ENCRYPTING VOLUMES

CHAPTER 9

723025.

ABOUT THIS CHAPTER

In this chapter, we look at the problem of protecting an entire storage device, as opposed to protecting individual files. We look at the following:

- Risks and policy alternatives for protecting drive contents
- Block ciphers that achieve high security
- Block cipher encryption modes
- Hardware for volume encryption
- Software for volume encryption

9.1 Securing a Volume

When we examined file systems in Section 5.1, Eve had asked to borrow a USB drive from which all data had been deleted. Eve could have tried to recover private bookkeeping files from the drive. We can avoid such risks and protect everything on the drive, including the boot blocks, directory entries, and free space, if we encrypt the entire drive volume.

Note how we use the word *volume* here: It refers to a block of drive storage that contains its own file system. It may be an entire hard drive, a single drive partition, a removable USB drive, or any other mass storage device. If the system "sees" the volume as a single random-access storage device, then we can protect it all by encrypting it all. We sometimes call this ***full-disk encryption*** (FDE) because we often apply it to hard drives, but it applies to any mass storage volume (Figure 9.1).

File encryption lets us protect individual files from a strong threat. If we encrypt a particular file, then attackers aren't likely to retrieve its contents, except in two cases. First, there is the file-scavenging problem noted previously. Second, people often forget things, and file encryption is a forgettable task.

In many cases, sensitive data is vulnerable simply because nobody bothered to encrypt it. Even when users encrypt some of their data files, it's not likely that they have encrypted *all* of their sensitive files. It is challenging to identify the files at risk and to remember which ones to encrypt.

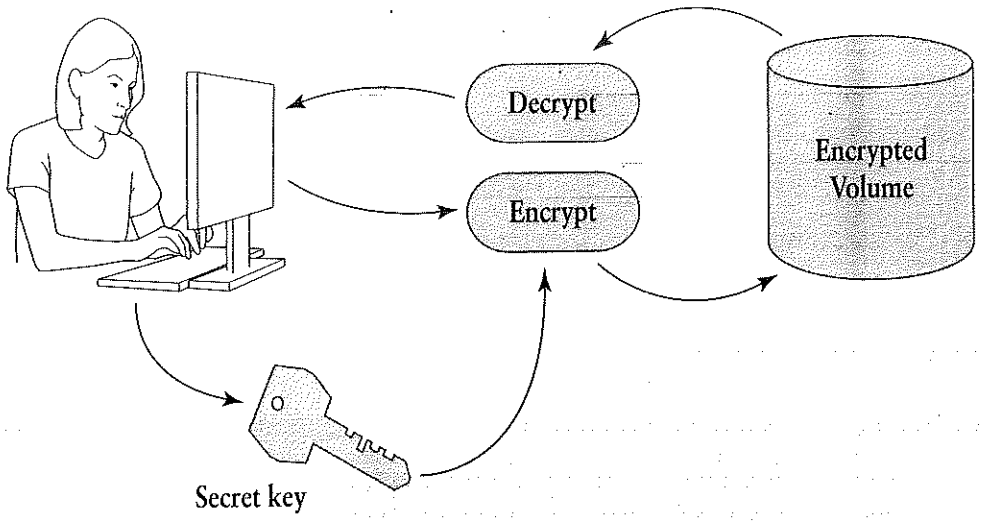


Figure 9.1

A hard drive volume with full-disk encryption (FDE).

The principal benefit of encrypting a whole volume is that the encryption takes place automatically. When we plug in a removable encrypted drive or we start up the operating system, the drive encryption system retrieves the volume's encryption keys and mounts the volume. If the keys aren't available, then the volume can't be used. Once the volume is mounted, the data is encrypted and decrypted automatically. The user doesn't need to decide which files to encrypt. In fact, the user doesn't even have the choice of saving a plaintext file; *everything* on the volume is encrypted, including the directories and free space.

Volume encryption is convenient but it does not solve every security problem. For example, it protects Bob if the attacker physically opens up his tower computer and connects directly to his hard drive. Disk encryption does not protect Bob from a Trojan program that copies files to a separate, unencrypted storage device, like a USB stick.

To put volume encryption in context with other security measures, we look at risks and policy trade-offs. First, we will look at risks facing an unencrypted volume. Next, we look at policy trade-offs between volume encryption, file encryption, and file-based access control.

9.1.1 Risks to Volumes

There are three general risks we address with volume encryption:

1. Losing the storage device
2. An eavesdropper that looks at the volume without the operating system in place
3. Discarding a hard drive or other device without wiping it

The first risk is everyones' worst-case scenario: Our computer is stolen. Laptops especially seem likely to "sprout legs and walk away." While our first worry might be the

cost of replacing the laptop, many people also worry about identity theft. Some thieves exploit the data stored on the hard drive to masquerade as the laptop's owner. Identity theft is a booming business, and personal computers often contain extensive financial records and lists of online passwords.

The theft risk poses even greater problems for large organizations. A stolen laptop may contain large databases of sensitive information about customers. According to statistics collected for the first half of 2009, there were 19 incidents in the United States where a misplaced or stolen laptop placed databases at risk. The incidents involved private companies, schools, hospitals, and government agencies at all levels. The smallest data loss involved 50 personal records, while one incident involved over a million records.

If a stolen computer contains a customer database, many companies are required legally to contact all affected customers and warn them that their customer information may have been stolen. If the database includes credit card numbers or Social Security numbers, then the company may need to provide some defense against possible identity theft.

Eavesdropping

The eavesdropping problem arises if we can't always keep our drive physically safe, or under appropriate software protection. When Bob and Alice started using file encryption, they decided to keep the encrypted file on a shared USB drive. Because the file was encrypted, they assumed it was safe from eavesdropping. For convenience, they left it on the bulletin board in their suite.

We still may risk eavesdropping even if we don't share a drive on a bulletin board. If we back up the data to a separate drive, we need to physically protect the backup drive. If attackers have physical access to our computer, then we can try to protect our data by protecting our Chain of Control; we can disable booting of other drives and enable file protections. However, these defenses don't protect us if the attacker cracks open the computer's case. An attacker can bypass any BIOS or operating system by connecting the hard drive to a different motherboard.

Discarded Hard Drives

When we upgrade a computer, we often trade up to a larger hard drive. The old drive disappears along with the old computer, but what about our sensitive data?

Research at MIT by Simson Garfinkel found that countless companies and individuals sell or discard computers without properly erasing the hard drives. The researchers found that most people took very simple steps to "clean" their drive of data before passing the computer onto the next owner. Using "undelete" programs and other file-recovery tools, researchers were able to recover numerous personal and business files.

There are four strategies for cleaning a hard drive of personal data:

1. Delete personal files and "empty the trash."
2. Reformat the hard drive.
3. Run a "disk wipe" program.
4. Physically damage the hard drive so that it can't be read.

The first two strategies yield similar results. As noted earlier, files continue to exist in a ghostly form after they have been deleted. The same is true for reformatting. When we reformat a drive, especially a "quick format," we simply rewrite the first few blocks of the hard drive with a new directory that contains no files. Disk-recovery software looks at the other blocks on the disk and tries to reconstruct the previous directories. The attempt often succeeds. Once the directories are restored, the software recovers as many files as possible.

Either of the last two strategies may eliminate the risk of recovered files. We examined disk wiping as part of encrypting a file in Section 7.4.3, but there are also utility programs that wipe a drive by overwriting every one of its blocks. Some hard drives have a built-in mechanism to wipe the drive, but these mechanisms are rarely evaluated for effectiveness. Even with the built-in wipe, an attacker might be able to retrieve data.

Note that it may be challenging to physically destroy a modern hard drive. The major components are metal and are assembled to operate at very high rotational speeds. For all but the more extreme threats, we render the data irrecoverable if we open the drive case and remove all of the drive's read/write heads.

9.1.2 Risks and Policy Trade-Offs

Full-disk encryption clearly protects against scavenging data off of an unprotected hard drive. However, just as file encryption was no cure-all, drive encryption doesn't guarantee our safety. Depending on the risks, we may need one or the other or both.

Simply because some risks call for one type of encryption or another does not mean that the encryption solves our problems. Encryption introduces practical problems, especially with respect to key handling.

At this point we have three practical security measures: access controls enforced by the operating system, file encryption using an application program, and drive encryption. We can implement all three, or some combination, depending on the risks. Table 9.1 summarizes the individual alternatives.

Table 9.1 answers the following questions about whether the particular security measure protects against the particular risk:

1. **Hostile users.** Does it protect against hostile, logged-in users trying to steal information from other users on the computer?
2. **Trojans.** Does it protect against data leaking through Trojan horse applications?
3. **Trojan crypto.** Does it protect against Trojan software embedded in the crypto software? The general answer is NO.
4. **External files.** Does it protect files that we copy onto removable storage and carry to other computers?
5. **Lost control.** If attackers take physical control of the computer and physically disassemble it to attack the hard drive, do we have any protection?
6. **Theft.** This is the ultimate case of lost control. Do our files remain protected?
7. **Recycling.** When we want to sell, discard, or recycle the drive, do our files remain protected so that a scavenger or new owner can't recover them?

TABLE 9.1 Effectiveness of access control and encryption

Risk	Access Control	File Encryption	Volume Encryption
Hostile users	YES	YES	NO
Trojans	NO	YES	NO
Trojan crypto	NO	NO	NO
External files	NO	YES	YES
Lost control	NO	YES	YES
Theft	NO	YES	YES
Recycling	NO	YES	YES

Access control clearly plays a role on computers where we have to protect data against hostile users. We also could use file encryption as an alternative to access control. This isn't a good idea in practice. We can only protect files with encryption if we always remember to encrypt them. People often forget to do such things and leave information unprotected. Users must also erase completely all plaintext copies of the sensitive data. This is very hard to ensure.

Moreover, we can't use file encryption to protect critical operating system files or even our application programs. If the operating system doesn't provide access controls, then there's no way to prevent someone from installing a Trojan inside our file encryption software.

Volume encryption provides no real protection against hostile users or Trojans. If the hostile users are on the encrypted system, then they are *inside* the boundary protected by the encryption. They see everything as plaintext anyway. If the access control doesn't restrict hostile users, then volume encryption won't improve matters.

Volume encryption becomes important when we risk losing control of our storage device. Clearly, access control gives little protection if the attacker uses a different operating system or special software to retrieve data from a hard drive. File encryption may protect the encrypted files, but risks remain: We might have forgotten to encrypt a critical file, and there are still risks of scavenging information from scratch files.

Identifying Critical Data

There are several ways to identify critical data. In some cases we know already which files we need to protect and which aren't important. However, it's best to develop policy statements that help define critical data.

For example, if we handle information that others deem sensitive (like credit card numbers, health information, or proprietary information), we are obliged to safeguard it. In some cases, there are specific security measures we must use. In all cases, however, we

need a way to tell which files might contain the sensitive information, so that we may protect those files.

Critical data should always be marked as such. We should clearly distinguish between critical and noncritical data. This makes it easier to ensure that we adequately protect critical data. For example, we may restrict critical data to specific folders or to folders with particular names.

We also may put readable markings in documents to remind us they contain sensitive information. For example, government agencies routinely put classification markings or other warnings about restricted distribution in the document's page header and/or footer. Law firms may do this for documents that are restricted according to judicial orders. Many companies systematically mark all proprietary information.

We also should physically mark any removable storage volumes that might contain sensitive or critical information. Even if the volume is encrypted, it is essential to easily tell whether sensitive information should—or should not—be stored on a particular volume.

Policy for Unencrypted Volumes

If an unencrypted volume contains data we can't afford to leak, then we must guarantee its physical protection. It is very hard to guarantee such protection in practice. We often need to leave hardware unattended, even laptops. This opens the risk of a peeping Tom even if we aren't risking theft. If we don't encrypt the drive and we store sensitive information on it, then we need the policy statements shown in Table 9.2.

In practice, the policy statements should be more specific about the information and storage. Somewhere, our policy should identify what we mean by sensitive data. We should also be more specific about which mass storage devices we protect.

These policy statements are easy to write but difficult to implement. If we keep the drives locked in a private office, we may also have to consider the risks of cleaning people. If the drive is in a laptop, we need a locked cabinet for it, and we should avoid carrying it in public.

Policy for Encrypted Volumes

In practice, we can justify volume encryption for almost any computer hard drive or USB drive. Although it may not protect against every attack, it prevents many common risks.

TABLE 9.2 Policy statements for an unencrypted volume

Number	Policy Statement	Risks
1	The mass storage device shall be physically protected against eavesdropping and theft at all times.	4, 5, 6
2	The mass storage device shall be completely wiped of data before it is recycled or discarded.	7

TABLE 9.3 Policy statements for volume encryption

Number	Policy Statement	Risks
1	Every removable storage volume used for sensitive data shall be fully encrypted.	4
2	The computer's system drive shall be fully encrypted.	5, 6
3	There shall be a mechanism that fully erases and purges the encryption key used to encrypt any mass storage volume.	7

In most cases, we *cannot* keep a hard drive or laptop safe continuously. As of 2010, however, volume encryption remains a relatively rare and sophisticated feature, though this should change as the technologies evolve.

Table 9.3 presents policy statements for volume encryption. As with the previous statements, a practical policy needs to tell people how to identify sensitive data. Policy and procedures also should provide a way to distinguish between volumes containing sensitive data and those that don't.

Now that we know we need volume encryption, we need to figure out how to implement it. Volume encryption places a serious burden on an encryption algorithm. Most algorithms become vulnerable if we encrypt a great deal of information, especially if it includes known plaintext and duplicated information. This is exactly what happens on a large, encrypted hard drive.

9.2 Block Ciphers

The strongest and most respected modern ciphers are ***block ciphers***. While stream ciphers encrypt bit by bit, block ciphers encrypt fixed-sized blocks of bits. The block cipher takes the block of data and encrypts it into an equal-sized block of data. The old Data Encryption Standard worked with 64-bit blocks of data. The new Advanced Encryption Standard works on 128-bit blocks. To apply a block cipher, we must break the data into block-sized chunks and add padding as needed to match the data to the block size (Figure 9.2).

In a sense, block ciphers are like the Caesar cipher examined in Section 7.2. We substitute one letter for another, no more and no less, based on the key. We could in theory create a "codebook" for a given key that lists all plaintext values and the corresponding ciphertext values. The book would be impractically huge. For example, if we have a 64-bit block size, the book would have 10^{19} entries. We also would need a different book for each key.

Building a Block Cipher

Encryption is a systematic scrambling of data. Most block ciphers encrypt by applying a function repeatedly to the plaintext to encrypt. Each repetition is called a ***round***.

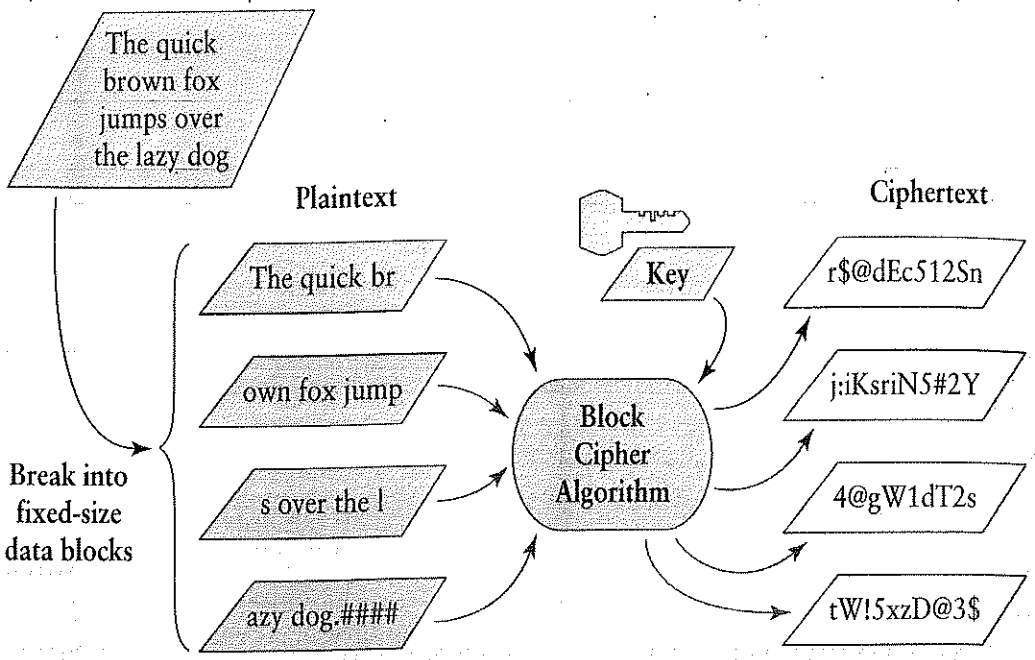


Figure 9.2

A block cipher encrypts data in fixed-sized blocks.

Before the cipher scrambles the data, it takes the key and produces a ***key schedule.*** In some cases, the schedule consists of small blocks of the key that the cipher applies to successive rounds. Often, however, the algorithm performs ***key expansion*** to yield a schedule that is larger than the original key.

Typically, each round uses a successive part of the key schedule. The rounds repeat until the key schedule is exhausted. Here is a general description of the block encryption procedure:

- Generate the key schedule.
- Divide the key schedule into subsections, one per round.
- For each subsection, perform a round:
  - For the first round, take the plaintext as the input text; for the remaining rounds, take the output of the previous round as the input text.
  - Take the next unused subsection of the key schedule as the key input.
  - Scramble the input text using permutations and substitutions as specified in the encryption algorithm.

The round scrambles the data by applying arithmetic and logical operations, particularly the xor operation. A round often uses both permutations and substitutions to scramble the data. The substitutions may be controlled by special data structures called *S-boxes*. These serve as look-up tables to provide the bits to substitute for particular input bits, given a particular key.

The decryption procedure takes the key and the ciphertext and unscrambles the data. Each round tries to undo the work of a corresponding round in the encryption process.

In some algorithms, decryption simply applies the key schedule, the permutations, and the S-boxes in the reverse order.

This procedure highlights three important features of block ciphers:

1. It takes time to change keys, because the procedure must generate a new key schedule for each key.
2. The number of rounds reflect a trade-off between speed and security. More rounds may scramble the data more thoroughly, but each round takes time to execute.
3. Key sizes, block sizes, and the number of rounds performed are built into the procedure. We can't arbitrarily change any of these without redesigning the procedure.

To highlight the third point, compare the flexibility of RC4 to the flexibility of AES. In RC4, the key may contain as few—or as many—bytes as desired. Fewer key bytes provide dramatically inferior encryption. AES supports exactly three key sizes: 128 bits, 192 bits, and 256 bits. The AES procedure performs 10 rounds for a 128-bit key, and additional rounds when using longer keys.

With RC4, we can encrypt a single bit, gigabytes, or more. AES encrypts data in fixed-sized blocks of exactly 128 bits. If we encrypt less than 128 bits, we must pad the data out to 128 bits. AES ciphertext must be a multiple of 128 bits. If we omit part of a ciphertext block, we won't be able to decrypt it correctly.

Despite the inflexibility, AES provides far better encryption than any known stream cipher. Researchers have found numerous shortcomings in RC4. Researchers have studied AES at least as thoroughly, though over a shorter time period, and have found no significant weaknesses.

The Effect of Ciphertext Errors

In Section 7.3.3, we noted how ciphertext errors affect stream ciphers. If we change a single bit of ciphertext, then the corresponding bit will change in the plaintext.

Block ciphers behave differently. There is no one-to-one relationship between individual bits in the plaintext and the ciphertext. The block cipher behaves more like a one-way hash; if we change a single bit in the plaintext, the resulting ciphertext will change dramatically.

Thus, if we encrypt with a block cipher and an error or attack changes a single bit in some ciphertext, it scrambles the entire plaintext data block. In a well-designed block cipher, a change to one input bit will affect bits throughout the output block. This is true when encrypting or decrypting.

Consider, on the other hand, what happens if we rearrange entire blocks of ciphertext. If we rearrange ciphertext bits in a stream cipher, every bit we change will decrypt incorrectly. Stream cipher decryption is sensitive to the order of the data; each bit in the ciphertext must line up with its corresponding bit in the key stream.

In block ciphers, however, we encrypt and decrypt each block independently. Thus, if we change the order of blocks in the ciphertext, we will recover the reordered plaintext as long as we move whole blocks of ciphertext. If we rearrange ciphertext within a block,

we lose the entire block when decrypted. As long as whole blocks remain unchanged, the ciphertext decrypts cleanly into plaintext.

9.2.1 Evolution of DES and AES

Section 7.3 introduced the Data Encryption Standard and controversies surrounding its key size. DES was a block cipher that worked on 64-bit blocks using a 56-bit key. The algorithm evolved from "Lucifer," an encryption algorithm developed by IBM. Lucifer implemented rounds, permutations, and S-boxes in a form called a "Feistel structure." It also used a 128-bit key.

DES and Lucifer

In the 1970s, the NSA was the only organization in the U.S. government with cryptographic expertise, so it was naturally given the job of assessing the algorithm. NSA recommended two changes to Lucifer before making it a national standard. One change was to the S-boxes. A second change was to reduce the key size to 56 bits. The changes were made before the algorithm was approved as a U.S. standard.

As noted earlier, critics immediately decried the 56-bit key as too small. Researchers started cracking DES keys successfully in the 1990s.

The changes to the S-boxes also caused concern. The NSA instructed IBM not to explain why the changes were made. Some observers feared that the changes would give the NSA a secret shortcut to crack DES encryption. If it existed, the shortcut would allow the NSA to crack DES encryption without performing a brute force attack that tried every possible DES key. Any shortcut in a brute force attack would render DES ineffective.

These worries never came true. No shortcut was ever uncovered. Although cryptanalysts have found attacks that work against DES, none of these attacks are significantly better than a traditional brute force attack.

Looking back, the principal weakness in DES has been its key size. The design concepts embodied in DES have stood the test of time. DES has served as a good, working example for future cryptographic designers.

DES was designed to be efficient in hardware. Aside from its key size, this may be its only significant shortcoming. The algorithm required so much low-level bit manipulation that some experts believed—or at least hoped—that DES software implementation would be too slow to be practical. This proved untrue. Software implementations flourished despite the inefficiencies.

Triple DES

To minimize the risk of the 56-bit encryption key, some organizations tried applying DES several times with different keys. The American National Standards Institute (ANSI) developed a series of standards for using DES. These standards were adopted by the U.S. banking industry. One of these standards included a technique that encrypted data with two or three 56-bit keys instead of a single key (Figure 9.3). This was called Triple DES.

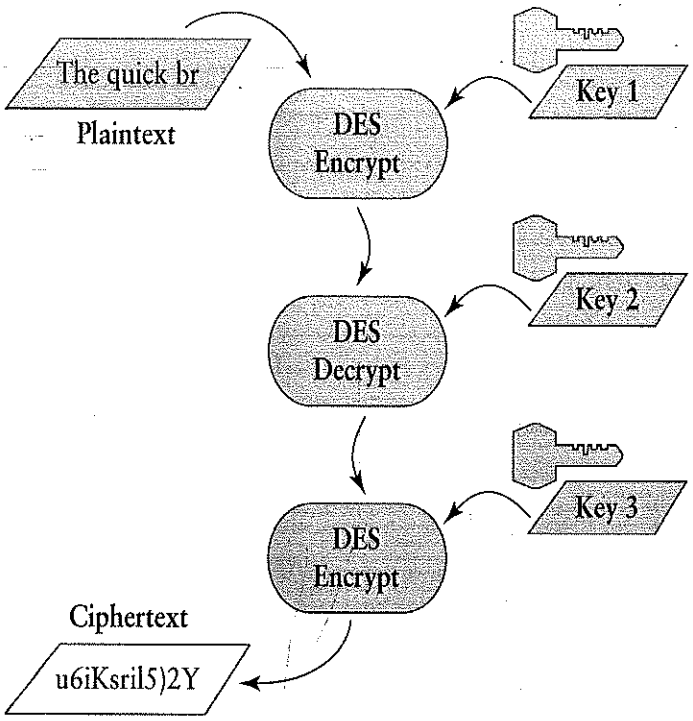


Figure 9.3

Triple DES encryption.

In Triple DES, each DES operation may use a separate 56-bit key. The technique does not simply apply encryption or decryption three times in a row. Instead, the middle step performs the opposite function of the outer steps. The swapped step allows users to operate Triple DES with one, two, or three different 56-bit DES keys.

Here is how we use different numbers of keys with Triple DES:

- One key: Use the same key in all steps. The first two steps invert each other, and the final step performs the encryption.
- Two keys: Use the first key for the first and last steps, and use the second key in the middle step.
- Three keys: Use a different key for each step.

When using three keys, Triple DES requires a 168-bit random key. This may sound significantly stronger than AES with its 128-bit key. In fact, there are attacks on Triple DES that reduce the strength of the 168-bit key to be closer to 112 bits. If Triple DES really worked at the full strength of its key, it would take 2^{168} trials to decrypt the ciphertext. In fact, a more efficient attack only takes 2^{112} trials. When using two keys, Triple DES requires a 112-bit random key. Clever attacks, however, can recover the ciphertext in 2^{80} trials.

Triple DES is less efficient because it performs DES three times. The extra cycles improve security over the original 56-bit key, but the improvement costs a great deal in computing time.

The Development of AES

As DES began to show its age in the 1990s, NIST began the process of replacing it. Unlike the closed, secretive process that produced DES, NIST announced an open competition to produce its replacement. The process for selecting the replacement, called the Advanced Encryption Standard, or AES, had several important features:

- Anyone, anywhere, could submit an AES candidate algorithm. Authorship was not limited to U.S. citizens.
- AES candidates must be free of patents or other intellectual property restrictions.
- The design rationale for every AES candidate would be made public, including the rationale for choosing built-in constants, for designing S-boxes, and so on.
- All "unclassified" evaluations of AES candidates would be made public.
- The assessments and relative rankings of AES candidates would be made public.
- The rationale for choosing one candidate over the others to be the new standard would be made public.

Twenty-one candidates were submitted by the June 1998 deadline. Six candidates were eliminated for not meeting requirements. Ten more were eliminated during the first round of evaluation.

AES Finalists

The five finalists were announced in 1999:

1. Rijndael, submitted by Joan Daemen of Proton World International and Vincent Rijmen of the Katholieke Universiteit Leuven, in Belgium. NIST ultimately chose Rijndael to become the new AES.
2. MARS, submitted by IBM.
3. RC6, submitted by RSA Laboratories.
4. Serpent, submitted by Ross Anderson of the University of Cambridge, United Kingdom, Eli Biham of Technion, Israel, and Lars Knudsen of the University of California, San Diego.
5. Twofish, submitted by Bruce Schneier, John Kelsey, and Niels Ferguson of Counterpane Internet Security; David Wagner of University of California, Berkeley; Doug Whiting of Hi/Fn; and Chris Hall of Princeton University.

All five finalists were block ciphers designed around the execution of multiple rounds. The final review involved three categories: flexibility, implementation efficiency, and security. There are ways to fairly compare flexibility and efficiency. Each algorithm was assessed as to how easily it could adapt to larger block and key sizes. Although DES was explicitly designed to work efficiently in hardware, the AES candidates were assessed for their efficiency in both hardware and software implementations.

Security assessment posed more of a challenge. It is hard to come up with an objective and appropriate comparison for encryption algorithms. The challenge was to determine

which differences in the algorithms might significantly affect their security. One approach was to assess "reduced round" versions of each algorithm. All finalists implemented rounds, and this assessment sought weaknesses in variants that had fewer rounds than were really required.

For example, Rijndael normally performs 10 rounds when encrypting with a 128-bit key. To search for weaknesses, analysts reduced the number of rounds and looked at the result. With Rijndael, the algorithm did not show weaknesses until at least three rounds were omitted. This was not the widest safety margin in terms of rounds, but was judged to provide a good trade-off between security and efficiency.

All five AES finalists have been judged to be excellent block ciphers. Some products and applications use one or another AES finalist and probably provide good security. However, the only algorithm recognized as a national standard by the U.S. government is AES itself, which is a variant of Rijndael.

9.2.2 The RC4 Story

Ron Rivest developed the RC4 algorithm in the late 1980s. As we saw in Chapter 7.3.2, it is a stream cipher and it has been used in numerous products. It is still used occasionally by older web servers and older wireless networks.

Rivest was a cofounder of RSA Data Security, and the company marketed the algorithm. When introduced in 1987, the company offered RC4 as a *trade secret*. A vendor could pay a license fee to RSA and use the algorithm in a product, but they were forbidden to share the RC4 source code with anyone.

When the United States was founded in the late 1700s, the founders favored the notion of *patent protection* for inventions. A patent requires inventors to publish details of their invention. In exchange, the inventor gets a monopoly on its use for roughly 20 years. This promotes science, education, and knowledge, because the invention's principles are made public. It also produces an Open Design so that others can analyze the invention and verify its behavior. The temporary monopoly gives the inventor some time to commercialize the discovery and pay for the costs of development.

A trade secret, on the other hand, is *never* published. In theory, an inventor could keep a permanent monopoly on an invention by keeping it a trade secret. If anyone who shares the secret publishes it, the inventor can sue them for breach of contract. On the other hand, if someone else makes the same discovery independently, the original inventor has no protection.

Export Restrictions

The trade secret approach proved to be a smart move at the time. Network software vendors realized that their products needed security, and that encryption was the only effective mechanism. However, the U.S. International Traffic in Arms Regulation (ITAR) classified encryption as a weapon of war. Encryption systems, whether in hardware or software, required an export license from the state department. Naturally, the state

department turned to the NSA for advice on such licenses. In practice, the state department only allowed exports for use by U.S. companies overseas.

In the early 1990s, software vendors started negotiating with the NSA to establish some type of exportable encryption. NSA eventually adopted an unwritten rule allowing the use of RC4 and an earlier block cipher, RC2 (Rivest's Cipher 2).

The unwritten rule included two additional restrictions. First, if the product used secret encryption keys, the secret part was limited to 40 bits or less. Second, the algorithms had to remain secret or at least unpublished.

RC4 Leaking, Then Cracking

The RC4 algorithm was leaked to the Internet in 1994. The person responsible was never identified. This produced an odd situation. RSA did not officially acknowledge that the leaked algorithm was RC4, so vendors still could tell the NSA that the algorithm was "secret." Today, however, the crypto community accepts the leaked algorithm as being RC4.

The U.S. government still requires licensing of crypto products for export, but the 40-bit key requirement was effectively dropped in 1999. This was fortunate: By 1997, researchers like Ian Goldberg were cracking 40-bit keys in university computing labs. The revised regulations also permitted other crypto algorithms, just as RC4's weaknesses fully emerged.

Before RC4 was leaked, RSA Data Security claimed that its own analyses confirmed its cryptographic strength. However, cracks appeared soon after the algorithm was leaked. Within a year, researchers found sets of *weak keys*, all of which generated similar key streams.

In an ideal stream cipher, the entropy in the secret key is spread out among the bits in the stream. For example, we might use a 128-bit key to generate a million-bit key stream. Ideally, we should need to see about 8000 bits of the key stream before we start to detect a bias. In RC4, biases appear in the first bytes of the key stream.

In 2001, several researchers published strategies to attack RC4. When RC4 became the centerpiece of wireless encryption, researchers soon developed practical procedures to crack the encryption. Today, product developers avoid RC4.

Lessons Learned

This story illustrates some important points.

- We can't depend on the algorithm's owner to find its flaws. RSA Security claimed to have performed some analyses that provided favorable results. It was not until third parties analyzed RC4 that problems were found. Notably, RSA Security acted responsibly in the face of these results. Although some vendors intentionally have suppressed information about weaknesses in proprietary crypto algorithms and even threatened legal action against published critiques, there is no evidence that RSA Security ever tried to suppress information about RC4's weaknesses.

- It is harder to find flaws if we limit the number of people investigating the algorithm. Cryptology is a relatively new field in the public sphere. We can't rely on any individual expert—or handful of experts—to find the flaws in a particular algorithm. The best way to find flaws is to open the search to as many researchers as possible.
- It is hard to keep an algorithm secret. It is not clear how RC4 leaked. Someone may have transcribed a description published by the owner, or someone may have *reverse engineered* a program containing RC4. In 1994, RC4 was being used in Netscape's Web browsers. Someone might have picked apart the instructions in the browser that encrypted data and derived the RC4 algorithm from that listing.

It took a few years, but the crypto community found and exploited vulnerabilities in this "secret" algorithm. It is hard to keep a crypto algorithm secret, especially if it resides in software that most people have installed on their computer.

9.2.3 Qualities of Good Encryption Algorithms

Since the development of DES in the 1970s, designers have discussed and reviewed numerous cipher designs. Some have stood the test of time in one form or another. Others, like DES and RC4, have been dropped from newer developments because researchers found problems with them. This is the benefit of Open Design; a larger community of analysts may study algorithms, and hopefully identify the weak ones before attackers take advantage of them.

Large, complex products always pose a challenge to buyers, whether they are automobiles or crypto algorithms. In the case of crypto algorithms, however, we can look for certain qualities that greatly reduce our risks. Here are six qualities of good encryption algorithms:

1. Explicitly designed for encryption.
2. Security does not rely on its secrecy.
3. Available for analysis.
4. Subjected to analysis.
5. No practical weaknesses.
6. Implementation has completed a formal cryptographic evaluation.

Several of these qualities reflect the principle of Open Design.

Explicitly Designed for Encryption

In Section 7.3.2, we constructed a key stream algorithm out of a one-way hash (Figure 7.11). In one sense, this seems like a clever thing to do; the one-way hash output seems random. However, this isn't really what the one-way hash is designed to do. A series of hashes might have unexpected statistical properties. Moreover, the hash feedback is itself vulnerable to a known plaintext attack.

We do *not* build a strong encryption algorithm by reusing an existing algorithm for a different purpose. Encryption puts its own requirements on how the algorithm works and what types of patterns are acceptable. Occasionally, we can use an encryption algorithm for a different purpose, like to produce a one-way hash. Even in those cases, we must be careful not to use the algorithm in a way that undercuts its effectiveness.

Security Does Not Rely on Its Secrecy

Open Design, Kerckhoff's law, and Shannon's maxim, all argue aganist keeping a crypto algorithm secret. We build secrecy systems to rely on an independent piece of secret information: the key. If we keep the key secret, then attackers can't efficiently recover the protected information.

If the algorithm's security relies on its secrecy, then everyone who looks at the algorithm will be a member of its cryptonet. This is impractical in countless ways: Not only does it restrict its security analysis, but it makes the algorithm hard to implement and deploy securely.

Available for Analysis

It is hard to assess the strength of a cryptographic algorithm. Some weak algorithms fall quickly to simple attacks, while others like RC4 survive for a period of time before problems arise. In any case, we can't analyze an algorithm unless we can examine it and see what it does. The first step is to make the algorithm available for analysis. If we prevent or delay that analysis, we risk building products with vulnerable encryption.

We can't rely on the algorithm's owners to analyze the algorithm for us. The owners have to pay for the analysis, and they will only pay as much as necessary for *some* good news. They don't need to be thorough and, in fact, it's not clear what a thorough analysis would entail. If we make the algorithm available for public examination, then a broader community of experts have the opportunity to pass judgment. Potential customers who are interested in the algorithm could arrange their own analyses, which are much more likely to be unbiased than those arranged by the owners.

Subjected to Analysis

It's not enough to publish an algorithm. Most experts are busy people and aren't going to pick up arbitrary algorithms and analyze them for pleasure. If experts have analyzed the algorithm and published positive results, that gives us a basis for confidence in its behavior.

If recognized experts have been publishing analyses about an algorithm, we have a better reason to trust that algorithm. These analyses should be in peer-reviewed publications where the analyses themselves are studied by experts. If the consensus of these experts supports the algorithm's use, then we can trust it.

No Practical Weaknesses

Cryptographic experts are going to find weaknesses in just about any cryptographic algorithm. This is a natural result: As we use these algorithms in new ways, they become vulnerable to new attacks.

Ideally, the attacks are not relevant to the way we intend to use the algorithm. For example, there are *chosen plaintext* attacks against DES, but they require enormous amounts of data to be effective. The attack doesn't really apply to typical DES implementations. On the other hand, a theoretical weakness in RC4 was cleverly converted by researchers in Darmstadt, Germany, into a real-time attack on commercial wireless network products.

Cryptographic Evaluation

Our checklist for choosing a file encryption program in Section 7.4.4 noted the importance of cryptographic product evaluation. This remains true when using a block cipher. There are "tricks" to using block ciphers that we will discuss further, and the evaluation helps ensure that the developer uses the right tricks to keep the encryption secure.

Unfortunately, no evaluation process is 100 percent foolproof. In fact, different evaluation regimens focus on different issues, and one process may tolerate security flaws that another might reject. For example, the FIPS-140 process focuses primarily on correct implementation of cryptographic mechanisms. It does not always detect and reject even egregious security flaws, such as the authentication weakness found in several encrypted USB products in late 2009.

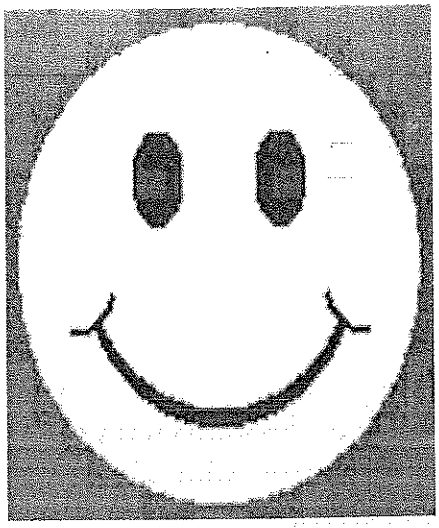
Choosing an Encryption Algorithm

Here are the general recommendations:

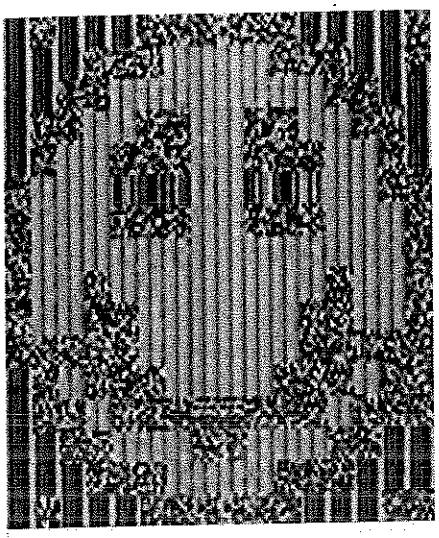
- Use AES if at all possible. AES was subjected to incredible public scrutiny before its selection, which suggests that it will stand the test of time. It is also a U.S. national standard, which provides some legal protection if flaws arise in the algorithm itself.
- Use evaluated cryptographic products if using a certified algorithm. The evaluation process ensures that a trained third party has reviewed the implementation to ensure that the cryptography works correctly.
- Use a well-known, respected algorithm if AES is not available. The best alternatives today are the AES candidates. An older alternative is Triple DES.
- Check recent news and research results in the crypto community regarding the algorithm you want to use. Emerging problems with crypto algorithms have become significant news items in the technical press.
- Do not use "private label" algorithms that have not been published and reviewed by the cryptographic community. There is no way to ensure the effectiveness of such algorithms. History is riddled with examples of weak proprietary algorithms.

9.3 Block Cipher Modes

Block ciphers suffer from a built-in shortcoming. If we feed a block cipher the same plaintext and the same key, it always yields the same ciphertext. This is what block



Smiley face plaintext image



Ciphertext after encrypting the image block-by-block

Courtesy of Dr. Richard Smith

Figure 9.4

Encryption failure using a block cipher.

ciphers do. By itself, this isn't a vulnerability. It becomes a vulnerability when we encrypt data that has block-sized patterns in it.

For example, look at Figure 9.4. On the left, we have a plaintext image, originally rendered in three colors. On the right, we have the ciphertext after it was encrypted with a block cipher. The block cipher transformed the areas of uniform color into distinctive corduroy patterns. Each color in the plaintext yielded its own pattern in the ciphertext. Although this doesn't yield a quality copy of the plaintext image, the ciphertext reveals the essence of the plaintext.

Figure 9.5 shows how this works. We encrypt the text "ABRACARDABRA" with a block cipher that works on 4-byte blocks. Given a particular key, a repeated block of four letters always encrypts to the same ciphertext. In this case, "ABRA" encrypts to the letters "HHIE" every time.

In the early days of encryption, people tried to avoid such patterns by restricting the data being sent. They might use an extra layer of encryption or encourage the radio clerks to rearrange messages to avoid patterns. This is hard to do effectively, because the clerk might change a message's meaning when rearranging its text. In times of stress, when secrecy could be most important, a busy clerk might not rearrange messages at all, and send several in a row with repeated patterns. Cryptanalysts would then focus on messages sent by hurried or careless clerks to crack the encryption.

The most effective way to avoid this problem is to build the solution into the encryption process. The solutions are called block cipher ***modes of operation*** ("modes" for short). Typically, a mode mixes data together from two separate encryptions. This scrambles

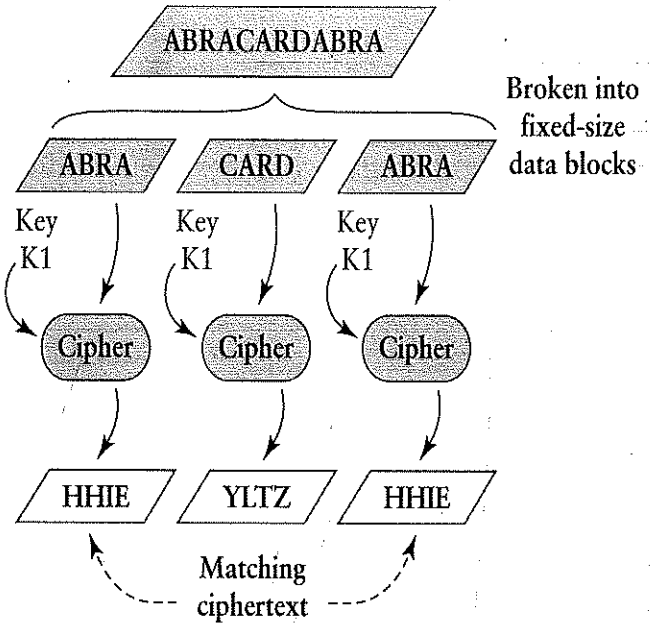


Figure 9.5

Identical blocks encrypt to identical ciphertext.

the result and eliminates patterns in the ciphertext. This all takes place automatically when applying the mode. Figure 9.6 shows the result of applying a mixing mode while encrypting the smiley face.

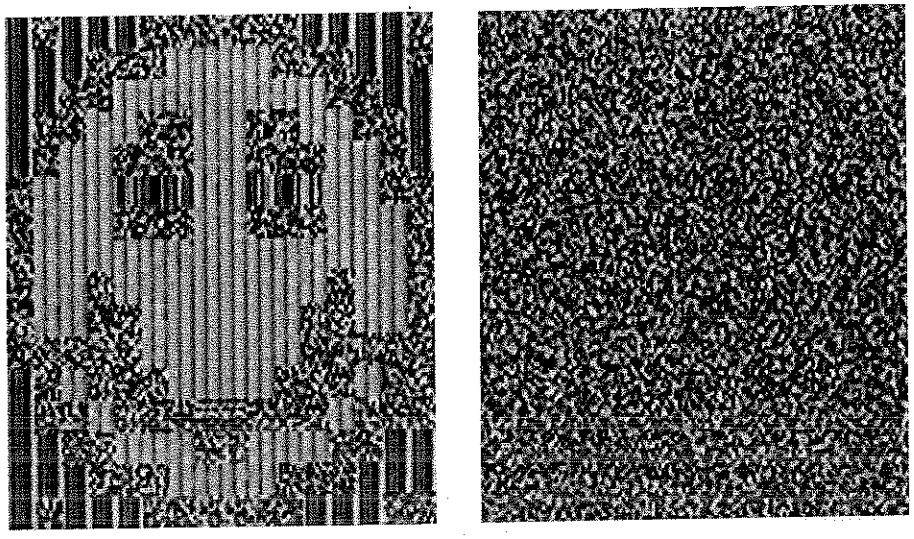
Numerous modes have evolved over the years. Here we examine five common modes. Two of these modes are fairly well behaved and are widely used. Two modes generate a key stream that we can use in a stream cipher.

- **Electronic codebook:** encrypts block by block without mixing
- **Output feedback:** generates a key stream
- **Counter:** generates a key stream; **recommended**
- **Cipher feedback:** a mix of stream and block cipher
- **Cipher block chaining:** encrypts in blocks; **recommended**

These modes are specifically intended to keep information confidential. They do not by themselves protect information from modifications. In some cases, ciphertext errors or other changes only affect the specific bits modified. In other cases, small changes may damage entire blocks of decrypted plaintext.

When we encrypt data block by block, we call that the ***electronic codebook*** (ECB) mode. This term arises from the notion that a block cipher acts like a "codebook" in which every block of plaintext has a unique block of ciphertext associated with it. This is much like traditional codebooks published on paper that list plaintext messages and their corresponding code words.

We rarely use ECB mode because block-sized patterns in the plaintext will appear in the ciphertext. We only use ECB when encrypting data that fits within the cipher's block size.



Encrypted without mixing using the "electronic codebook" mode

Encrypted with mixing using the "cipher block chaining" mode

Courtesy of Dr. Richard Smith

Figure 9.6

Using a mixing mode with a block cipher.

9.3.1 Stream Cipher Modes

Following our discussion in Section 7.3.2, we may use a block cipher to generate a key stream. The simplest approach is the ***output feedback*** (OFB) mode, shown in Figure 9.7.

Note how it strongly resembles the key stream generator in Figure 7.12. We use the block cipher to generate blocks of ciphertext that serve as the key stream. As shown earlier, we encrypt by applying xor to bits of the plaintext and corresponding bits of the key stream. To decrypt, we generate the matching key stream and apply xor to retrieve the plaintext.

Remember from Section 8.2 that we must never encrypt two messages with the same key stream. To avoid this, OFB introduces an extra data item, the ***initialization vector*** or IV. This is a nonce: we want it to be different from one plaintext message to another and we don't want it to be selected by a potential attacker. If we use the same key and IV for two different messages, we generate the same key stream.

When we use IVs, the ciphertext is always larger than the plaintext. Most applications can accommodate the larger ciphertext, though there are a few cases where the plaintext must match the size of the ciphertext. In those cases, we must use a mode that does not require a separate IV.

Ciphertext Errors

In Section 9.2, we explained how ciphertext errors affect block encryption. Errors behave differently when we use a mode. In OFB, the actual encryption and decryption use xor, so errors behave the same as in other stream ciphers. Modified or mistaken bits in the IV

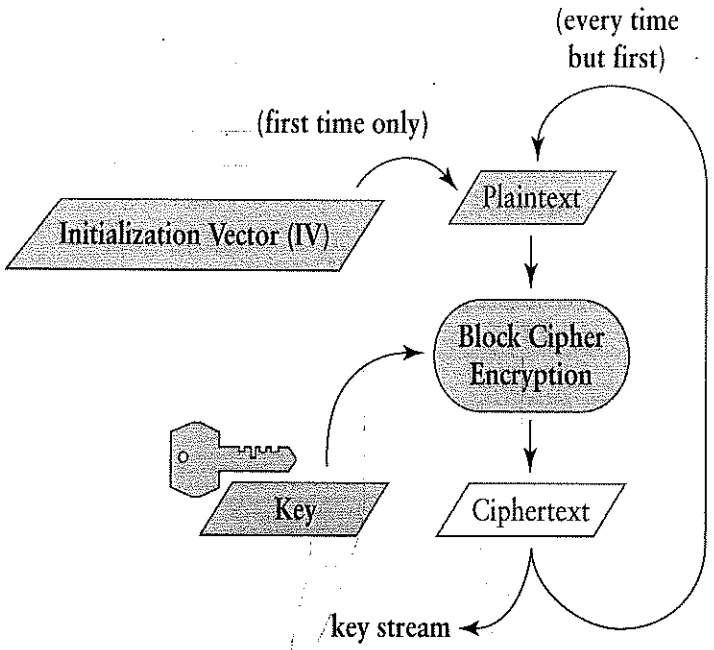


Figure 9.7

Key stream made with OFB (output feedback mode).

will change the key stream and decryption will fail. Modified or mistaken ciphertext bits only affect corresponding plaintext bits. We do not recover the plaintext by running it through the block cipher. We see the same effects if we rearrange ciphertext data; every changed bit will decrypt incorrectly, but the rearrangement affects no other bits.

OFB in Practice

IVs aren't exclusive to OFB; they are used in most modes. When we save the encrypted ciphertext, we must save the IV with it (Figure 9.8). Look at how we generate the key stream; we can't reproduce the same key stream unless we keep the IV. We don't encrypt the IV; we treat it like a nonce and store it in plaintext form with the corresponding ciphertext.

Figure 9.9 shows OFB encrypting a series of data blocks: Plaintext 1, 2, and 3. We encrypt the initialization vector to produce the first block of key stream bits (Key Stream 1). We xor those bits with the first block of plaintext, then we use the block cipher on Key Stream 1 to generate the next block of key stream bits. We next apply xor to encrypt the next block of plaintext.

The process repeats until we have encrypted the entire ciphertext. Because we need the initialization vector to decrypt the ciphertext, we must include it with the ciphertext. To decrypt, we apply the same operations, except that we swap the plaintext with the ciphertext.

Note that Figures 9.7 and 9.9 use different ways to illustrate the same cipher mode. The first figure, Figure 9.7, is a ***key stream diagram***; it shows how the process generates a

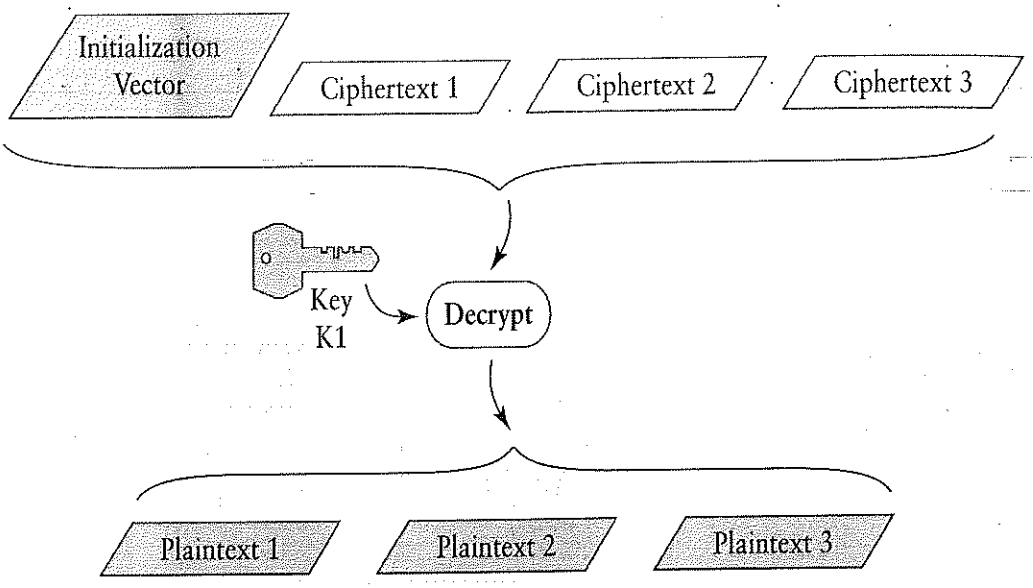


Figure 9.8

Include the IV with the ciphertext when required.

key stream. The second figure, Figure 9.9, is a ***mode encryption diagram***; it shows how successive blocks of plaintext and ciphertext interact to implement the mode. This second approach is used most often to illustrate modes. Not all modes generate a key stream independent of the message's plaintext and ciphertext, so we can't use the first style to portray every mode.

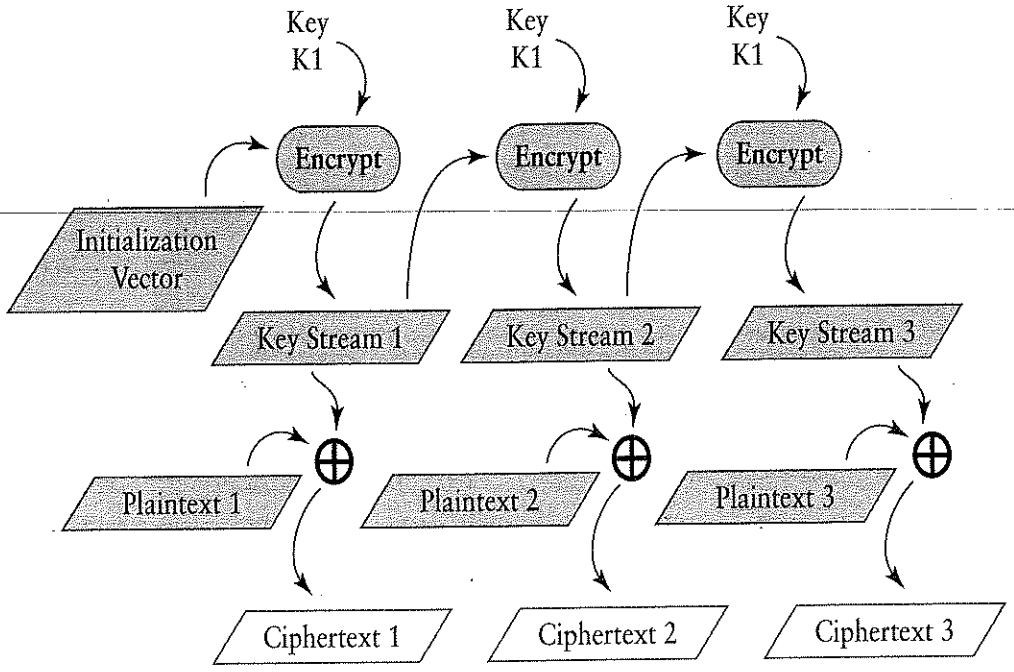


Figure 9.9

Mode encryption diagram: Encrypting with OFB.

Weaknesses

OFB is a very simple mode and we encounter security problems if we choose the IV poorly. If, by some chance, an IV appears as the output of the block cipher, then we produce a duplicated key stream.

For example, imagine that we use the value 123456 as our IV. Later, using the same key, we use a different IV in a second message. By coincidence, the second message's key stream generates 123456 as one of its values. Look at what happens when we feed it back to the cipher: From that point on in the message, we use the same key stream as was used in the first message. Fortunately, this is unlikely to happen if the cipher's block size is very large and we often change encryption keys.

Counter Mode

Counter (CTR) mode, like OFB, generates a key stream and applies xor to implement a stream cipher (Figure 9.10). Instead of using feedback, the counter mode encrypts successive values of a counter. This eliminates the OFB risk of an accidentally repeated IV and key stream. To avoid this, we *never* encrypt the same counter value with the same key. The initialization vector tells us where to start the counter.

In practice, we can construct the counter in either of two ways. First, we can simply keep a running counter for every block we encrypt with a particular encryption key. We could start at a random value like 12345 and increment the value for each block we encrypt. If we encrypt 100 blocks, the counter will be 12444 when we encrypt the final block. We then use 12445 when we encrypt the 101st block.

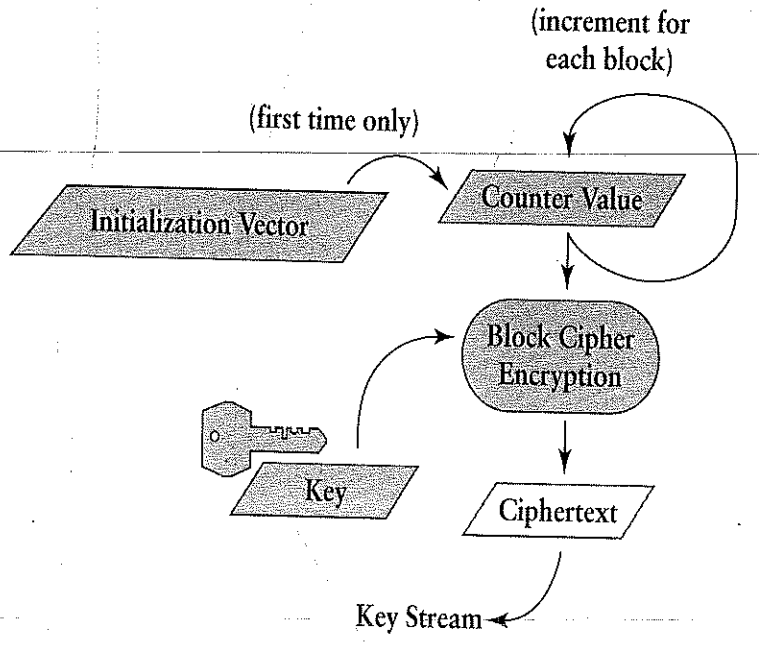


Figure 9.10

Key stream with CTR—the counter mode.

For a different approach, we form the counter in two parts. A nonrepeating nonce provides the upper part. We increment the lower part. We may use either approach as long as we can guarantee not to reuse a counter value before we change to a different encryption key.

Because CTR is essentially a key stream generator that encrypts like a conventional stream cipher, it has the same ciphertext error properties as the OFB mode.

9.3.2 Cipher Feedback Mode

The *cipher feedback* (CFB) mode, shown in Figure 9.11, combines properties of stream ciphers and block ciphers. We generate a key stream one block at a time and combine it with the text using xor. Instead of feeding back the block cipher's output, CFB feeds back the actual ciphertext block. CFB doesn't face the same risk of repeated IVs that we have with OFB. Because each block of encryption depends on both the key stream and the message data, OFB won't repeat the key stream unless we reencrypt exactly the same data with exactly the same IV.

Although we encrypt complete blocks of data to generate the key stream, the plaintext does not have to fit into an integral number of blocks. When we reach the final block, we may encrypt the remaining plaintext and discard any extra key stream bits.

Ciphertext Errors

Because CFB feeds the ciphertext through the block cipher to decrypt it, ciphertext errors affect it differently. Remember what happens if we change a *single bit* of a block cipher's input: It affects the entire output block.

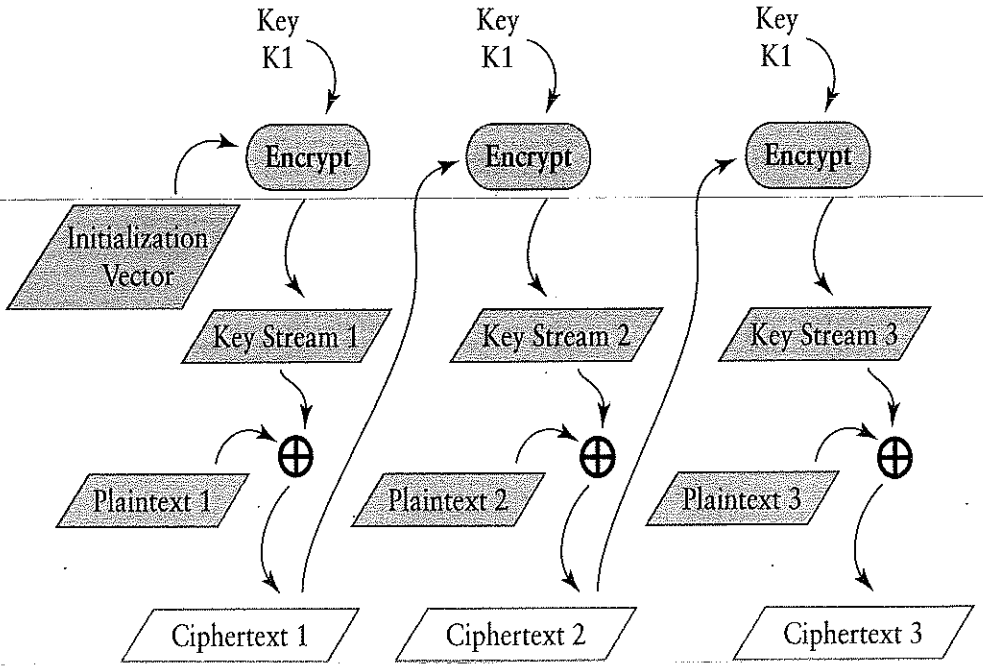


Figure 9.11

Mode encryption diagram for CFB (cipher feedback mode).

Note how CFB ciphertext affects the decrypted plaintext: Each block gets used twice. First, we xor the ciphertext with the key stream. Next, we use the ciphertext as input to the block cipher to produce the next block of key stream.

A single bit error produces two separate changes to the plaintext. First, the corresponding plaintext bit is inverted (as in a stream cipher error). Second, the next block of plaintext is completely scrambled. Thus, we see ***error propagation*** because this single error in the ciphertext damages additional plaintext when we decrypt. The error propagation only affects the following block.

If we rearrange ciphertext blocks encrypted with CFB, we get different results than with a straight block cipher or with a stream cipher. As noted, a misplaced block decrypts incorrectly and causes the block following it to decrypt incorrectly. However, if all subsequent blocks appear in the right order, they will all decrypt correctly. Because of this, CFB is called a "self-synchronizing" cipher; there is a limit to error propagation during decryption, after which the ciphertext decrypts into the correct plaintext. For example, let's switch ciphertext blocks 2 and 3 before decryption and see what happens:

- We retrieve plaintext block 1 correctly from the IV and from ciphertext block 1.
- We generate key stream 2 correctly from ciphertext block 1.
- We incorrectly generate plaintext block 2. We xor key stream 2 with the misplaced ciphertext block 3. This yields nonsense.
- We incorrectly generate key stream 3. We decrypt the misplaced ciphertext block 3, which actually yields key stream 4. We use it in the place of key stream 3.
- We incorrectly generate plaintext block 3. We xor the misplaced key stream 4 with misplaced ciphertext block 2. This yields nonsense.
- We incorrectly generate key stream 4. We decrypt the misplaced ciphertext block 2, which actually yields key stream 3. We use it in the place of key stream 4.
- We incorrectly generate plaintext block 4. We xor the misplaced key stream 3 with the properly placed ciphertext block 4.
- Output returns to normal with plaintext block 5, because the undamaged ciphertext block 4 generates the correct key stream.

Thus, if we swap two adjacent ciphertext blocks, we see errors in three plaintext blocks when we decrypt. The same is true if we change one or more bits in two adjacent ciphertext blocks. The decryption resynchronizes and yields correct plaintext one block past the last ciphertext error.

9.3.3 Cipher Block Chaining

Last, and certainly not least, is the ***cipher block chaining*** (CBC) mode (Figure 9.12). CBC is one of the oldest and most widely used cipher modes. Unlike the other modes examined, this one is block oriented. To use this mode, the plaintext must be a multiple of the cipher's block size. If it is shorter, we must add padding.

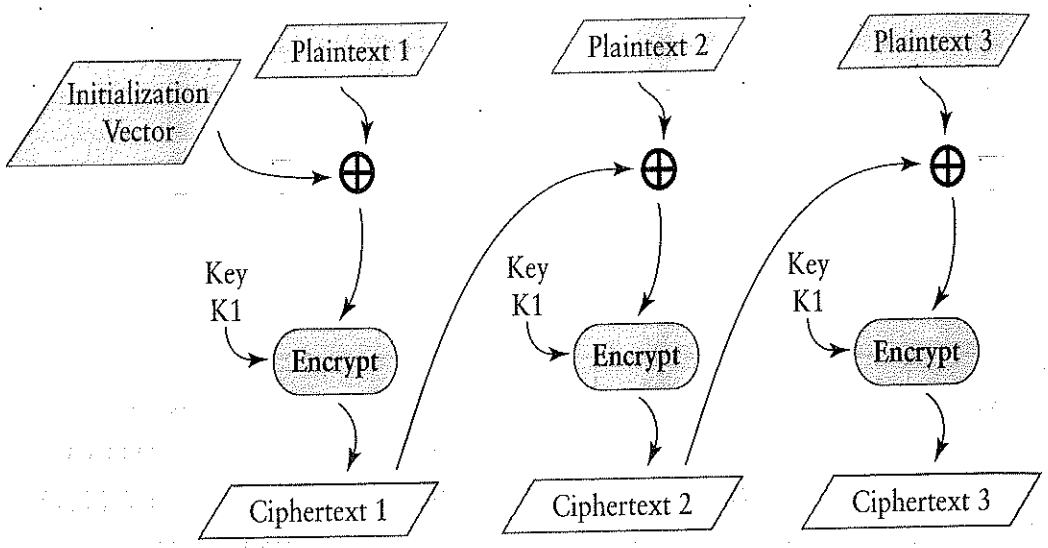


Figure 9.12

Mode encryption diagram for CBC (cipher block chaining).

CBC encryption involves two steps. First, we scramble the plaintext by combining each plaintext block with the previous ciphertext block. As shown in the figure, we use xor to combine the plaintext and ciphertext. We combine the first plaintext block, which has no previous ciphertext, with the IV. The second step is block encryption. Its output provides the ciphertext block. We also use this ciphertext block when we encrypt the next plaintext block.

Figure 9.13 illustrates CBC decryption using a ***mode decryption diagram***. We start by decrypting the first ciphertext block. Unlike other modes, we actually use the block

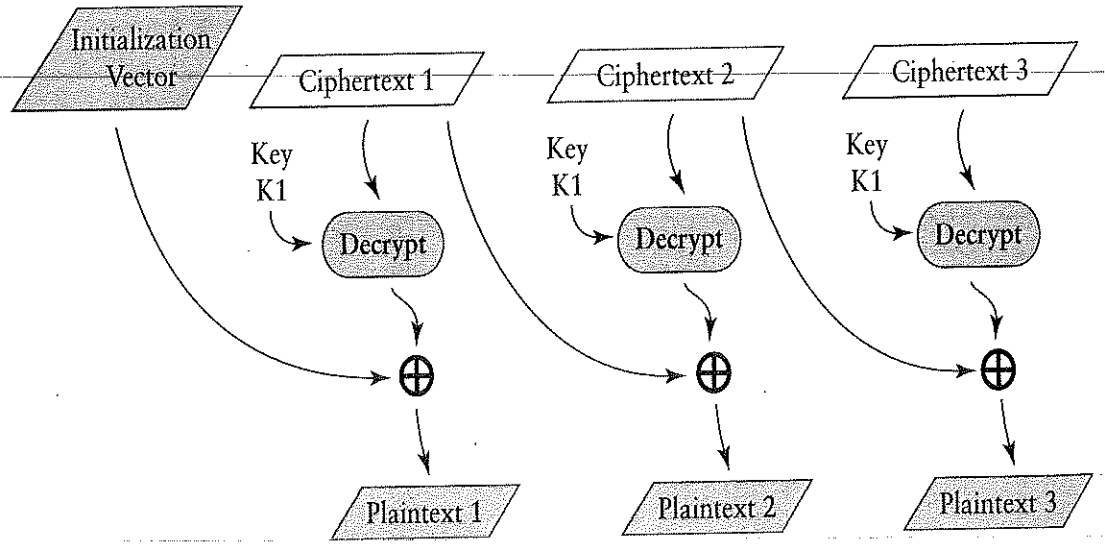


Figure 9.13

Mode decryption diagram for CBC.

cipher's decryption procedure to decrypt the ciphertext. We combine the result with the IV to recover the first plaintext block. For subsequent blocks, we first decrypt the ciphertext block, and then we xor the result with the previous ciphertext block.

Like CFB mode, we use each ciphertext block twice when decrypting. Unlike CFB, both the plaintext and the ciphertext must fit into an integral number of blocks.

Ciphertext Errors

CBC has similar error propagation and self-synchronizing properties to CFB. A single bit error in a ciphertext block will affect both the current block and the next block. Subsequent error-free blocks will decrypt correctly. If we rearrange ciphertext blocks, we lose the corresponding plaintext block contents, plus one block following the ones that moved.

9.4 Encrypting a Volume

The risk and policy discussions in Section 9.1 argue strongly for encrypting storage volumes of all shapes and sizes, but volume encryption also poses some challenges. If a cryptanalyst steals an encrypted drive, there will be gigabytes of ciphertext with which to work. The attacker will be able to guess some of the corresponding plaintext because disk formats are public knowledge. In addition, the attacker can be certain that a great deal of duplicate data resides on the drive, simply because that's what happens on a modern hard drive. If the encrypted drive is a system drive, the attacker might be able to make astute guesses about the drive's contents and make undetectable changes to it.

These risks lead to the following objectives for FDE implementations:

- **Strong encryption:** Use a strong, modern block cipher that can reliably encrypt trillions of bytes of data and more.
- **Large encryption key:** Strong modern ciphers require large, random encryption keys. The ciphers don't provide their strongest security except with fully random keys.
- **High speed:** In practice, this means that we want to encrypt all of the drive's data with the same key. If we switch between keys while encrypting the drive, then we'll have delays awaiting key expansion.
- **Suppress any data patterns:** Use a block cipher mode that mixes data during encryption so that plaintext patterns aren't duplicated in the ciphertext. Moreover, it should be impractical for an attacker to choose plaintext data that creates a recognizable pattern in the stored ciphertext.
- **Plaintext size = ciphertext size:** Modern file systems assume they have full access to the disk. We can't steal space from the file system to store IVs, for example.
- **Integrity protection:** This is not the primary concern, but at least it should be hard to make undetectable changes.

In addition, the FDE implementation must handle the crypto keys safely and efficiently. If we have permission to use the encrypted drive, we should be able to supply the keys and mount the drive. When we dismount the drive, the keys should disappear. In hardware

implementations, this is tied to the drive hardware status: The drive is "locked" when it is powered off or reset and "unlocked" when its keys are in place and it is mounted for use. We discuss this further as we examine key management and drive hardware issues.

Choosing a Cipher Mode

In order to avoid patterns in the drive's ciphertext, we need to encrypt data differently according to *where* it resides on the hard drive. Conventional cipher modes eliminate patterns by using IVs and chaining. We don't really have a place to store IVs, so we incorporate location information into the encryption instead.

One solution is called a ***tweakable cipher***. A normal cipher takes two inputs: the key and the plaintext, and yields the ciphertext. A tweakable cipher has a third input, the *tweak*, a nonce-like value that modifies the encryption without the cost of changing the encryption key.

In drive encryption, the tweak identifies the disk sector and selects a block within the sector. In practice, we don't really have to design a new block cipher. Instead, we use a block cipher mode to tweak the cipher for us.

Here are two general approaches:

1. Adapt a classic cipher mode. In Section 9.4.2, we adapt CTR and CBC modes to drive encryption.
2. Develop a special, tweakable mode for drive encryption. In Section 9.4.3, we look at the XTS mode, which eliminates shortcomings of CTR and CBC.

Hardware Versus Software

Modern FDE techniques are based in either hardware or software. Hardware-based systems may be built into the hard drive's controller or they may exist as add-on circuits. Software-based systems often operate as a device driver. Both hardware- and software-based techniques share these four properties:

1. Both appear "below" the file system in terms of software layering (Figure 5.18), either at or below the device driver.
2. Both are typically unlocked with a typed passphrase.
3. Both often store their working keys as wrapped keys and use a passphrase to produce the KEK.
4. Both can encrypt the system volume. In both cases, there needs to be special software, or BIOS firmware, that collects the passphrase to unlock the drive.

We discuss hardware-based FDE in Section 9.5. We discuss software-based FDE next.

9.4.1 Volume Encryption in Software

When we look beyond the similarities between hardware and software-based FDE, we find important differences. Because software products reside within the operating

system, they have both additional features and additional weaknesses. Here is a listing of better-known FDE software implementations:

- Apple's Macintosh OS-X—can create encrypted drives and "virtual disks" through its Disk Utility.
- Microsoft BitLocker—an optional feature of certain higher-end versions of Microsoft Windows.
- PGPDisk—a commercial package from the company that sells PGP email encryption software. The program uses the same public-key certificates to wrap disk keys that it uses to wrap email encryption keys.

Figure 9.14 illustrates how the software FDE product fits into an input/output operation. This is a variant of Figure 5.18. In this example, we store the actual data on a hard drive partition. The file system treats the FDE package like another type of hardware. When the file system writes data to the FDE, the FDE software encrypts the data, then it passes the ciphertext to the actual hard disk driver. When the file system reads data, it sends the request to the FDE driver. The driver retrieves the corresponding cluster from the hard drive, decrypts it, and passes it back to the file system.

In addition to the driver, the FDE software includes other utility functions. There is typically an application to mount and dismount encrypted drives. This software requests the passphrase and then uses it to unwrap the drive's encryption keys (Figure 9.15). If the FDE software can encrypt the system volume, there is also an installation program and a

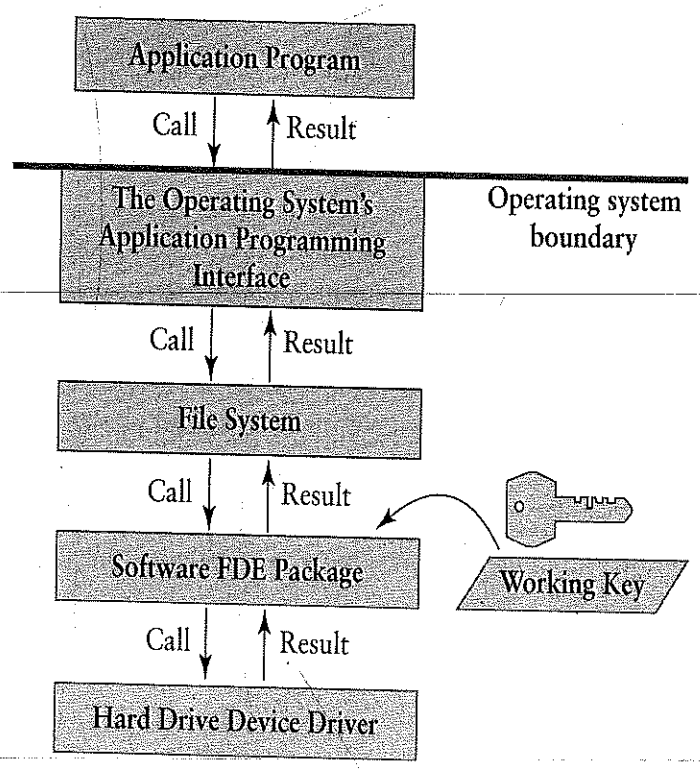
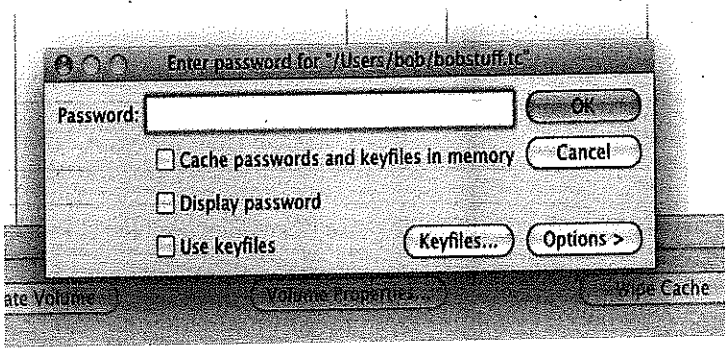


Figure 9.14

Full disk encryption in software.



Screen shot reprinted with permission from Apple Inc.

Figure 9.15

Password prompt to mount an encrypted volume.

loader for bootstrapping the encrypted system volume. In better implementations, the installation program encrypts the system volume while the system is running, and then updates the bootstrap sectors to boot from the encrypted system volume.

Files as Encrypted Volumes

A modern file system treats a physical device—or a partition on that device—as a collection of numbered clusters. The inside of a file is also a collection of numbered clusters. If we can make access to the file look like access to a partition, then we can store a file system *inside* a file.

We then can encrypt the file system inside the file using FDE. We call this a ***file-hosted volume*** as compared to a ***device-hosted volume***. If we store the volume on an entire hard drive or a drive partition, it is device hosted. If we store the volume on a file within our file system, then it is file hosted.

Many FDE packages, including PGPDisk and OS-X, support both device-hosted and file-hosted volumes. Such packages have a utility program that opens the file, and the encrypting driver redirects disk accesses to the file. This is called a ***loop device***; the system must "loop" through the file system twice to read a file stored on a file-hosted volume.

Bob has created a file-hosted PGPDisk volume. He started up the "mount" utility, which prompts him for a password, as shown in Figure 9.15. Once Bob types the password, PGPDisk opens the file, attaches it to its encrypting driver, and directs all accesses to the file. Encrypted drives often are treated as removable drives and formatted with FAT, but most systems allow the owner to apply a different file format.

9.4.2 Block Modes for Volume Encryption

Volume encryption requires a tweakable cipher mode. When we encrypt or decrypt a block of data, the transformation must depend on a volume-wide secret key and a block-specific tweak value. We use the same key to encrypt sectors 12345 and 12346, but we must tweak each block's encryption to produce different ciphertexts. The tweaks

appear as IVs or counters in typical cipher modes. For drive encryption, we must derive the IV or counter from the data's sector address on the drive.

Drive Encryption with Counter Mode

Counter mode produces a key stream by encrypting successive values of a counter. We then encrypt or decrypt by using xor with the key stream. To avoid ciphertext patterns, we must produce a unique counter value for every 128 bits of data on the hard drive. We assume each sector contains 512 bytes, so we have 32 separately encrypted blocks of data in every sector.

This yields a tweakable cipher mode. If we encrypt the same data to store on the same 128-bit block in a particular sector of the drive, we always produce the same ciphertext. This makes writing much more efficient. To achieve this, the mode establishes a separate tweak value for every 128-bit block of encrypted data on the drive. In Counter mode, the tweak value is the counter.

Constructing the Counter

The counter value must be unique for every 128-bit block on the device. To do this, the counter incorporates the sector number and the block's index number within the sector. We construct the counter in three parts:

1. The counter's low-order digits select a single block within the sector. There are 32 blocks of 128 bits each in a standard sector containing 512 bytes. We assign the five lowest bits in the counter to select a block within the sector.
2. The middle digits in the counter select the sector on the hard drive.
3. The remaining high-order digits in the counter will contain a nonce that remains constant for all sectors on the hard drive.

Let's plan ahead: Assume that the arrangement needs to work with hard drives that contain up to an exabyte of storage.

- 1 exabyte = 1 quintillion bytes = 10^{18}
- 1 exabyte's approximate size in binary = 2^{60}
- Number of 512-byte sectors = $2^{60}/512 = 2^{60}/2^9 = 2^{51}$

To produce unique counter values for every sector on a 1-exabyte hard drive, we need a 51-bit counter. To uniquely count each AES block on a sector requires another 5 bits. We use a nonce to set the other 72 bits in the 128-bit counter value. The counter value yields the 128-bit block that we encrypt (Figure 9.16).

We never actually increment a counter value when we encrypt a sector on the hard drive. Instead, we calculate the counter value based on the sector's numerical address and the index of the block within the sector.

We insert the nonce in the high-order bits in the counter. We encrypt this assembly of bits to get the 128 bits of key stream. We xor the key stream to encrypt or decrypt the corresponding data.

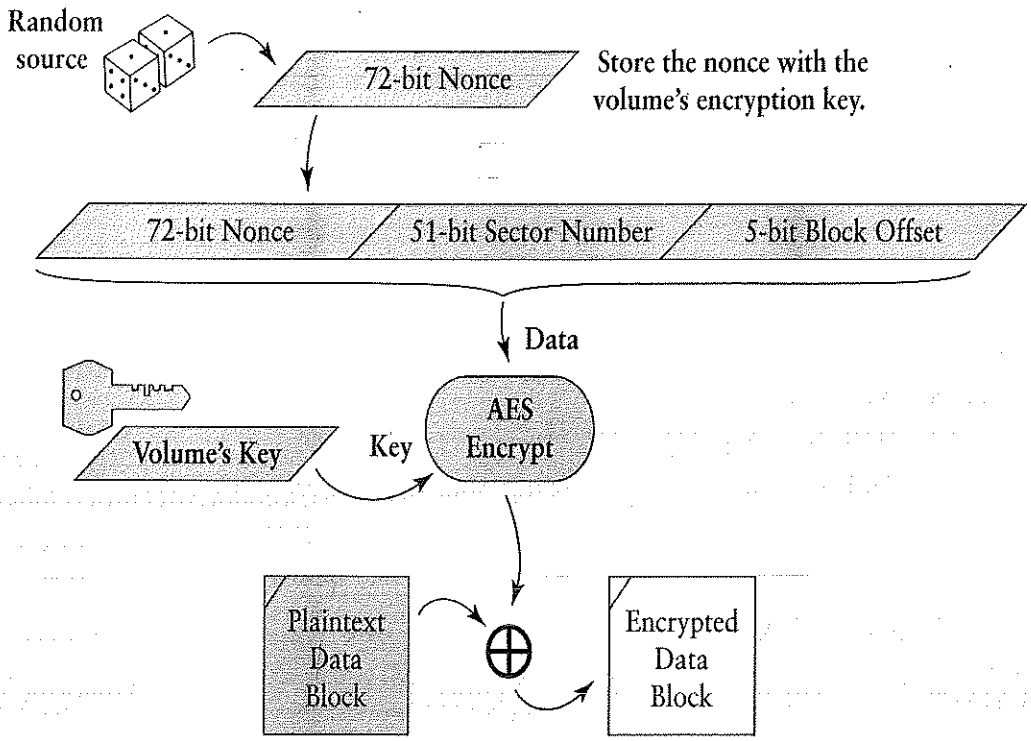


Figure 9.16

Encrypting disk data with AES and Counter mode.

An Integrity Risk

A major problem with Counter mode is that it relies on a simple xor operation. An attacker can use a bit-flipping attack to replace known data on the drive.

For example, Kevin knows that a certain computer system stores the bootable operating system in a particular place. Because it's easiest to administer systems if they're all alike, Kevin knows the *exact* data contents of the bootable operating system. Thus, he can extract the encrypted data and use a plaintext copy of the system to isolate the key stream. Then he simply reencrypts a modified version of the operating system using that same key stream and returns the system to its owner. When the owner reboots, the computer decrypts and runs the modified operating system.

In practice, hard drive encryption can't protect 100 percent against integrity attacks. However, this sort of attack is so simple to implement that it is worth preventing.

Drive Encryption with CBC Mode

CBC does not produce a tweakable mode. We can't encrypt or decrypt individual blocks on a sector with CBC as we can with a tweakable mode. Moreover, the same plaintext will yield different ciphertext even when saved in the same place on the drive, because the ciphertext depends on adjacent ("chained") data during encryption.

However, CBC provides a well-understood cryptographic mechanism. We encrypt a sector at a time and use the sector number as the IV. A prototype hard drive encryptor

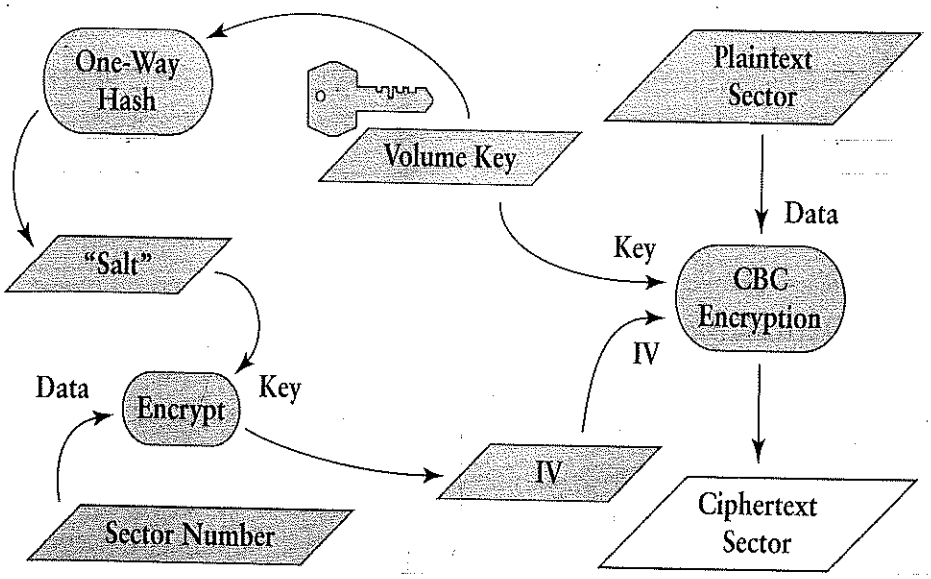


Figure 9.17

Sector encryption with CBC and ESSIV.

built for the U.S. government in the early 1990s used this approach with a variant of the CFB mode.

Our first challenge is to ensure unique IVs across the entire volume. One encryption product cut corners and used only 16 bits of the sector number to distinguish the IV. This produced identical IVs for disk sectors that were exactly 65,536 sectors apart.

However, trouble emerges even if we use the entire, unique block number for the IV. The problem is subtle and, in some applications, relatively minor: An attacker can "leak" information through the ciphertext. Specifically, an attacker can construct data that combines with the IV during encryption in a way that stores predictable ciphertext values on different sectors of the drive. Although this might not lead to immediate trouble in many environments, it indicates an unnecessary weakness. We eliminate it by making the IV hard to predict.

Linux-based drive encryption software developed a technique to address this. The solution encrypts the sector number with "salt" to construct the IV, yielding the acronym "ESSIV." In this case, the "salt" is a hashed copy of the volume key (Figure 9.17). In the diagram, CBC encryption has three inputs: the data, the key, and the IV.

Integrity Issues with CBC Encryption

Section 9.3.3 described the error propagation properties of CBC in general terms. The matter becomes more interesting if we apply them to rearranging data on disk sectors.

When CBC encounters an error while decrypting, the error affects the encrypted block containing the error and the following block. After that, decryption continues without error. If we rearrange blocks on a disk sector, there will be garbage in the blocks at the point of rearrangement and the rest will decrypt normally.

This does not pose as large a risk as the bit-flipping attack on CTR mode. An attacker can modify CTR-encrypted data cleanly without leaving telltale blocks of garbage. Moreover, hard drive behavior makes it impractical to prevent all possible integrity attacks.

9.4.3 A "Tweakable" Encryption Mode

The XTS cipher mode evolved from work on tweakable cipher modes over several years. The acronym actually is derived from three other acronyms. This mode was based on an earlier mode called "Xor-Encrypt-Xor" implemented as a "Tweaked Codebook Mode" with "Ciphertext Stealing."

Although XTS may work in a broad range of applications, we focus here on disk encryption using the AES algorithm. XTS handles its tweak input in two parts: One part selects a large chunk of data, like a sector, and the other part selects a single block for encryption by the block cipher. In our case, blocks within a sector are 128 bits each to work with AES. Figure 9.18 shows an encryption operation for a selected 128-bit block within a numbered disk sector.

There are essentially two parts to the XTS mode: generating the tweak and doing the encryption or decryption. We use two keys, one for each part. We generate the tweak the same way for encryption and decryption, as shown in the upper left of the diagram. We

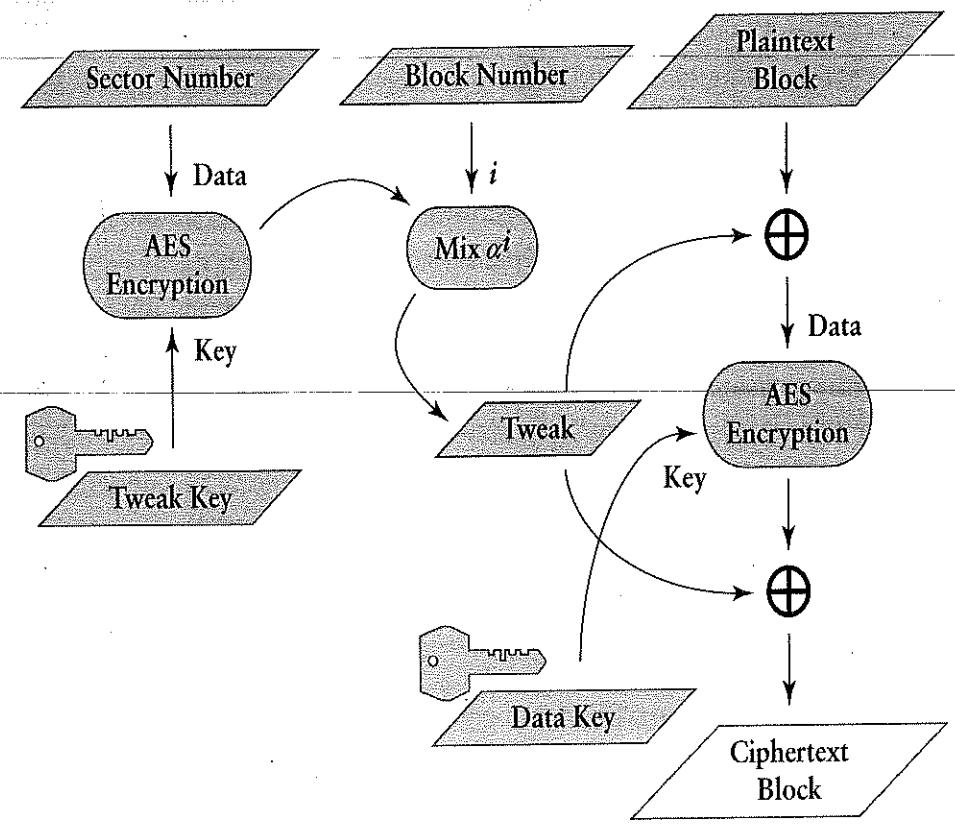


Figure 9.18

XTS mode to encrypt data on a disk sector.

encrypt the sector number and then we combine it with the block number to construct the tweak. To perform the encryption, we xor the tweak with the plaintext, encrypt it, and xor the tweak with the ciphertext. We reverse that process to decrypt.

The "mix" operation shown in the figure is a modular multiplication of the encrypted sector number by α^i, where i is the block within the sector. The value α was chosen for its mathematical properties, so that the resulting mix is effective but only requires a small number of adds, shifts, and loops.

The phrase "Ciphertext Stealing" refers to the strategy for handling partial AES data blocks. If individual sectors do not end on a 128-bit boundary, then the mode takes special steps to encrypt or decrypt the partial block at the end of the sector. Those steps combine ciphertext from the previous block with the partial block. Here we only look at the case where we encrypt or decrypt full AES blocks. Partial blocks are rarely a problem because sectors are 512 bytes or a multiple of that size.

IEEE Standard 1619-2007 specifies the details of the XTS mode, and NIST published a recommendation for its use (NIST SP 800-37E). There is no requirement to use the XTS mode as opposed to the others, but it poses a stronger challenge to attackers. Although it is hard to rearrange blocks successfully in CTR mode, attackers can easily mount a bit-flipping attack on it. Even though it isn't practical to attack CBC mode with bit-flipping, some rearranged blocks on a sector will decrypt correctly. XTS mode resists both bit-flipping and block rearrangement.

9.4.4 Residual Risks

Volume encryption can provide strong protection but, like all security measures, includes residual risks. For example:

- Untrustworthy encryption
- Encryption integrity
- Leaking the plaintext
- Data integrity

We address the first one by picking a quality volume encryption product. We can address the second by using hardware-based encryption (Section 9.5); otherwise, we depend on the security of our operating system. Regardless of the encryption implementation, the third risk is tied to the security of our operating system. The fourth requires additional security measures.

Additional risks arise as we examine key management issues. Those are addressed in Section 9.6.

Untrustworthy Encryption

This problem arises if the encryption implementation is faulty. The encryption might use a weak algorithm or apply the algorithm in a way that weakens its protection.

Early commercial volume encryption often used DES encryption. These drives appeared around the turn of the millennium. Ironically, this was shortly after researchers

demonstrated the DES Cracker. Although a few volume encryptors used Triple DES, those that relied on 56-bit keys provided relatively weak encryption.

The evolution of AES produced a whole suite of strong block ciphers: the AES finalists and AES itself. These ciphers appear in many modern products. Even though these ciphers are strong, they require an appropriate encryption mode to suppress patterns in the ciphertext.

We reduce the risk of untrustworthy encryption by using certified products. In the United States, the recognized certification is under the latest FIPS 140 standard (FIPS 140-2 is the version as of 2011). Government agencies and most financial institutions are required to use FIPS 140 certified products. Many other countries recognize FIPS 140 certification, and some have implemented their own version of the certification. Note, however, that FIPS 140 only applies to U.S. government standard algorithms, including AES, and not to the AES finalists.

Encryption Integrity

Even if the product implements a quality algorithm with an appropriate mode, an attacker might be able to weaken, disable, or bypass the encryption. Thus, the stored data is either weakly encrypted or not encrypted at all.

This is a particular risk for software-based encryption. We must install the software correctly and then rely on the operating system to protect its integrity.

Hardware-based products, described in Section 9.5, may be less vulnerable to these types of attacks. To disable or bypass the encryption, attackers would need to attack the driver controller circuitry. This poses a serious challenge for most attackers. Integrity risks may arise if the hardware relies on firmware and the firmware may be updated.

Looking for Plaintext

Robert H. Morris (1932–2011), an NSA chief scientist, liked to repeat a popular saying among his colleagues: "Rule 1 for cryptanalysis: Check for plaintext." In other words, the smart attacker doesn't start by trying to crack the encryption. The attacker first looks for an opportunity to retrieve the data *before* encryption or *after* decryption. File encryption yields a number of opportunities to look for plaintext if the attacker shares a computer with the target. This is true especially when using full-disk encryption.

During 2013, the news sites Der Spiegel and the Guardian reported on and released NSA documents describing several highly sensitive "Special Intelligence" programs (see Section 17.2.4). These were part of a cache of documents released to news organizations by Edward Snowden. One program, called BULLRUN, focused on defeating encryption in commercial Internet devices. These attacks often relied on subverting the "endpoint" computers that originally encrypted or finally decrypted the data. Such an attack did not have to crack strong encryption; it simply retrieved the data while in plaintext form. The attacks exploited weaknesses in computer operating systems to install back doors to retrieve the plaintext.

When we use full-disk encryption, everyone on the computer sees the drive's contents as plaintext. This is a case of Transitive Trust: When we rely exclusively on full-disk encryption, we implicitly trust everyone who shares the computer.

Data Integrity

We examined three different modes for encrypting data at the volume level. By including tweaks in volume encryption, we make it impractical to rearrange sectors on the volume. Even though we use the same key to encrypt and decrypt all sectors, the tweaks make the encryption specific to individual sectors.

Within an encrypted sector, the different modes face different integrity problems. CTR mode suffers the worst because we can apply the bit-flipping attack to it. If we know the plaintext contents of a sector, we can change those contents to anything else by flipping the appropriate ciphertext bits. In CBC mode, we can rearrange blocks in a sector, and most blocks will decrypt correctly.

Even though XTS mode doesn't allow us to arrange data within a block, it shares an integrity problem with the other two modes. An attacker always can rewrite the data on a sector with an *older* copy of that sector.

For example, we might make a complete copy of an encrypted volume. Later, we examine the volume to see which sectors have changed. We can rewrite any of those changed sectors with their earlier contents. The earlier contents will decrypt correctly, because they were encrypted with the same key and tweak.

This attack in which we rewrite a sector with older data will work successfully against all three disk encryption modes. It poses a real security challenge. Fortunately, this is not a major risk and not the risk for which we encrypt our drives.

9.5 Encryption in Hardware

If we want to protect our files with encryption and we want the encryption built in, it makes sense to incorporate it into the drive hardware. We call such a device a ***self-encrypting drive*** (Figure 9.19).

We want the encrypted drive to work with our computer exactly as if it were a normal, nonencrypted drive, with one exception: We want our data protected if the drive is unattended or stolen. In practice, we do this by erasing the encryption key when the drive is unplugged or reset. This "locks" the data on the drive. When we plug in the drive or power it on, we must provide the encryption key to "unlock" the drive.

The locking mechanism is tied to the fundamental problem of all crypto security: How do we manage the keys? If we have a relatively bulky drive, we could store the key on a removable flash memory, either on a card or plugged into a separate USB port. If we encrypt a smaller device, like a USB flash drive, then we need a different strategy for handling the keys. We examine different key management approaches in Section 9.6.

Recycling the Drive

Drive encryption makes it very easy to recycle the drive. The owner directs the controller to purge itself of every copy of the current encryption key and of the associated nonces and IVs. Once finished, there is no practical way to decrypt any data on the hard drive.

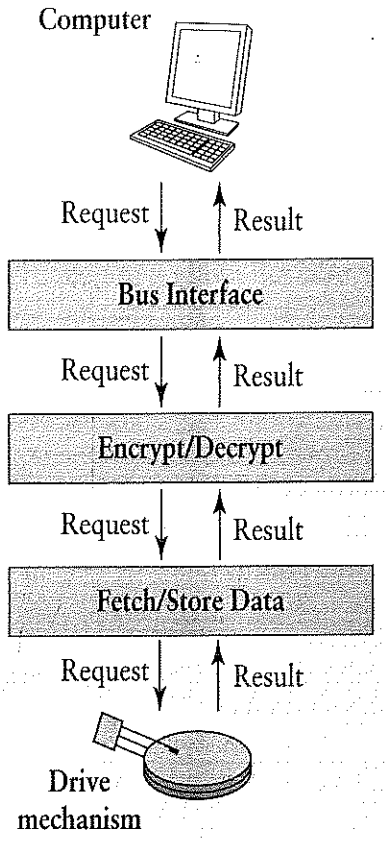


Figure 9.19

Internal functions of a self-encrypting drive.

The drive may still be used, but the owner must produce a new encryption key and reformat the drive's file system.

9.5.1 The Drive Controller

To add encryption to the hard drive, we make no changes to the physical hard drive mechanism. All changes are made to the hard drive controller. Section 5.3.1 described the operation of a typical hard drive controller circuit. A self-encrypting drive shares many of the same features and introduces three features:

1. Algorithms to encrypt and decrypt the stored data, described earlier
2. Mechanisms to manage "locking" and "unlocking" the drive, described in Section 9.5.2
3. Mechanisms to handle the encryption key, described in Section 9.6

Figure 9.20 shows the hardware block diagram of a self-encrypting drive controller. This is a revision of Figure 5.4, which shows the controller for a nonencrypting drive. The connections between the different components pass commands (marked "C"), data (marked "D"), or keys (marked "KEK" or "CEK"). Drive locking and unlocking are

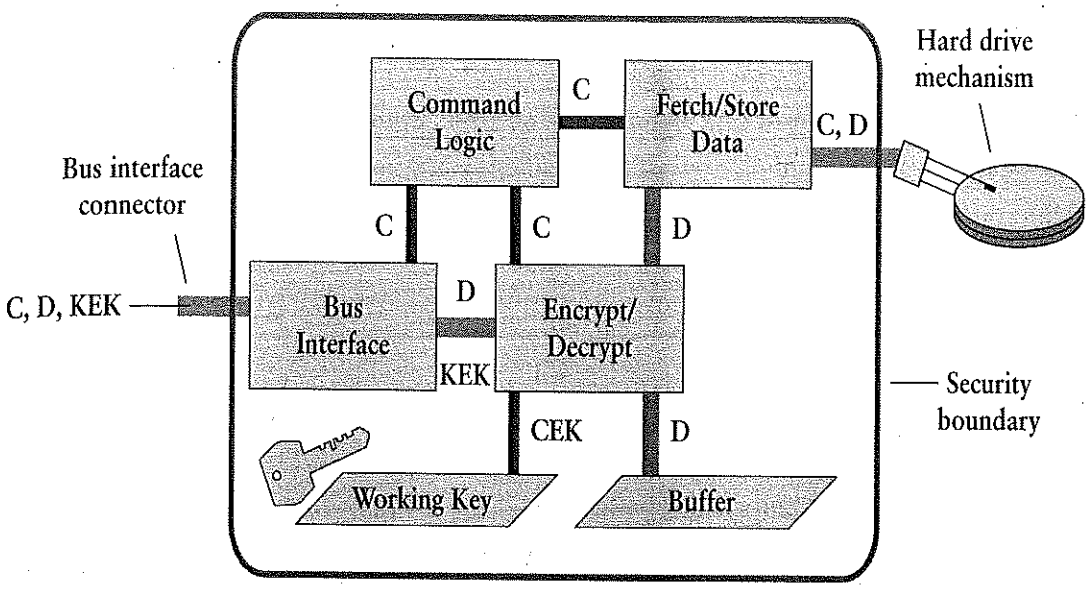


Figure 9.20

Block diagram of a self-encrypting drive controller.

often implemented through key wrapping. We use the key encrypting key (KEK) to decrypt the content encryption key (CEK).

Security Boundary

The security boundary indicates the area within which the drive performs its security activities. As long as attackers don't disturb anything inside that boundary, the controller protects the data on the drive.

Actions outside the boundary should not affect the security of the drive. The drive should reject invalid commands or requests to decrypt data when the drive is locked. If the attacker tries to bypass the controller and read data directly from the drive mechanism, the attacker will only retrieve unreadable, encrypted data. The attacker should not be able to trick the drive into decrypting data by modifying data stored on the drive mechanism.

Drive Formatting

Drive encryption usually does not affect a hard drive's low-level formatting. The sector headers can remain in plaintext. The sector checksum is calculated on the ciphertext as it is stored on the drive.

High-level formatting places the file system on the drive. Whenever the drive discards a previous CEK and starts using a new one, all previous data becomes unreadable. The owner then must reformat the file system and start over with an empty drive.

When we change keys and reformat the file system, we don't need to rewrite the drive's sectors with new ciphertext encrypted with the new key. When we reformat the file system, it marks unused sectors as being "free." The operating system then assumes that all such

sectors contain garbage. The indecipherable ciphertext in the free sectors is rewritten as the sectors are allocated to new files.

Although most commercial products will use conventional low-level formatting, a drive manufacturer could incorporate a modified sector format to improve security. Additional data might be stored with each sector to help detect the case where an attacker rewrites older sector contents. However, such measures come at the expense of storage capacity. The security improvement must justify the loss of capacity.

9.5.2 Drive Locking and Unlocking

To implement locking and unlocking, the controller implements three separate states: power off, locked, and unlocked. The state diagram in Figure 9.21 shows the relationship between these states.

The arrows between states indicate the events that move the controller from one state to another. The arrows in the state transition diagram must account for every event that may take place in any state.

Initially, the controller's power is off, so it starts in the power-off state. While the power is off, the controller ignores everything until it receives power. When power arrives, the controller starts running and enters the locked state. The controller enters the unlocked state only when the KEK is made available to the controller and the CEK is decrypted. While unlocked, either a reset signal or a loss of power will lock the drive. We keep the drive's data secure by only supplying the key, when conditions are right. Section 9.6 discusses strategies for controlling the keys.

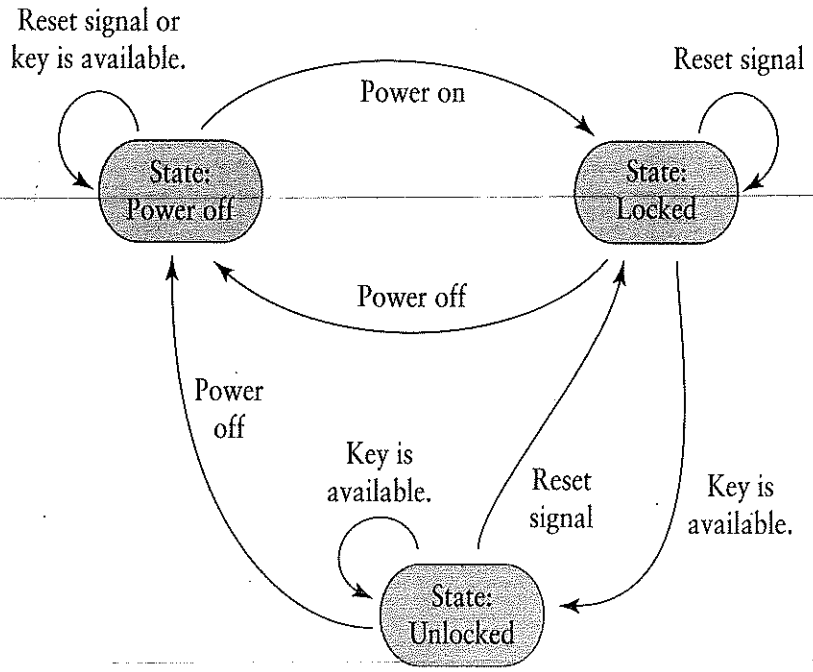


Figure 9.21

Self-encrypting drive controller's state diagram.

Once the controller has unlocked the drive, the computer can read data from the drive. Here is an overview of the process of reading encrypted data from the hard drive. We start with the computer telling the hard drive controller to read data from a sector.

- The interface logic receives the command over the bus. It extracts the details of the command and passes them to the command logic. The command logic sees that it is a "read" operation and that it requests a particular sector.
- The command logic directs the read/write logic to move the heads to a particular track.
- The command logic tells the crypto logic to prepare to read data from a particular sector.
- The crypto logic sets up the encryption to read that sector. As the data arrives from the disk head, the logic decrypts the data and stores it in the buffer.
- As data becomes available in the buffer, the crypto logic feeds it back to the interface logic, which packages the data and delivers it back to the computer.

9.6 Managing Encryption Keys

Encryption keys pose a challenge in any crypto product. This section reviews the following topics:

- Key generation
- Rekeying
- Key storage, in Section 9.6.1
- Booting an encrypted drive, in Section 9.6.2
- Residual key management risks, in Section 9.6.3

Now we examine the issues of key generation and rekeying in drive encryption. The challenges and solutions are similar in software and hardware implementations, so we examine both together.

Key Generation

We have many options for storing and handling keys, but it is best to generate the crypto keys automatically, using a true random number generator. Even though it is challenging to generate truly random numbers inside a typical CPU, software designers have found ways to do this by measuring unpredictable activities inside the operating system. Some software products prompt the user to perform some "random" actions. The software monitors those actions and generates the keying material from them.

Self-encrypting drives may use custom electronic circuits to generate keys. The designers can construct circuits that generate high-quality, truly random numbers. Such circuits often measure noise from a circuit with unpredictable noise patterns. The same circuits also can generate nonces or IVs to use with block cipher modes.

Rekeying

In Section 8.1.1, we talk about the need to change encryption keys occasionally. By their very nature, encrypted hard drives provide a vast amount of ciphertext encrypted with a single key. NIST recommendations suggest to rekey hard drives at least every 2 years. In some environments this may match the drive's useful life span. Thus, we might simply coordinate rekeying with drive replacement.

CEK rekeying poses a challenge on a hard drive because it would take an extremely long time. The controller would have to read each sector on the drive, decrypt it, reencrypt it with the new key, and write it back. Although it might be possible to use the drive during rekeying, it would require very careful implementation by the software designer or drive manufacturer.

Drive vendors rarely provide a CEK rekeying feature. Instead, the owner can change keys by copying the drive to a different self-encrypting drive. The new drive will generate a new CEK when we start it up. Once we have copied the data onto the new drive, we can rekey the old drive and reuse it, which eliminates the aging CEK. Some systems use "RAID" techniques to provide redundancy (see Section 13.5) and, in many cases, the administrator may simply swap a new, blank drive for the old one and let the RAID system reconstruct the removed drive.

9.6.1 Key Storage

Drive encryption systems manage two types of key storage: ***working key storage*** and ***persistent key storage***. The working key storage provides the CEK for the encryption algorithm. In a software implementation, we try to keep this block of RAM in storage at all times. We *never* want the operating system to "swap" this RAM to temporary storage, especially on an unencrypted hard drive. This can be difficult to achieve.

Working Key Storage in Hardware

Matters are somewhat simpler for hardware-based drive encryption. Figure 9.20 shows the storage area for the working key inside the drive's security boundary. The working key resides in *volatile* storage. It is automatically erased whenever the drive is reset or shut down.

The persistent key storage provides *nonvolatile* memory to preserve a wrapped copy of the CEK even when power is turned off. When we mount a drive or unlock it, the wrapped CEK is retrieved from persistent key storage. The drive uses the KEK to decrypt the wrapped key and then saves the CEK in working storage. When we are finished with the drive, the CEK is erased.

The working key storage is inside the drive's security boundary. The storage area should be protected from access while the drive is unlocked. This may involve ***antitamper*** features. For example, the controller may be encapsulated in plastic to prevent an attacker from probing its electronics while it operates. The controller may even include circuits to detect if coverings are removed that otherwise protect the working key storage from tampering. DVD players have used similar encapsulation techniques to protect their internal keys.

Persistent Key Storage

A truly secure encryption key can't possibly be memorized. At best, we might encode a key from a long, memorized phrase, but that's not practical as a general rule. Practical drive encryption requires a strategy for storing the drive keys safely and conveniently. Here are three common strategies:

1. Storing the key in "protected storage"
2. Wrapping the key using a passphrase
3. Storing the key in a file or removable storage device

All three are used in some form in some products. The first approach seems riskiest, because it is hard to protect a block of storage. The second approach is similar to authentication using "something you know." The third approach is similar to authentication using "something you have."

Protected Storage

This approach usually is restricted to hardware products, because it is difficult to enforce storage protection in a pure software product. The product stores the keys in persistent storage inside the drive's security boundary. Upon receipt of an established "unlock code," the drive retrieves the keys from protected storage and provides them to the encryption subsystem. When properly implemented, this approach should provide adequate security. Unfortunately, it is not always implemented reliably.

In late 2009, researchers at SySS, a German security firm, found that several self-encrypting USB drives relied on a predefined, unvarying unlock code. The unlocking process relied on AES encryption to transform the user's password into the unlock code. However, the transformation process always yielded the same unlock code, *regardless of the original password*. Thus, an attacker simply had to know the standard unlock code in order to access *any* USB drive of that design. The details are discussed further in Section 9.6.3.

Key Wrapping

Key wrapping provides a simple, general-purpose approach for storing and protecting persistent key storage. We introduced key wrapping in Section 8.2.2. We can use a passphrase to generate a KEK. When the encryption system retrieves the wrapped key, it uses the KEK to decrypt (unwrap) the drive's encryption key. The encryption system stores the unwrapped key in the working key storage.

If we allow two or more different people to use the drive, we can generate separately wrapped drive keys for each user. The drive protects each set of wrapped keys with its own passphrase. This allows us to grant or revoke access to additional users without having to change the passphrases of other users. Not all products allow this, but it provides us with a safe way of extending access to the drive.

Wrapped keys provide an important security benefit: The encryption key is never unwrapped except when the drive actually is using it. This serves a useful purpose even in hardware-based encryption. If the unwrapped key resides in persistent memory, an attacker with strong electronics experience might be able to read the key. A sophisticated

attacker won't need the controller to read the drive data. Instead, the attacker can extract the encrypted data and decrypt it elsewhere.

Wrapped keys provide an interesting balance of risks and benefits. Because we use passphrases, we can select a memorable phrase that we never need to write down. We still face the risk of losing our credentials: If we lose or otherwise forget the passphrase, we can't unwrap the keys and unlock the drive. However, we also may have the option of creating extra passphrases, including one we write down and store in a strongly protected location, like a safe or lock box.

Managing Removable Keys

If the drive uses removable storage for persistent key storage, then the drive can monitor the presence of the removable key. When the key appears, the drive becomes available; when the removable key disappears, the drive locks itself. Otherwise, the drive behaves exactly like a normal, nonencrypting volume.

It seems rare to see removable keys in the current generation of self-encrypting drives. It was common in older designs. Today, a drive would probably store external keys on a commercial flash memory. The drive may have a socket for a flash memory card or a USB socket for a USB drive.

The owner controls and protects the drive by carefully handling the key. If Alice has such a drive in her laptop, she would have to insert the flash memory containing the key before she could boot the hard drive. Whenever Alice leaves her laptop unattended, she needs to remove the flash memory. She then must either take the flash memory with her or store it in a safe place. An attacker who finds the flash memory can boot her laptop.

Alice faces two problems with this flash memory. First, there is the risk of losing it. If she loses the flash memory and she has no copies of her hard drive's key, she loses the contents of her hard drive. There is no practical way to read the encrypted hard drive once its key is lost.

Alice can make backup copies of her encryption key to avoid the risk of loss. However, this introduces a second problem. Every copy Alice makes of her drive's key will increase the risk of someone else finding it and making a copy.

9.6.2 Booting an Encrypted Drive

Laptops really need encrypted hard drives; they are highly prone to theft. Most owners treat them as very personal objects and store sensitive personal information on them, as well as sensitive company information the owner needs. Laptops have always been targets of theft, both for their intrinsic resale value and for their contents. In the 1990s, laptops stolen at the Reagan National Airport reportedly commanded a premium of two to three times their purchase price in certain criminal circles.

If the encrypted drive uses external keys, booting is simple, as noted earlier. The owner installs the key and the laptop boots normally. Without the key, the drive remains locked.

If the drive uses internal keys, matters are more complicated. When we plug in a removable encrypted drive or file-hosted volume, we run a program to collect a passphrase and mount the drive. We don't have this option when we boot from the encrypted

drive; there is no operating system. We need a program that runs during the bootstrap process. This is called *preboot authentication*.

Preboot Authentication

When the BIOS bootstraps an encrypted drive, it must run a preboot authentication program. There are two strategies for providing preboot authentication:

1. Integrate the program into the BIOS.
2. Include the program on a separate, bootable, plaintext partition.

The operator might not notice a difference in the boot process between these alternatives. Both run essentially the same preboot authentication program. The principal difference is where the program resides. The BIOS-based program is stored in the BIOS-ROM on the motherboard. The plaintext partition takes up space on the hard drive and boots from there.

BIOS Integration

Vendors who integrate self-encrypting drives into large-volume commercial products probably will integrate the preboot authentication into the BIOS. Modern systems are designed so that vendors can easily customize the BIOS.

Disk-Based Authentication

If the encrypted drive is an "aftermarket" upgrade, either hardware or software, then the BIOS-based software might not be available. The drive-resident program requires a small plaintext partition on the hard drive, separate from the larger, encrypted partition, shown in Figure 9.22. This plaintext partition contains a bootstrap program. When we boot this hard drive, the BIOS loads the boot program from the plaintext partition. This

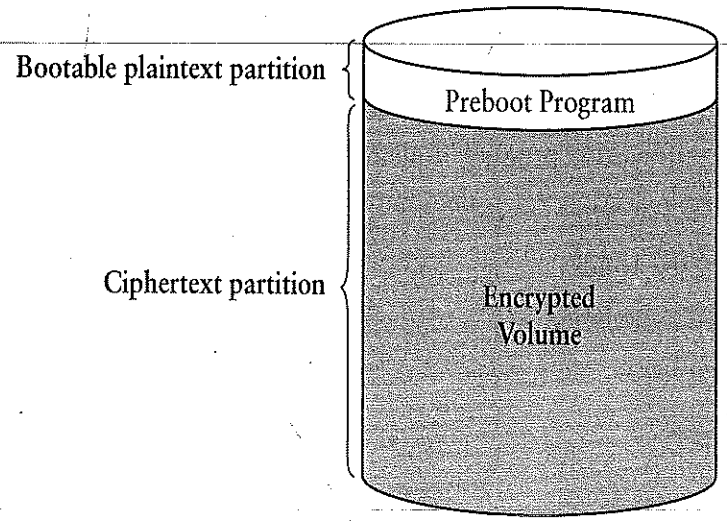


Figure 9.22

Preboot authentication with software encryption.

program collects the volume's passphrase and uses it to decrypt and boot the operating system.

If the encryption is hardware based, then the boot program collects the passphrase and delivers it to the hard drive. The hard drive then unlocks itself and the boot process proceeds from the unlocked hard drive.

The preboot authentication program does not need to restrict itself to passphrases or passwords. If the drive supports more sophisticated authentication (like tokens or biometrics), then the preboot program can provide those credentials instead. The drive controller would need to maintain a database of appropriate base secrets and must have the appropriate software to match the secrets to the credentials.

Automatic Reboot

Computers that serve a lot of people or that perform a vital task really need to be running at all times. If such a computer crashes and restarts, we want it back and running as quickly as possible.

For example, a network server or firewall might suffer a power failure. When power returns, the machine may try to reboot automatically. This is simple if the key resides on a removable flash memory. We simply leave the memory plugged in to the hard drive.

If the drive uses wrapped keys or some other mechanism that requires authentication, we don't want to delay the reboot waiting for preboot authentication. To reboot automatically, the drive must keep a plaintext, unwrapped copy of the drive's encryption key during the crash and restart. The controller then retrieves the key and automatically unlocks the drive.

This option might not protect the hard drive if someone steals it. However, it still provides important protection because we can effectively erase the drive simply by erasing all copies of the encryption keys. It may otherwise be impractical to recycle a modern, large-capacity hard drive.

9.6.3 Residual Risks to Keys

After the volume encryption product implements its protections and we take into account the risks identified in Section 9.1.1, the following four risks remain:

1. Recycled CEK attack
2. Intercepted passphrase
3. Intercepted keys
4. Built-in master key

Recycled CEK Attack

On a self-encrypting hard drive, the wrapped CEK resides in an inaccessible storage area built into the drive controller. Attackers are unlikely to be able to retrieve those wrapped

keys. On software-encrypted volumes, the wrapped CEK is often stored in the volume's partition or file. An attacker may collect such wrapped keys and try to attack them offline.

The owner of the encrypted volume can't protect against such an attack by changing the volume's passphrase. A passphrase change simply rewraps the same CEK with a different passphrase. Instead, the owner must reencrypt the volume with a new CEK.

Intercepted Passphrase

Passphrase interception risks exist in both software and hardware implementations. Figure 9.23 illustrates the points of risk.

1. Install a keystroke logger on the USB cable that connects the keyboard to the computer. The attacker could use a logger like those discussed in Section 6.2.2.
2. Sniff the passphrase inside the computer during the volume-mounting process. This may attack preboot authentication software or OS-based drive-mounting software.
3. Eavesdrop on the drive interface cable and intercept a passphrase being sent to a self-encrypting drive.

The first two risks apply to both hardware and software FDE. The third risk only applies to hardware FDE.

Intercepted Keys

This risk affects hardware and software FDE differently. Most hardware products don't face this risk because the key never leaves the drive. There may be technical attacks that

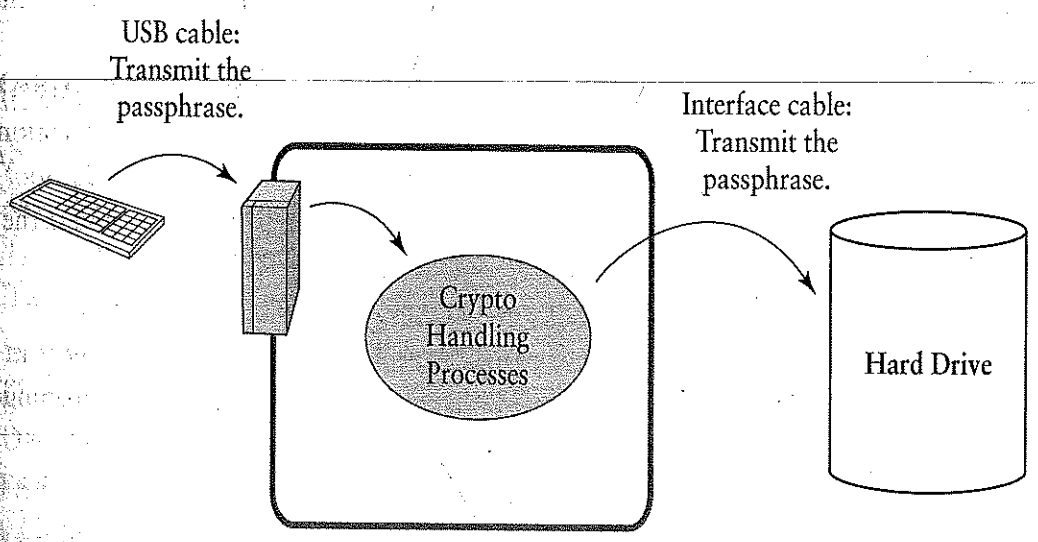


Figure 9.23

Passphrase interception risks.

can extract keys from the electronics, but they require a highly motivated threat. The risk is much greater, of course, if the hardware uses removable keys; then we face the risk of an attacker stealing the key or making a copy of it.

The major software FDE risks involve scavenging keys from RAM. These risks manifest themselves in three ways:

1. Eavesdrop on a software encryption process.
2. Scavenge keys from swap files.
3. Scavenge keys from a powered-off RAM: a *cold-boot attack*.

Eavesdrop on the Encryption Process

While the driver encrypts and decrypts data from the volume, it uses the key schedule derived from the secret key. If another process can retrieve data from the driver's private RAM, the process can sniff the key schedule and derive the secret key from it.

Sniffing Keys from Swap Files

Most modern operating systems use the hard drive to store blocks of RAM in certain cases. Many systems use "paging files" or "swap files," which store blocks of a process' working RAM if the process is blocked for a period of time. Other systems will write RAM to the hard drive when the system goes into a *hibernate* state, in which the system pauses its operation and allows it to resume later.

If the encryption driver's RAM is written to the hard drive, then the key schedule—and possibly the key itself—also may be written to the hard drive.

Cold-Boot Attack

Modern RAMs are not designed to retain their contents after power is turned off. In fact, however, typical RAMs may retain their contents for several minutes, and we can extend retention time by cooling down the RAM chips.

In 2008, researchers at Princeton University demonstrated techniques to retrieve RAM contents after the computer had been powered off. They then extracted drive encryption keys from RAM and used them to decrypt an encrypted drive. This is called a *cold-boot attack*. The attack does not necessarily work against hardware-based systems unless the RAM still contains the drive's passphrase.

It may be difficult to avoid such attacks in all cases. The basic defense is to dismount encrypted drives whenever there is a risk of theft or loss. If the computer is in "sleep" mode with encrypted drives mounted, then the keys also reside in the sleeping RAM. The attacker then can extract the keys from RAM or from saved data on the hard drive.

The "Master Key" Risk

In late 2009, security researchers discovered that several encrypted USB drive products used an egregiously bad authentication protocol. When the drive enrolled a particular

password to authenticate a user or administrator, the drive constructed a special "challenge" value that it subsequently used to authenticate that user.

To authenticate, the drive issued the challenge, and the drive-mounting software constructed the response by encrypting the challenge using the user's enrolled passphrase. If the drive received the right response, it unlocked the drive.

The problem was that the right response was *always* the same. The challenge was tailored to the password to yield the required response. Therefore, the drive always sent a specific challenge to verify a specific passphrase and expected a specific response regardless of the chosen passphrase.

An attacker could build a simplified version of the drive-mounting software that omitted the password entirely. When it sent the standard response, the drive would unlock without a password.

9.7 Resources

IMPORTANT TERMS INTRODUCED

ANTITAMPER
BLOCK CIPHER
DEVICE-HOSTED VOLUME
ERROR PROPAGATION
FILE-HOSTED VOLUME
KEY EXPANSION
KEY SCHEDULE
KEY STREAM DIAGRAM
LOOP DEVICE
MODE DECRYPTION DIAGRAM
MODE ENCRYPTION DIAGRAM
MODES OF OPERATION
PATENT PROTECTION
PERSISTENT KEY STORAGE
PREBOOT AUTHENTICATION
ROUND
S-BOX
SELF-ENCRYPTING DRIVE
TRADE SECRET
TWEAKABLE CIPHER
WORKING KEY STORAGE

ACRONYMS INTRODUCED

ANSI—American National Standards Institute
CBC—Cipher block chaining mode
CFB—Cipher feedback mode
CTR—Counter mode
ECB—Electronic codebook mode
ESSIV—Encrypted sector salt initialization vector
FDE—Full-disk encryption
ITAR—International Traffic in Arms Regulations
IV—Initialization vector
OFB—Output feedback mode
RC2—Rivest's Cipher 2
XTS—Xor-Encrypt-Xor Tweaked Codebook Mode with Ciphertext Stealing

9.7.1 Review Questions

R1. Explain the difference between file encryption and volume encryption.

R2. Explain the fundamental difference between block and stream ciphers.

R3. Describe the steps performed in a typical block cipher.

R4. We encrypt some data with a block cipher. The ciphertext suffers a 1-bit error. How does this affect the plaintext after we decrypt the ciphertext?

R5. Briefly summarize the development of DES and AES.

R6. Does Triple DES yield the same result as single DES if we use a single 56-bit key? Why?

R7. Briefly summarize the history of RC4 and what it tells us about the development of secure encryption algorithms.

R8. Summarize the six qualities of good encryption algorithms.

R9. What is the purpose of block cipher mixing modes?

R10. Describe a simple way to produce a key stream using a block cipher.

R11. Briefly describe the common cipher modes.

R12. Summarize the objectives of a typical FDE implementation.

R13. Explain how software FDE fits into the operating system, relative to other components like the file system and device drivers.

R14. Compare the behavior of CTR, CBC with ESSIV, and XTS modes when encrypting a volume.

R15. Summarize residual risks to information protected by volume encryption. Include key-handling risks.

R16. Summarize the differences between a standard drive controller and a self-encrypting drive controller.

R17. Explain how the boot process takes place when booting a system mounted on an encrypted volume.

9.7.2 Exercises

E1. Search the Internet for the description of a cryptographic algorithm, not including AES, DES, or RC4.

   a. Is it a block or a stream cipher? Provide its "vital statistics," including the supported block sizes and key sizes.

   b. Assess the algorithm in terms of the six recommended properties of an encryption algorithm. For each property, describe how and why the algorithm fulfills—or doesn't fulfill—each property.

E2. Look up the "Feistel structure" and describe how it is related to the structure of block ciphers given in Section 9.2.

E3. For each of the general types of attacks listed in Figure 1.6, describe how volume encryption helps resist that attack. Is there any attack that volume encryption might make more likely to occur?

E4. Tables 3.3 and 3.4 describe the security policy for Bob's computer. Consider the risks that led to this policy. Does Bob's situation justify volume encryption? Why or why not?

E5. Draw a mode encryption diagram illustrating encryption with CTR mode. Where needed, include a function that increments the counter.

E6. Draw a mode decryption diagram for CFB mode.

E7. We have encrypted a three-block message with CFB mode. A 1-bit error occurred in the second ciphertext block. Draw a mode decryption diagram that shows which plaintext bits are affected by error propagation.

E8. We have encrypted a five-block message with CFB mode. An attacker rearranged blocks 2 and 3. Draw a mode decryption diagram that shows which plaintext bits are affected by error propagation.

E9. We have encrypted a three-block message with CBC mode. A 1-bit error occurred in the second ciphertext block. Draw a mode decryption diagram that shows which plaintext bits are affected by error propagation.

E10. We have encrypted a five-block message with CBC mode. An attacker rearranged blocks 2 and 3. Draw a mode decryption diagram that shows which plaintext bits are affected by error propagation.

E11. Find the detailed technical specifications for a commercial hard drive. The specifications will indicate the drive's size in bytes or sectors and its maximum data transfer speed.

a. Identify the hard drive by brand and model number. Identify the drive's size in bytes and its maximum data transfer speed.

b. A simple drive-wiping program will simply write random data over every sector on a hard drive. Given the specifications, calculate how long it would take to wipe all of the data on the drive.

c. A stronger program that tries to "purge" the data may write a series of three different data patterns to each sector. Calculate how long it would take to purge all of the data on the drive.

d. Assume that the hard drive is encrypted. How long would it take to "rekey" the data on the drive?

E12. Supertech has produced a 10 exabyte hard drive. To support the larger capacity, they had to change the sector size to 4096 bytes. Design a counter layout like that shown in Figure 9.16 to support Counter mode encryption on this drive.

E13. We want to implement a drive controller that includes persistent key storage to store-wrapped keys. The design is based on the

controller diagram shown in Figure 9.20.

a. Redraw Figure 9.20 to include the persistent key storage and any other components required for wrapped keys. Include all data paths necessary to support wrapped keys. Be sure to mark paths to indicate if they carry commands, data, or keys. Clearly indicate the security boundary to show which components reside inside and outside the boundary.

b. Using the components in your diagram, give a step-by-step description of drive unlocking.

c. Is the persistent key storage inside or outside the security boundary? Why?

E14. Locate the description of a self-encrypting drive product. Identify the encryption algorithm and cipher mode used. Describe how the drive manages encryption keys. Identify any product certifications held by that product (FIPS 140, for example).

E15. Search the Internet and find a "suspicious" volume encryption product. Identify the elements of the product description that suggest the product may not be reliable.