AES-CBC + Elephant diffuser
A Disk Encryption Algorithm for Windows Vista

Niels Ferguson
Microsoft
niels@microsoft.com

August 2006

Abstract

The Bitlocker Drive Encryption feature of Windows Vista poses an interesting set of security and performance requirements on the encryption algorithm used for the disk data. We discuss why no existing cipher satisfies the requirements of this application and document our solution which consists of using AES in CBC mode with a dedicated diffuser to improve the security against manipulation attacks.

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Contents

1 Introduction

The Enterprize and Ultimate editions of Windows Vista contain a new feature called Bitlocker™ Drive Encryption which encrypts all the data on the system volume. BitLocker imposes some security requirements on the encryption algorithm that are not met by common encryption algorithms and modes. This creates a real problem: a new cipher cannot be trusted without many years of public review, and existing ciphers that satisfy the additional security requirements are either too slow or insufficiently analyzed.

We resolved this dilemma by combining a well-established cipher (AES in CBC mode) with a new component that we call the Elephant diffuser. The basic encryption security is provided by AES-CBC, which has been widely reviewed and is generally used in the industry for encryption. The diffuser layer adds some additional security properties that are desirable in the disk encryption setting but which are not provided by AES-CBC cipher methods.

This combination gives us the best of both worlds. All the security properties traditionally provided by encryption algorithms are provided by AES-CBC, which is an accepted cipher. We only depend on the diffuser for the additional security properties not provided by AES-CBC. The AES-CBC + diffuser approach is also faster than any of the alternatives, which is important for our application.

Fielding any kind of non-standard cryptographic algorithm like the diffuser is bound to be controversial. In this paper we explain the reasons why we made this choice, and document the diffuser so that the public cryptographic community can analyze it.

2 An overview of BitLocker Drive Encryption

Our design is driven by the particular requirements of BitLocker. Therefore, we will first describe BitLocker in its most common setting.

2.1 What it does

BitLocker is a security technology that targets a very specific scenario: the lost laptop.

Laptop computers regularly go missing, either because they are lost or because they are stolen. Our research indicates that typical laptop loss rates are around 1–2% per year. A large organization with 100,000 laptops loses several laptops each day.

These laptops contain confidential information, in the form of documents, presentations, emails, cached data, and network access credentials. This confidential information is typically far more valuable than the laptop hardware, if it reaches the right people. A laptop

is easy to replace at moderate cost; the cost of a data compromise can be many orders of magnitude higher.

BitLocker makes it harder to access this confidential information on a lost laptop. With the current generation of operating systems it is very simple to break into a laptop. One obvious way is to take the disk drive out of the laptop and connect it to a second machine as an auxiliary drive. All data can now be accessed using the administrator (root) privileges of the second machine. An even easier solution is to use a bootable floppy disk, CD, or USB key that contains a script that resets the Administrator (root) password. Such scripts and disks are available on-line, and anyone with an Internet connection and half an hour of time can download them. Once the Administrator password is reset, the laptop can be booted and the attacker can log in as the Administrator, giving him complete access to all information on the laptop.

The classic solution to this problem is to run a low-level disk encryption driver with the key provided by the user (passphrase), a token (smart card) or a combination of the two. The disadvantage of the classic solution is the additional user actions required each time the laptop is used. Most users are unwilling to go through these extra steps, and thus most laptops are unprotected.

BitLocker improves on the classic solution by allowing the user actions during boot or wake-up from hibernate to be eliminated. This is both a huge advantage and a limitation. Because of the ease of use, corporate IT administrators can enable BitLocker on the corporate laptops and deploy it without much user resistance. On the downside, this configuration of BitLocker can be defeated by hardware-based attacks.

Hardware attacks require the attacker to have significant skill and/or very specialized hardware, whereas software-only attacks can usually be automated, distributed over the internet, and carried out without any knowledge of the details of the system. We can therefore reasonably expect that the number of people that is capable and equipped to perform a hardware attack is far smaller than the number of people capable of performing a software-only attack. Thus, BitLocker significantly reduces the risk of data compromise, and makes it more likely that the laptop will simply be sold for the hardware value, rather than the value of the information on it.

The remaining vulnerability to hardware-based attacks seems fundamental for systems without user actions on boot. The cryptographic keys used to protect the confidential data must be available to the laptop during a normal boot, and can therefore be recovered by a hardware attack.

Stopping hardware attacks is possible, but requires the use of a token (e.g. USB key) and/or a user-memorized password or PIN. These options are fully supported by BitLocker, and they improve the security of the system. However, we expect that a large number of laptops will be used without PIN or USB key to avoid the need for user action on each reboot.

2.2 How it works

BitLocker obviously cannot be a software-only technology. Every software-only solution is vulnerable to software-only attacks.

BitLocker makes use of the TPM security chip which will be incorporated in most PCs in the near future. The TPM is a tamper-resistant chip mounted on the motherboard. Though the TPM has many functions [5], BitLocker uses only a few basic ones. Our description of the TPM is simplified and only covers those parts relevant for our purpose.

The TPM keeps several Platform Configuration Registers, or PCRs. At power-up the PCRs are set to zero. PCRs are only modified by the `extend` function which (effectively) sets a PCR to the hash of its old value and a supplied data string. We can think of a PCR as a hash over all the data strings provided in extend function calls for that PCR. There is no other way to set the value of a PCR, so if a PCR has value x after a sequence of extends, then the only way to reach the value x again is to perform the exact same sequence of extends after a power-up.

The seal/unseal functions of the TPM allow selective access to cryptographic keys based on PCR values. The seal function is used to encrypt a key into a string which can only be decrypted by that same TPM. Furthermore, the TPM will decrypt the string if and only if the selected PCRs have the value that was specified during the seal operation. In other words: we can store a key in an encrypted string so that it can only be accessed when selected PCRs have a particular value.

During the boot process the PCRs are used to keep track of the code that runs. The key used to encrypt the disk is sealed against a particular set of PCR values. During a normal boot the PCRs reach the same values, and the key can be unsealed by the TPM. If an attacker boots into any other operating system, the machine will be fully functional but the PCR values will be different and the TPM will not unseal the key. Thus, other operating systems cannot read the data on the disk, or find out how to modify the disk to reset the Administrator password.

In more detail, the process is the following: at power-up the processor starts running the BIOS from ROM. The first part of the BIOS cannot be modified. This part extends the BIOS PCR with the entire BIOS code and proceeds with the rest of the BIOS startup. After BIOS initialization the BIOS reads the Master Boot Record (MBR) of the hard disk, extends the boot sector PCR with the sector's data, and then executes the code in the boot sector. The boot sequence of a PC contains several more iterations, but in each case the newly-loaded code is first measured using an `extend` function before it is executed.

These functions do not interfere with the boot process of another operating system. Other operating systems can boot normally; but the TPM PCRs will have a different value. The PCRs merely report what software was run during the boot process.

Although extending PCRs provides good authentication of the code that is being run,

it is extremely inflexible. Any change to the code, for example for a patch or upgrade, leads to a different PCR value, and requires that the keys be re-sealed to the new PCR values. Therefore, the boot sequence switches to using BitLocker encryption at the first opportunity. Before the switch, PCRs are used to measure what code is running. At the switch point the TPM unseals the BitLocker volume encryption key. After the switch, all further data is read from the encrypted volume. Though BitLocker does not authenticate the data it reads from the disk in a cryptographic sense, it is very hard to perform a meaningful manipulation of the encrypted data.

2.3 Disk encryption and authentication

BitLocker encrypts almost all the data on the hard-disk. More precisely, it encrypts all the data on the operating system volume, which for most PCs spans almost the whole hard disk. The encryption protects the confidentiality of the data, which is a straightforward cryptographic problem, and can be solved with traditional ciphers.

However, to have a secure boot process we also need to authenticate the data from the disk. We don't want an attacker to modify the OS code in order to weaken the OS security. The normal cryptographic solution is to use an authentication code, but as we will see this is not practical. Therefore, we are left with using the encryption as a poor-man's authentication.

Imagine an attacker trying to break into the laptop using only software. He boots into some other OS. Because the PCRs are different he can't unseal the BitLocker key, so he can't read the encrypted volume. However, the attacker can modify the ciphertext on the hard disk in the hope of introducing a weakness in the OS. He then boots the machine normally, and exploits the created weakness during the boot or login process to gain access to the machine.

2.3.1 Why not use a MAC?

The obvious cryptographic solution to this problem is to add a Message Authentication Code (MAC) to each block of data on the disk. Disks store information in fixed-size sectors. Sectors are typically 512 bytes, though in the near future this will grow up to 4096 or even 8192 bytes. The operating system is designed to use sectors whose size is a power of two. Furthermore, the operating system accesses individual sectors of the disk in random order. These properties impose two constraints.

The first constraint is that the BitLocker encryption is done on a per-sector basis. Each sector is encrypted and decrypted independently of the other sectors. The second constraint is that the ciphertext cannot be larger than the plaintext. There is no extra room to store additional data. This means that we cannot store any nonce, IV, or MAC value with the

ciphertext. Together these constraints imply that the disk is effectively encrypted with a large block cipher in ECB mode where the block size is the sector size.

There are good engineering reasons behind these constraints. The encryption/decryption of one sector cannot depend on any other sector. There are applications, such as databases, that rely on the fact that they can write to sector x without danger of damaging sectors $x-1$ or $x+1$. They use this property to ensure that no information (other than possibly the sector that is being written) is lost in case of a crash or power failure. Suppose the encryption algorithm works in larger blocks than a single sector. To write sector x, the system first has to read the other sectors in the block, decrypt them, encrypt the new block, and then write all the sectors that make up the block. If the power fails when some of the sectors in the new block have been written but others have not, then the block has been corrupted. More importantly, unless the cipher works on a per-sector basis, writing sector x could corrupt some *other* sector. This violates the expectation of the application, and can lead to a loss of reliability of mission-critical applications. That is clearly unacceptable. (And re-writing all applications that depend on this property is impossible.)

It is tempting to add a nonce or MAC value to the ciphertext. However, making the ciphertext larger than the plaintext is not feasible. Both the disk and the operating system work in sectors whose size is a power of two. We could map a 512 byte OS sector into a 1024 byte disk-sector, but that would entail the loss of half the disk capacity, a price users are not willing to pay. We could reserve one in every 16 sectors to store the MAC and nonce values for the other 15 sectors, but this has several problems. First of all, writing to sector x means updating an additional sector that contains the nonce and MAC. This turns a write into a read-then-write with the associated performance loss. Furthermore, it could damage the nonce/MAC sector (e.g. if there is a power failure), which would lead to the loss of the other 14 data sectors; also unacceptable. Finally, for various usability, manageability, and deployment reasons, it must be possible to enable and disable BitLocker on an existing disk. (Think of a user upgrading to a new Windows version that includes BitLocker and enabling BitLocker on an existing disk.) Adding nonce/MAC sectors modifies the disk layout (and reduces the amount of available disk space) making it extremely complicated to enable and disable in-place. Add to that the various failure modes that can happen during the conversion, and it becomes infeasible to do this conversion in a reliable way.

The final conclusion is that we cannot add a MAC to each disk sector in a way that would be acceptable to most users.

2.4 Poor-man's authentication

That leaves us with an encryption algorithm that provides no authentication, yet we need authentication to provide a secure boot process. The best solution is to use poor-man's authentication: encrypt the data and trust to the fact that changes in the ciphertext do

not translate to semantically sensible changes to the plaintext.[1] For example, an attacker can change the ciphertext of an executable, but if the new plaintext is effectively random we can hope that there is a far higher chance that the changes will crash the machine or application rather than doing something the attacker wants.

We are not alone in reaching the conclusion that poor-man's authentication is the only practical solution to the authentication problem. All other disk-level encryption schemes that we are aware of either provide no authentication at all, or use poor-man's authentication.

To get the best possible poor-man's authentication we want the BitLocker encryption algorithm to behave like a block cipher with a block size of 512–8192 bytes. This way, if the attacker changes any part of the ciphertext, all of the plaintext for that sector is modified in a random way. We also want to prevent the attacker from moving the ciphertext of one sector to another sector, so the encryption algorithm should behave as a tweakable block cipher [8] with a slightly different algorithm for each sector.

For completeness we should mention that there are other authentication mechanisms that are used by the OS during the boot process, such as checking digital signatures on executables. Though these mechanisms are very valuable, they do not cover all of the data used during the boot and login process. From our point of view, these other mechanisms provide a second line of defense for some of the data, but we will ignore them for the rest of our discussion.

BitLocker also allows users to use a PIN that the TPM checks, or a USB key that contains a cryptographic key. Without the right PIN or USB key the laptop doesn't have the right information to even find the disk decryption key, so the information is safe unless the PIN is written on a post-it stuck to the machine, or the USB key is left in the laptop bag. In practice, we expect that many laptops will be used in the TPM-only mode and that scenario is the main driver for the disk cipher design.

2.5 Performance

The BitLocker disk cipher must be fast. If using BitLocker results in a significant and noticeable slowdown of the laptop there will be great user resistance to its deployment. When talking to customers about BitLocker, the question of performance is always one of the first questions they ask. Our analysis concluded that the performance loss of BitLocker must be minor for the feature to be used by a large number of customers. And given that Microsoft is not in the business of providing niche solutions, good performance was one of the hard requirements for BitLocker.

[1] This is not a compromise we like, but the only one that makes sense for the problem we're trying to solve.

A typical desktop machine today has a 3 GHz P4 CPU and a hard disk that can read at about 50 MB/s. That means that the CPU has 60 clock cycles for each byte that the disk reads. Laptops have slower CPUs, often around the 1 GHz mark. Laptop disks are also slower but not by nearly as much. (For example, the Seagate Momentus 5400.2 laptop drive can read data at almost 50 MB/s.) Our data shows that laptops tend to have fewer CPU clock cycles per byte read from disk, down to 40 or even 30 cycles per byte. We cannot predict what the CPU/disk speed ratio will be for the actual hardware that BitLocker will run on, but these numbers are the best guidelines we have.

If decryption is slower than the peak data rate of the disk, the CPU becomes the bottleneck when reading large amounts of data. This is very noticeable, both because of the reduced performance and because of the reduced responsiveness of the UI when all CPU time is being used to decrypt data.[2] Therefore, decryption, including all overhead, must be faster than the disk to get an acceptable user experience.

BitLocker is carefully designed to overlap the reading of data from disk with the decryption of previously read data. This is only possible to a limited extent, and when the disk finishes reading the data, the CPU still has to decrypt (some of) the data. Thus the decryption time increases the latency of the disk request and reduces performance accordingly. This obviously argues for a fast decryption algorithm.

A software implementation of AES runs in around 20–25 cycles per byte on a P4 class CPU. (Synthetic benchmarks can achieve somewhat higher speeds, but they exclude various overheads encountered in real system implementations.) Other overhead adds around 5 cycles per byte for a total of 25–30 cycles per byte.

Based on this data, our performance analysis concluded that a single pass of AES, for example using AES in CBC mode, would have acceptable performance. An algorithm twice as slow as AES (45–55 cycles/byte) would be on the edge of being unacceptable, and a high-risk choice given the many uncertainties in the analysis. Anything slower than that would be unacceptable.

2.6 BitLocker encryption algorithm requirements

We get the following major requirements for our BitLocker encryption algorithm:

- It encrypts and decrypts disk sectors of size 512, 1024, 2048, 4096, or 8192 bytes.
- It takes the sector number as an extra parameter (the tweak) and implements different encryption/decryption algorithms for each sector.
- It protects confidentiality of the plaintext.

[2] There are many interesting issues involved in this analysis, including CPU scheduling, prioritization, and deadlocks, but we will not go into those details in this paper.

- It is fast enough that the slow-down of the laptop is acceptable to most users. Our best estimate is that a speed of 40 cycles/byte or faster will be acceptable.
- It has been validated by public scrutiny, and is generally considered safe.
- An attacker cannot control or predict any aspect of the plaintext changes if he modifies or replaces the ciphertext of a sector.

This is an extremely challenging set of requirements. Initially we tried to find a cipher that satisfies all the requirements, but we found no suitable cipher. The performance and public scrutiny requirements are especially difficult to satisfy.

2.7 Attack model

Normally, block ciphers are designed to withstand chosen plaintext and ciphertext attacks where the attacker has access to a black box that performs both the encryption and decryption operation. Also, the cipher is considered to be broken if it can be distinguished from a randomly chosen permutation.

Ideally our disk cipher (with the large 512–8192 byte block size) would satisfy these requirements, but we were unable to find a solution that achieves this and the other requirements too. To find a better solution we examined the BitLocker attack model in more detail to be able to take advantage of the specifics of the model.

In the BitLocker attack model we assume that the attacker has chosen some of the plaintext on the disk, and knows much of the rest of the plaintext. Furthermore, the attacker has access to all ciphertext, can modify the ciphertext, and can read some of the decrypted plaintext. (For example, the attacker can modify the ciphertext which stores the startup graphic, and read the corresponding plaintext off the screen during the boot process, though this would take a minute or so per attempt.) We also assume that the OS modifies some sectors in a predictable way during the boot sequence, and the attacker can observe the ciphertext changes.

However, the attacker cannot collect billions of plaintext/ciphertext pairs for a single sector. He cannot run chosen plaintext differences through the cipher. (He can choose many different plaintexts, but they are all for different sectors with different tweak values, so he cannot generate chosen plaintext differences on a single sector.) And finally, though this is not a cryptographic argument, to be useful the attack has to do more than just distinguish the cipher from a random permutation.

In short, we have the following attack model:

- The attacker has many known (but not chosen) plaintext/ciphertext pairs for different sectors.

- The attacker has the ciphertexts for a large number of chosen plaintexts for different sectors. The plaintexts are chosen before the attacker gets access to the laptop.
- The attacker has access to a slow decryption function for some of the sectors.
- The attacker gets several ciphertexts of plaintexts for the same sector with a known (but not chosen) difference.

The attacker succeeds if he can modify a ciphertext such that the corresponding plaintext change has some non-random property.

We don't want to argue that the standard attack model for block ciphers is wrong. Quite the opposite. We'd love to have a disk sector cipher that satisfies the standard requirements, but we have not been able to find one that satisfies the performance and validation requirements.

3 Existing ciphers

Before we dive into the details of our new design we'll discuss existing ciphers and why they are unsuitable.

3.1 Stream ciphers

There are many stream ciphers, but by their very nature, they allow the attacker to flip arbitrary bits in the plaintext. This lack of diffusion makes them entirely ineffective for poor-man's authentication.

3.2 AES-CBC

Any time you want to encrypt data, AES-CBC is a leading candidate. In this case it is not suitable, due to the lack of diffusion in the CBC decryption operation. If the attacker introduces a change Δ in ciphertext block i, then plaintext block i is randomized, but plaintext block $i + 1$ is changed by Δ. In other words, the attacker can flip arbitrary bits in one block at the cost of randomizing the previous block. This can be used to attack executables. You can change the instructions at the start of a function at the cost of damaging whatever data is stored just before the function. With thousands of functions in the code, it should be relatively easy to mount an attack.

The current version of BitLocker implements an option that allows customers to use AES-CBC for the disk encryption. This option is aimed at those few customers that have formal requirements to only use government-approved encryption algorithms. Given the weakness of the poor-man's authentication in this solution, we do not recommend using it.

3.3 Bear and Lion

Bear and Lion are two large-block block ciphers proposed by Ross Andersen and Eli Biham [1]. Bear and Lion are very similar in construction. They split the data block into two unequal parts and create a 3-round Luby-Rackoff cipher by using a keyed hash function and a stream cipher to construct the round functions. The difference is that Bear uses two keyed hash function rounds and one stream cipher round whereas Lion uses one keyed hash function round and two stream cipher rounds.

Bear and Lion seem ideally suited, except for the fact that they are too slow. Both ciphers make three passes over the data. If we were to use SHA-256 for the hash function and AES-CTR for the stream cipher, the overall cipher would need close to 100 cycles/byte.

We tried several ways to make Bear/Lion suitable for BitLocker. However, the only way to make this solution fast enough is to use a fast stream cipher and a fast keyed hash function. We could not find suitable candidates that have had enough public review and have not yet been broken.

Our best attempt used AES-CBC for the keyed hash function, and AES-CTR for the stream cipher. Unfortunately, this solution is not fast enough.

3.4 Beast

Beast is a variation of Bear [9]. It is faster than Bear because it replaces the last round of Bear by a function that does not process the entire data block. Unfortunately, this change destroys the diffusion properties of the decryption function, making it unsuitable for our purpose.

We did consider using Beast upside-down, using the decryption function for encryption and the encryption function for decryption. This would seem to solve the immediate problem of the lack of diffusion. Even in this mode we do not feel comfortable with Beast. As far as we know Beast has not received any public review, and there has been no analysis of the upside-down mode. A final consideration is the speed of Beast. Though faster than Bear, it still requires two passes over the data; one with a hash function and one with a stream cipher. Our performance estimates for Beast with a well-established hash function and stream cipher were still slow enough to be a real problem.

3.5 VIL

VIL is a block cipher mode of operation that encrypts and decrypts arbitrary length messages [2]. This mode has received little attention, partly because it is patented.

For our purpose, the VIL mode is not suitable as most of the message is encrypted with a stream cipher. This provides no diffusion at all, and therefore very bad poor-man's-authentication properties.

3.6 Mercy

Mercy is a block cipher specifically designed for disk sector encryption [3]. Unfortunately, it was broken in 2001 by Scott Fluhrer [4], which eliminates it as a candidate.

3.7 LRW

LRW is a block cipher mode proposed by Liskov, Rivest, and Wagner. In LRW, a change to one block of the ciphertext results in a random change to the corresponding block of the plaintext, and no other changes. Effectively, it allows an attacker to randomize any block of plaintext.

LRW provides some level of poor-man's authentication, but the relatively small block size of AES (16 bytes) still leaves a lot of freedom for an attacker. For example, there could be a configuration file (or registry entry) with a value that, when set to 0, creates a security hole in the OS. On disk the setting looks something like `"enableSomeSecuritySetting = 1"`. If the start of the value falls on a 16-byte boundary and the attacker randomizes the plaintext value, there is a 2^{-16} chance that the first two bytes of the plaintext will be `0x30 0x00` which is a string that encodes the ASCII value '0'.

For BitLocker we want a block cipher whose block size is much larger. The same type of attack is still possible, but it is made harder by two factors: any particular attack point is far less likely to be on a suitable block boundary, and the attacker is forced to randomize more plaintext, increasing the likelihood that he will damage other parts of the system and crash the PC rather than open a usable hole.

3.8 CMC and EME

CMC and EME are two block cipher modes proposed by Halevi and Rogaway [7, 6]. They are directly targeted at our problem: encryption of disk sectors. CMC and EME have all the desired security properties, and are the leading contenders from the existing ciphers we identified.

However, they have not been widely studied or deployed, making them a relatively high-risk choice from a security point of view. (An earlier version of CMC was in fact broken.)

Furthermore, CMC and EME both require two AES encryptions for each block of data, making them a high-risk choice from a performance point of view.

A contributing factor in our decision not to use CMC or EME was that they are patented, and clearing the patent situation would require too much time.

3.9 Conclusion

In late 2004 we spent a lot of time looking for a suitable existing cipher, and failed to identify one. CMC and EME were the closest contenders, but for the reasons listed above we did not believe that they were suitable choices.

4 AES-CBC + diffuser

We finally chose to use AES-CBC for the primary encryption, and add a dedicated independently keyed diffuser to the plaintext side. This choice has advantages and disadvantages.

On the advantage side, it is trivial to see that AES-CBC + diffuser is at least as secure as AES-CBC, the industry workhorse algorithm for encryption. (See appendix A.) This provides a guaranteed security level for the confidentiality of the data, and one that customers can understand. The diffuser runs in about 10 clock cycles/byte so that the combination with AES-CBC satisfies our performance requirements.

On the disadvantage side, the diffuser is a new unproven algorithm, and this inevitably leads to questions. Without extensive public scrutiny and analysis of an algorithm there is a justified scepticism about its security. People are reluctant to trust new algorithms.

So why did we choose this option anyway? In our final analysis we decided this was the better choice for our product. The performance gain over the alternatives was important enough to outweigh the disadvantages of a new diffuser algorithm. Time will tell whether we made the right choice.

It bears repeating that even if the diffuser algorithm is utterly broken, all the data is also encrypted using AES-CBC and the confidentiality is still ensured. The only task of the diffuser is to make manipulation attacks harder by providing better poor-man's authentication than plain AES-CBC provides. Given the limited BitLocker attack model we consider it extremely unlikely that a practical attack against the diffuser will be found.

4.1 Overview

Figure 1 gives an overview of our solution. There are four separate operations in each encryption. The plaintext is exclusive-orred (xorred) with a sector key, then run through two (unkeyed) diffusers, and finally encrypted with AES in CBC mode.

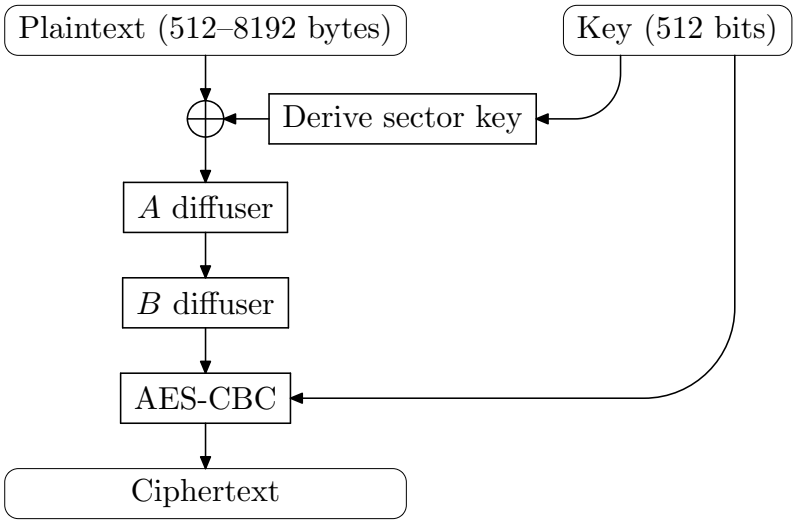

Figure 1: An overview of AES-CBC + diffuser

The sector key component and the AES-CBC component are independently keyed which allows an easy proof that our construction is at least as secure as AES-CBC (see appendix A). Both components are provided with 256 bits of key material, so that the full key is 512 bits. Depending on the selected version, the two keyed components can use fewer than 256 bits each, so some of the key bits may go unused. The full key is always 512 bits to support larger keys without any changes to the key management system. By default both the sector key and the AES-CBC layer use 128-bit AES keys; a version that uses 256 bits for each is also available.

The block size is variable: it can be any power of two within the range 512–8192 bytes (4096–65536 bits).

4.2 AES-CBC

The AES-CBC component is straightforward. The AES key K_{AES} is either 128 bits or 256 bits, depending on the selected version. The block size is a always a multiple of 16 bytes, so no padding is necessary. The IV for sector s is computed as:

$$IV_s := E(K_{\mathrm{AES}}, e(s))$$

where $E()$ is the AES encryption function, and $e()$ is an encoding function that maps each sector number s into a unique 16-byte value. Note that IV_s depends on the key and the sector number, but not on the data.

The plaintext is encrypted using AES-CBC and the IV for the sector. Decryption is the obvious inverse function.

The result of $e()$ is the tweak value of this part of the cipher. The choice of $e()$ has no security implications (as long as it is an injection) and will vary with the application. For BitLocker the encoding function e is the simplest one for the implementation. The first 8 bytes of the result are the byte offset of the sector on the volume. This integer is encoded in least-significant-byte first encoding. The last 8 bytes of the result are always zero.

4.3 Sector key

The sector key for sector s is defined by:

$$K_s := E(K_{\text{sec}}, e(s)) \parallel E(K_{\text{sec}}, e'(s))$$

where $E()$ is the AES encryption function, K_{sec} is the 128 or 256-bit key for this component, $e()$ is the encoding function used in the AES-CBC layer, and $e'(s)$ is the same as $e(s)$ except that the last byte of the result has the value 128.

The sector key K_s is repeated as many times as necessary to get a key the size of the block, and the result is xorred into the plaintext.

4.4 Diffusers

The A and B diffusers are very similar, but work in opposite directions. Our core diffuser design has good diffusion properties in one direction and bad diffusion properties in the other direction. Having two diffusers provides good diffusion in both directions.

The diffusers have been designed in the decryption direction, as decryption is the more common operation. We will describe them first in the decryption direction, and later show the corresponding encryption function.

Each diffuser interprets the sector data as an array of 32-bit words, where each word is encoded using the least-significant-byte first convention. Let n be the number of words in the sector, and $(d_0, d_1, \ldots, d_{n-1})$ be the words of the sector. For index values outside the range we define $d_i := d_{i \bmod n}$ to allow easy wrap-around without confusing notation.

The decryption function of the A diffuser is given by:

$$\begin{aligned} &\textbf{for } i = 0, 1, 2, \ldots, n \cdot A_{\text{cycles}} - 1 \\ &\qquad d_i \leftarrow d_i + (d_{i-2} \oplus (d_{i-5} \lll R^{(a)}_{i \bmod 4})) \end{aligned}$$

The value i is a loop counter that goes around the data array A_{cycles} times. (Remember that all indices are modulo n, so the wrap-around is automatic.) The addition is modulo

2^{32}, $\lll$ is the rotate-left operator, and $R^{(a)} := [9, 0, 13, 0]$ is an array of 4 constants that specify the rotation amounts.

The corresponding encryption function of the A diffuser is easy to derive:

$$\begin{aligned}&\textbf{for } i = n \cdot A_{\text{cycles}} - 1, \ldots, 2, 1, 0\\ &\qquad d_i \leftarrow d_i - (d_{i-2} \oplus (d_{i-5} \lll R^{(a)}_{i \bmod 4}))\end{aligned}$$

The asymmetric diffusion properties are easy to see. If we look at the A diffuser decryption, the result of one iteration is used 2 and 5 iterations later which quickly propagates changes to the rest of the sector. In the encryption direction the output of one iteration is used $n - 5$ and $n - 2$ iterations later, which provides a much slower diffusion.

The B diffuser is very similar. It has good diffusion in the encryption direction. The B diffuser decryption function is given by

$$\begin{aligned}&\textbf{for } i = 0, 1, 2, \ldots, n \cdot B_{\text{cycles}} - 1\\ &\qquad d_i \leftarrow d_i + (d_{i+2} \oplus (d_{i+5} \lll R^{(b)}_{i \bmod 4}))\end{aligned}$$

where $R^{(b)} := [0, 10, 0, 25]$. The B diffuser encryption function is

$$\begin{aligned}&\textbf{for } i = n \cdot B_{\text{cycles}} - 1, \ldots, 2, 1, 0\\ &\qquad d_i \leftarrow d_i - (d_{i+2} \oplus (d_{i+5} \lll R^{(b)}_{i \bmod 4}))\end{aligned}$$

The constants A_{cycles} and B_{cycles} define how many times each of the diffusers loop around the sector, and are chosen as $A_{\text{cycles}} := 5$ and $B_{\text{cycles}} := 3$.

To choose the rotation amounts we ran experiments on the diffusion properties of our diffuser recurrence (in the high-diffusion direction). For performance reasons we only use a nonzero rotation amount every other iteration (the P4 processor has a very slow rotation instruction). We concentrated on how quickly a single bit difference diffuses through the 32-bit active word. Our results are that if we flip a single bit in d_i, each of the bits of d_{i+43} has a chance of at least 1/3 of flipping in a single forward cycle of the diffuser.[3] As a sector is at least 128 words long, we get full diffusion within the word in about one third of a cycle.

4.5 About the name

The name Elephant was chosen for the diffuser component to fit in with the Bear and Lion ciphers discussed earlier.

[3]This holds for the forward B diffuser. For the backward A diffuser a bit in d_i flips bits in d_{i-43} with probability 1/3.

5 Performance

Our AES implementation uses about 20 cycles/byte for AES-CBC on a a Pentium 4. The diffuser takes about 10 cycles/byte. The overall cipher speed is just over 30 cycles per byte, including various overhead. This implies that the cipher is faster than the peak data rate of a typical disk.

Our current BitLocker implementation manages to limit the loss of performance to around 5% averaged over our test cases. Our typical end-user test scenarios show an even smaller overhead. This is good enough to allow widespread adoption of this security technology.

6 Analysis

An extensive analysis of our construction is still ongoing, and will be published separately.

7 Use of AES-CBC + diffuser

This cipher is designed specifically for the role of disk sector encryption algorithm in the BitLocker setting. It is *not* a general-purpose block cipher, and should not be used in other settings without careful analysis. As a pure block cipher our construction has many weaknesses when analyzed in the standard block cipher attack model. For example, some of the key bits are unused, and guessing part of the key is enough to distinguish it from a random permutation.

8 Acknowledgements

I'd like to thank Josh Benaloh for his helpful comments on the design, and the System Integrity team at Microsoft for their tireless energy in bringing this to actual use.

References

[1] Ross Anderson and Eli Biham. Two practical and provable secure block ciphers: BEAR and LION. In Dieter Gollmann, editor, *Fast Software Encryption: Third International Workshop (FSE'96)*, LNCS 1039, pages 113–120. Springer Verlag, 1996.

[2] Mihir Bellare and Phillip Rogaway. On the construction of variable-input-length ciphers. In Lars Knudsen, editor, *Fast Software Encryption: 6th International Workshop, FSE'99*, LNCS 1636, pages 231–244. Springer Verlag, 1999.

[3] Paul Crowley. Mercy: a fast large block cipher for disk sector encryption. In Bruce Schneier, editor, *Fast Software Encryption: 7th International Workshop, FSE 2000*, LNCS 1978, pages 49–63. Springer Verlag, 2001.

[4] Scott R. Fluhrer. Cryptanalysis of the Mercy block cipher. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001*, LNCS 2355, pages 28–36. Springer Verlag, 2002.

[5] Trusted Computing Group. *TCG TPM Specification Version 1.2*. Available from `www.trustedcomputinggroup.org`.

[6] Shai Halevi and Phillip Rogaway. A parallelizable enciphering mode. `http://eprint.iacr.org/2003/147`, 2003.

[7] Shai Halevi and Phillip Rogaway. A tweakable enciphering mode. `http://eprint.iacr.org/2003/148`, 2003.

[8] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology—CRYPTO 2002*, LNCS 2442, pages 31–46. Springer Verlag, 2002.

[9] Stefan Lucks. BEAST: A fast block cipher for arbitrary blocksizes. In Patrick Horster, editor, *Communications and Multimedia Security II, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security*, IFIP Conference Proceedings 70, pages 144–153. Chapman & Hall, 1996.

A Sketch of a proof that AES-CBC + diffuser is as secure as AES-CBC

It is not possible to prove that a particular algorithm is secure. But there are techniques for showing security implications: proving that algorithm A is as secure as algorithm B.

In our case we want to show that AES-CBC + diffuser is at least as secure as using just AES-CBC. We won't do a formal proof, which requires lots of formal definitions, but we present a sketch of the argument.

Suppose an attacker is attacking two identical disks, one encrypted with AES-CBC and one with AES-CBC + diffuser. The exact attack model (what the attacker can do with the disks during the attack) is not important here, as long as it is the same for both disks.

The crucial observation is that giving the attacker more information cannot make it harder for him to perform the attack. The attacker is always free to ignore any additional information we give him. Providing more information can make it easier to attack the system, but never harder.

So here is what we do: we give the attacker the key K_{sec} used for the AES-CBC + diffuser disk. This is the AES key used to derive all the sector keys. This means that the attacker can now compute every sector key, and perform all the diffuser operations on any plaintext. In effect, the sector key and diffuser operations become transparent to the attacker, and the remaining problem is to attack the AES-CBC layer, which is exactly the problem of attacking a disk encrypted with only AES-CBC. We have helped the attacker significantly, and he is still faced with attacking a disk encrypted with AES-CBC. This shows that AES-CBC + diffuser cannot be easier to attack than just AES-CBC.