear.com/secure-disk-hardware-encrypted-usb-drive.html.

SHAMIR, A. 1979. How to share a secret. In *Communications of the ACM* , November 1979, vol 22, 11, 612-613.

SHIROBOKOV, A. 2006. Drive Imaging as Part of Data Recovery. Whitepaper, ACE Data Recovery Engineering Inc, October 11th, 2006. Available online at http://www.acedre.com .

SISWG. 2008a. IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, *IEEE Std 1619-2007* , vol., no., pp.c1-32, April 18 2008. Available at http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4493450&isnumber=4493449.

SISWG. 2008b. Standard for Wide-Block Encryption for Shared Storage Media. Draft Standard P1619.2/D7, August 2008. Available at https://siswg.net/index.php?option=com_docman&task=doc_download&gid=134&Itemid=41.

SIVATHANU, M., PRABHAKARAN V., POPOVICI F. I., DENEHY T. E., ARPACI-DUSSEAU , A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, March 31-31, 2003, San Francisco, CA.

SIVANTHANU, M., BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2004. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004, 379-394.

SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. 2006. Type-Safe Disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (OSDI '06). Seattle, WA, November 2006.

SMITH, J. Mcrypt home page. http://mcrypt.sourceforge.net/,

SOBEY, C., ORTO, L., AND SAKAGUCHI, G. 2006. Drive-Independent Data Recovery: The Current State-of-the-Art. In *IEEE Transactions on Magnetics*, vol 42, 2, 1, 188-193.

SPIES, T. 2008. Feistel Finite Set Encryption Mode. NIST Proposed Encryption Mode. Available online at http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffsem/ffsem-spec.pdf .

SQUARE L. 2007. Stolen laptop may hold ID numbers. *The Daily Reveille*, May 3rd, 2007. Available online at http://media.www.lsureveille.com/media/storage/paper868/news/2007/05/03/News/Stolen.Laptop.May.Hold.Id.Numbers-2892874.shtml.

STINSON, D. 2002. *Cryptography: Theory and Practice, 2nd Ed.* Chapman & Hall/CRC, Boca Raton, FL.

SULLIVAN, B. 2005. Help! I left my identity in the backseat of a taxi. *MSNBC: The Red Tape Chronicles*, November 18, 2005. Available online at http://redtape.msnbc.com/2005/11/why_you_should_.html.

SZEREDI, M. 2008. Filesystem in USEr space. http://sourceforge.net/projects/avf.

U.S. Census Bureau. 2005. Computer and Internet Use in the United States: 2003. Available online at http://www.census.gov/prod/2005pubs/p23-208.pdf.

U. S. Department of Defense. 1995. National Industrial Security Program Operating Manual, 5220.22-M, U.S. Government Printing Office, January 1995.

WirelessWeek. 2007. Alcatel-Lucent Loses Computer Disk. *Wireless Week*, May 18th, 2007. Available online at http://www.wirelessweek.com/article.aspx?id=148070.

WHITTEN, A., AND TYGAR, J. D. 1999. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Washington, D. C., August 1999, 169-184.

WOODHOUSE, D. 2001. JFFS: The Journaling Flash File System. In *Proceedings of the Ottawa Linux Symposium*. RedHat Inc., 2001.

WRIGHT, C., DAVE, J., AND ZADOK, E. 2003. Cryptographic file systems performance: What you don't know can hurt you. In *Proceedings of the IEEE Security in Storage Workshop (SISW)* (October 2003), pp. 47–61.

WRIGHT, C. P., MARTINO, M. AND ZADOK, E. 2003. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, June 2003, 197-210.

YOUNG, E., AND HUDSON, T. 2008. OpenSSL project home page. http://www.openssl.org/.

ZADOK, E., BADULESCU, I. AND SHENDER, A. 1998. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

ZADOK E., AND NIEH, J. 2000. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, June 2000, 55-70.

Received October 2008; accepted February 2009