Visualisation of potential weakness of existing cipher engine implementations in commercial on-the-fly disk encryption software

C. B. Roellgen

Global IP Telecommunications, Ltd. & PMC Ciphers, Inc.

August 15, 2008

Abstract

Symmetric ciphers like AES are used to encrypt data in commercial disk encryption software. As key size is usually limited and as key setup times are considered by developers to be pretty long, ciphers are either used in ECB mode or a block counter is added to the plaintext prior to the process of encryption. It could be shown that disk encryption software that is usual in trade is susceptible to the presented ciphertext-only attack on backups of image files. The attack is described in this paper in a visual way.

Key words: Backup Attack , AES, advanced, encryption, standard, ECB, CBC, LRW, block cipher, polymorphic cipher, image, bitmap, chosen, plaintext, ciphertext, on-the-fly, OTFE, on-the-fly, one-time-pad

1. Introduction

AES or any other symmetric cipher may be used without re-keying and even without adding a block number to the plaintext (Electronic Code Book (ECB) mode). Especially for encrypting data on mass storage devices, re-keying might only occur for every new sector or even only when an encrypted volume is opened.

In order to visualize the effects, we've simulated the encryption of an image.
For our tests we've used the original image to the left as well as versions with a color palette size of only four colors. By doing this, we've been able to create large regions with a uniform color.

2. Encryption of a bitmap with different cipher operating modes

In oder to visualize the attack, typical operating modes of ciphers used in On-The-Fly-Encryption software are described in this section.

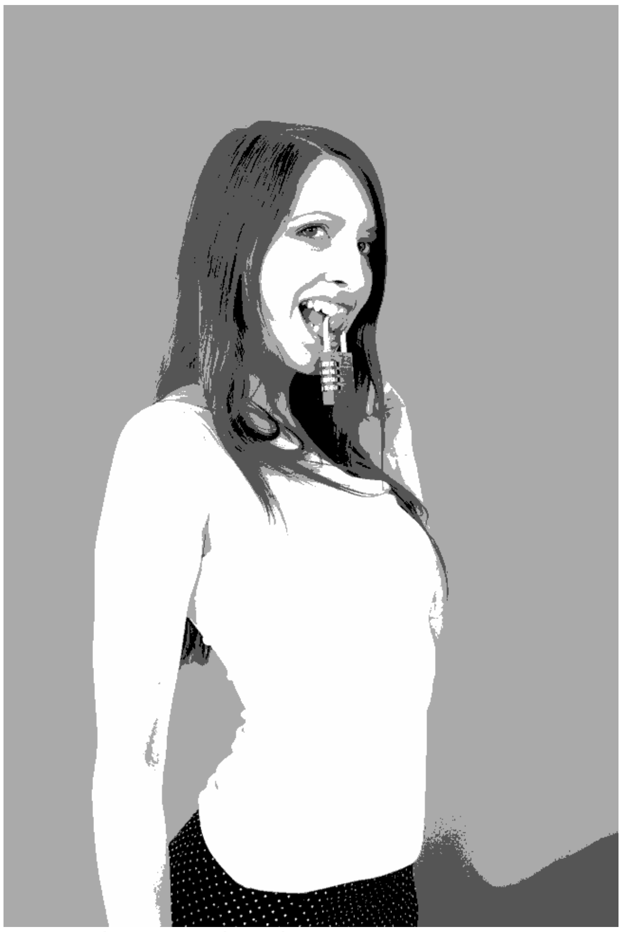
2.1 AES encrypts a bitmap with 648x972 pixels in ECB mode and without re-keying

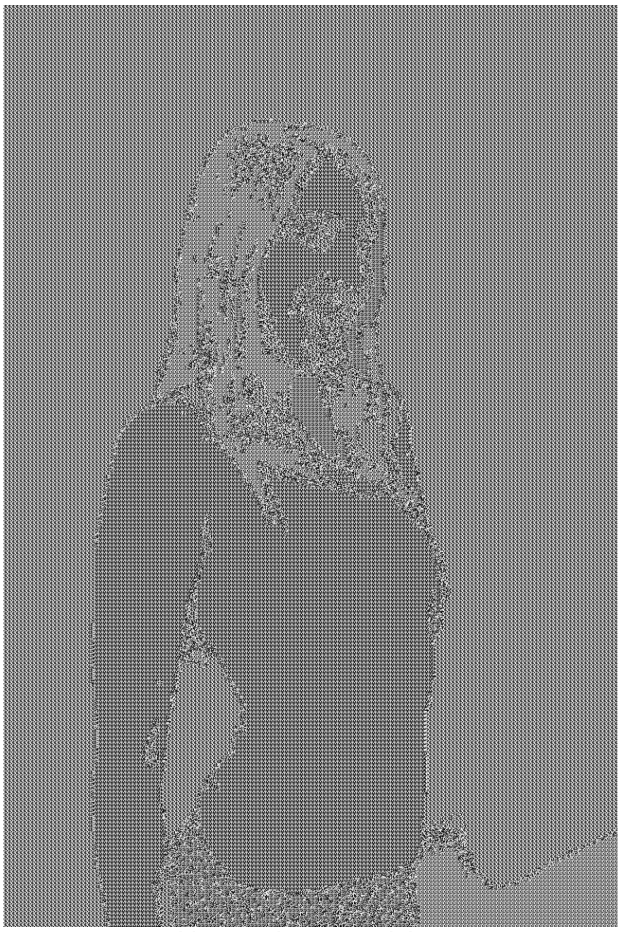
This test works with every symmetric cipher like DES, AES Rijndael, Twofish, Blowfish, Magenta, RC6, etc. Identical information is mapped by the encryption process into identical binary strings $\{0,1\}^n$ -> $\{0,1\}^n$. It is very unlikely that any commercial disk encryption application (OTFE) encrypts data this way. A lot of information leaks. We demonstrate the effect here.

Source code:

```
void test_AES()
{
	CImage image;
	HRESULT hr;
	int height,width;
	COLORREF pixel_color;
	int x,y,i,j,i_buf,count;
	uint8 data_in[16];
	uint8  key[32];
	struct aes_context ctx;
	WCHAR s[256];


	hr=image.Load(L"c:\\sample.bmp");
	if (hr==0)
	{
		height=image.GetHeight();
		width=image.GetWidth();

			// check if we can create 4x4 pixel blocks and encrypt these blocks
		if ((width%4!=0) || (height%4!=0)) return;



		for (i=0;i<32;i++) key[i]=0x00;
		aes_set_key(&ctx,key,256); // 256 bit key

		for (x=0;x<width;x=x+4)
		{
			swprintf(s,256,L"Progress: %d percent",(x*100)/width);
			m_msg.SetWindowText(s);

			for (y=0;y<height;y=y+4)
			{
					// read in 16 pixels
				count=0;
				for (i=x;i<x+4;i++)
				{
					for (j=y;j<y+4;j++)
					{
						pixel_color=image.GetPixel(i,j);


	i_buf=((int)GetRValue(pixel_color)+(int)GetGValue(pixel_color)+(int)GetBValue(pixe
l_color))/3;
						data_in[count]=(uint8)i_buf;
						count++;
					} // for (j=y;j<y+4;j++)
```

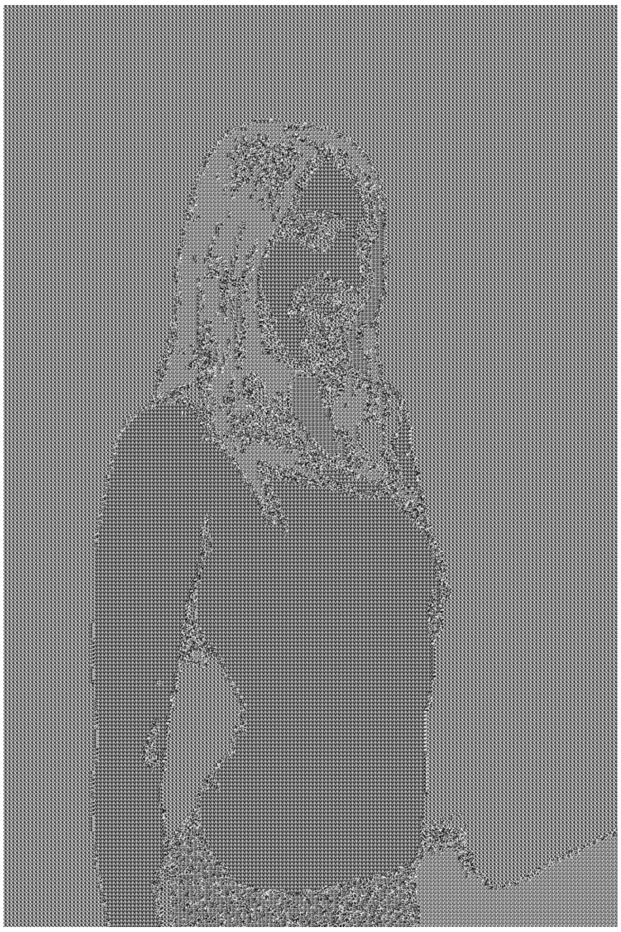
```
                        } // for (i=x;i<x+4;i++)

                        aes_encrypt(&ctx,data_in);

                              // write the 16 pixels back to the image
                        count=0;
                        for (i=x;i<x+4;i++)
                        {
                              for (j=y;j<y+4;j++)
                              {

      image.SetPixelRGB(i,j,data_in[count],data_in[count],data_in[count]);
                                    count++;
                              } // for (j=y;j<y+4;j++)
                        } // for (i=x;i<x+4;i++)


                  } // for (y=0;y<height;y=y+4)
            } // for (x=0;x<width;x=x+4)

            hr=image.Save(L"c:\\sample_encrypted.bmp");
            if (hr==0) m_msg.SetWindowText(L"Ready with AES-encrypting an image
file.");
            else m_msg.SetWindowText(L"Couldn't save image file.");
      } // if (SUCCESS(hr))
}
```

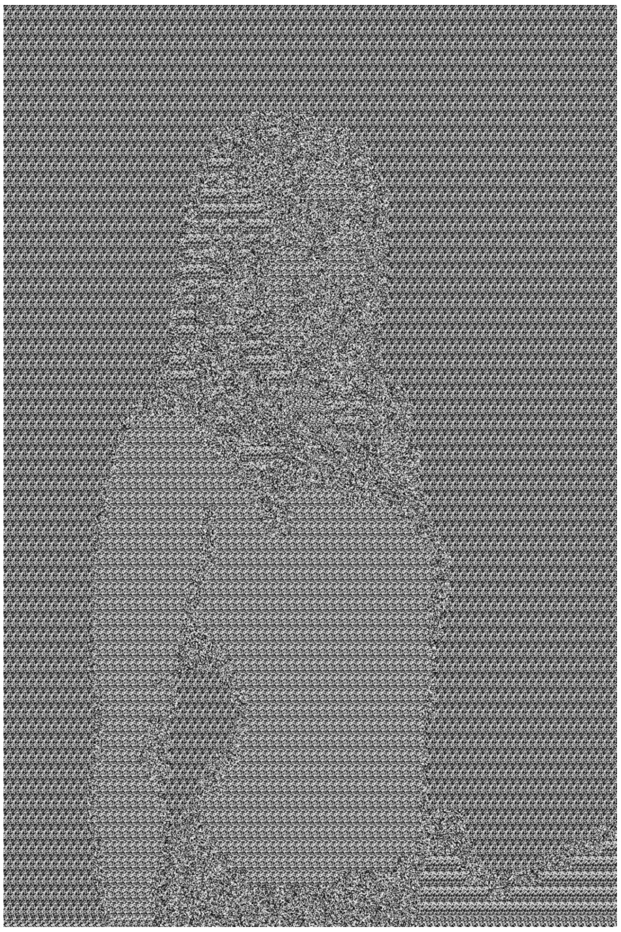

Sample image: reduced to 4 colors:


AES-encrypted image (ECB mode)

Block size alone doesn't help with ECB mode.

The two pictures below depict the AES-encrypted image and a 1024 bit PMC-encrypted image, both encrypted in ECB mode,


AES-encrypted image (ECB mode):


1024 bit PMC-encrypted image (ECB mode)

The only difference compared to the AES-encrypted image in ECB mode is the granularity of 4x4 pixels for AES compared with 16x8 pixels for PMC. This causes uneven areas like hair and face to be slightly better encrypted by the polymorphic cipher as can be seen in the direct comparison below. The sole reason for this improvement is the eight times greater block size of the 1024 bit polymorphic cipher.

2.2 AES encrypts a bitmap with 648x972 pixels in ECB mode with block count and another image with a uniform color (e.g. white=RGB(255,255,255)) as well in ECB mode with block count

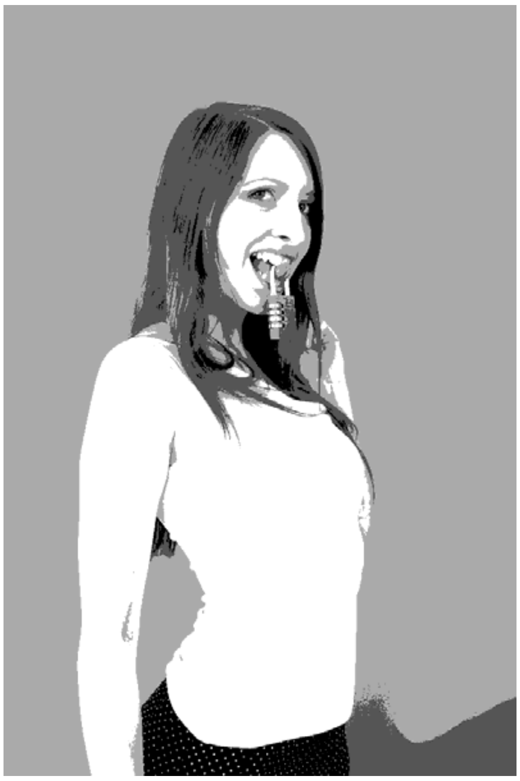
Truecrypt actually employs this method, as well as two more elaborate methods: LRW mode (AES cipher and Galois field multiplication) and XTS, which is XEX (Xor-Encrypt-Xor) - based Tweaked CodeBook mode with CipherText Stealing.
The key is initialized only once. For every 128 bit block, a bit pattern representing the block number is basically added prior to encryption (for LRW and XTS, more operations are executed, but the principle holds). The methodology works pretty well and security is very high.

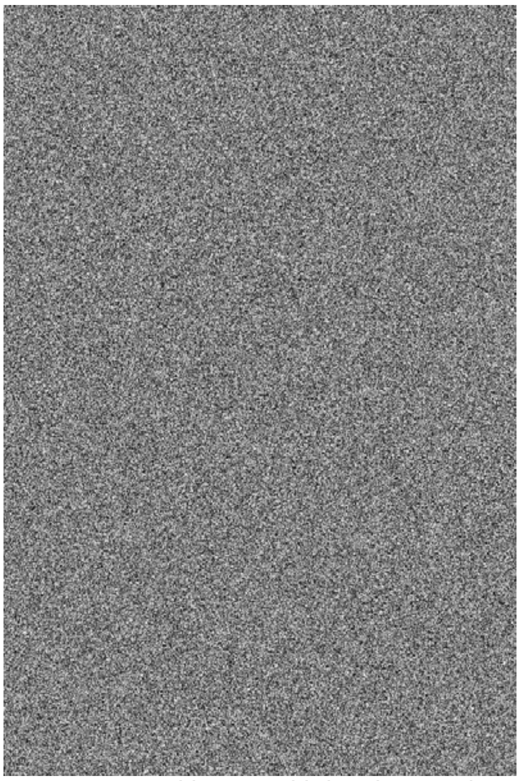
This is simulated by using this source code:

```
void test_AES()
{
        CImage image;
        HRESULT hr;
        int height,width;
        COLORREF pixel_color;
        int x,y,i,j,i_buf,count;
        uint8 data_in[16];
        uint8  key[32];
        struct aes_context ctx;
        WCHAR s[256];


        hr=image.Load(L"c:\\sample.bmp");
        if (hr==0)
        {
                height=image.GetHeight();
                width=image.GetWidth();

                // check if we can create 16x8 pixel blocks and encrypt these blocks
                if ((width%4!=0) || (height%4!=0)) return;



                for (i=0;i<32;i++) key[i]=0x00;
                aes_set_key(&ctx,key,256); // 256 bit key

                for (x=0;x<width;x=x+4)
                {
                        swprintf(s,256,L"Progress: %d percent",(x*100)/width);
                        m_msg.SetWindowText(s);

                        for (y=0;y<height;y=y+4)
                        {

                                        // plaintext is 255 = 0xff
                                for (i=0;i<16;i++) data_in[i]=0xff;


                                        // use x and y to modify plaintext
                                data_in[12]^=(uint8)x;
                                data_in[13]^=(uint8)y;
                                data_in[14]^=(uint8)(x>>8);
                                data_in[15]^=(uint8)(y>>8);


                                aes_encrypt(&ctx,data_in);

                                        // write the 16 pixels back to the image
                                count=0;
                                for (i=x;i<x+4;i++)
                                {
                                        for (j=y;j<y+4;j++)
                                        {
```

```
        image.SetPixelRGB(i,j,data_in[count],data_in[count],data_in[count]);
                                        count++;
                                } // for (j=y;j<y+4;j++)
                        } // for (i=x;i<x+4;i++)


                } // for (y=0;y<height;y=y+4)
        } // for (x=0;x<width;x=x+4)

        hr=image.Save(L"c:\\sample_encrypted.bmp");
        if (hr==0) m_msg.SetWindowText(L"Ready with AES-encrypting an image
file.");
        else m_msg.SetWindowText(L"Couldn't save image file.");
    } // if (SUCCESS(hr))
}
```

The following two pictures show plaintext and ciphertext. The ciphertext alone seems not to contain any information about the plaintext.


Sample image: reduced to 4 colors:

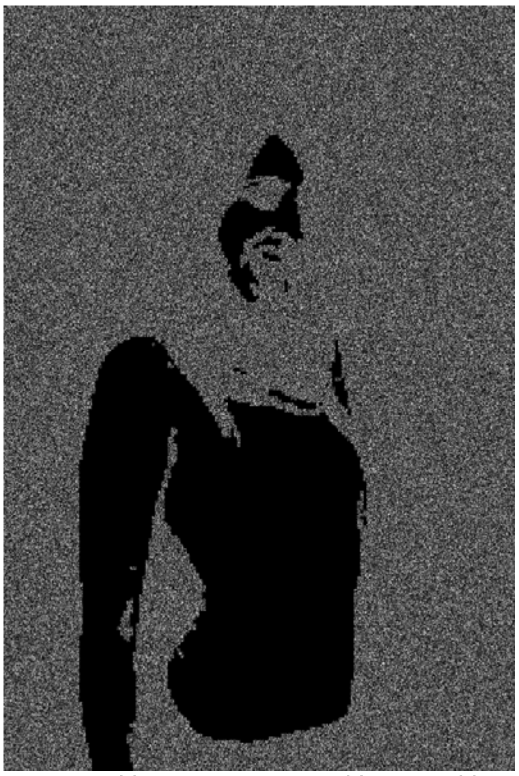

AES-encrypted image with block count incremented by 1 for each 4x4 pixel block:

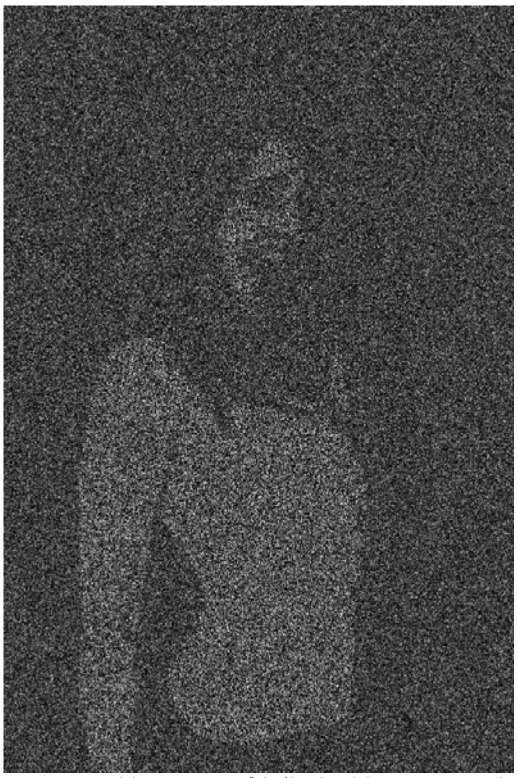
2.3 Ciphertext-only attack on backups of volume image files requiring NO knowledge of the key

If a volume file is copied and the original copy is used to encrypt data while the other copy contains known plaintext (e.g. all zeros), it is possible to simply subtract data bits with identical bit positions in the two files from each other. This attack requires NO knowledge of the key used for encryption and it applies to ECB Mode (Electronic Codebook), Counter Mode (CM), Galois/Counter Mode (GCM), LRW, XEX, XTS, as well as CBC-based modes of disk encryption applications (OTFE).

It is very easy to unveil large parts of the sample image. All that is needed is the ciphertext of the sample image and the ciphertext of an image with a uniform color. I've used white color to demonstrate the attack:

The two images below are simply created by subtracting (respectively multiplying) the color of each pixel that is located at the very same position in the two ciphertext images.


Encrypted image – encrypted image with all white pixels (subtraction)

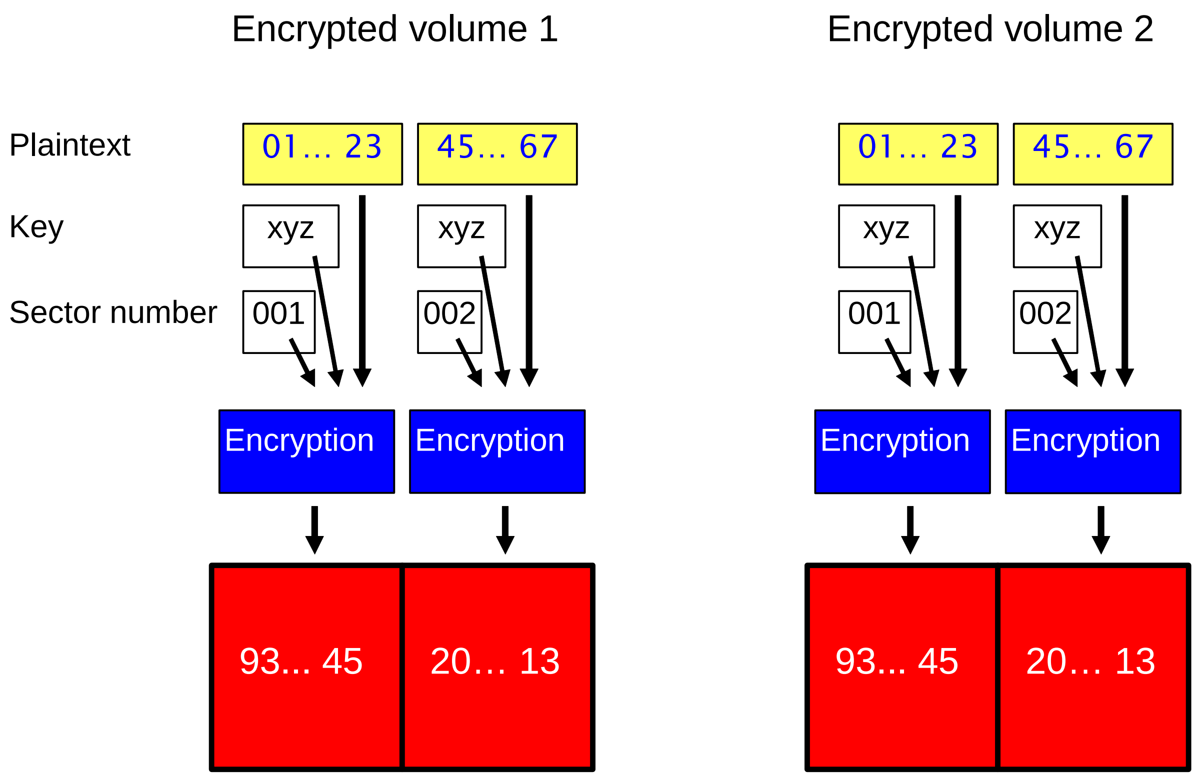

Encrypted image multiplied with encrypted image with all white pixels

The result is the logical consequence of encrypting identical information with an identical key. The result would be different if the user would have created another image file, instead of copying a volume file. When copying an encrypted volume, disk key and initialization vector information are also copied. This finally results in two identical keys that are used for both encrypted volumes.

3. Which disk encryption products are affected by this attack?

Generally all disk encryption programs that are available on the market seem to contain this security hole !!!
The attack has been proven for a number of popular and commercially available OTFE software packages.
Old versions of TurboCrypt are as well affected for data areas with identical plaintexts larger than 512 bytes.

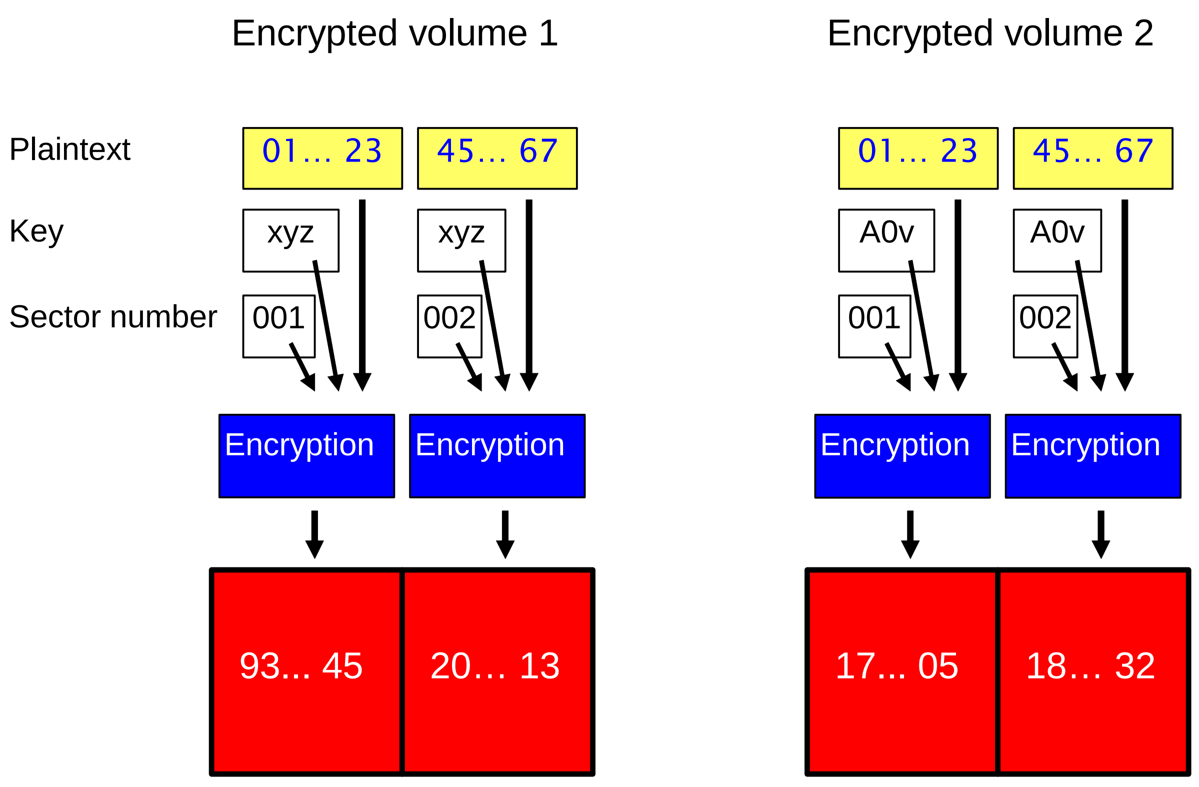
The reason why many or probably all on-the-fly encryption (OTFE) software packages are affected is easy to explain. Two or more volumes with the same encryption key that host the same data (plaintext) inevitably contain the same ciphertext. The following figure shows how encryption of sectors on a disk generally works:



On both independent encrypted volumes there shall be two identical plaintexts encrypted with an identical key.
OTFE software has no other information than key and sector number to encrypt gigabytes of data. In order not to yield always the same ciphertext when encrypting a static plaintext, sector number and the number of the block within a sector are both added to the key or this information is logically combined with the plaintext prior to the encryption process. Both methods can be applied likewise. No information about plaintext nor the key will leak at all.

The problem starts when creating a copy of volume 1. Volume 2, which is the copy of volume 1 may subsequently be used to store large pictures containing big areas with a uniform color. Volume 1 may contain data (e.g. pictures) containing blocks with the same bit pattern (= color). It is only natural that identical plaintext on volume 1 is encrypted into the very same ciphertext as on volume 2 if the data resides on sectors with the same sector number.

If the key for both volumes is identical – and one can be sure that it is for all copies of a volume file – this attack can be mounted very easily. There is nothing that can be done against this inherent weakness on the encryption level.



Most or all OTFE softwares take advantage of disk keys. Changing passwords does thus not require re-encryption of an entire image file and security does not suffer at all due to the fact that password encryption is performed using a one-time-pad. The user-selected key serves as key for the encryption of the disk key, which is a true random number.
When creating a backup of a volume image file, TurboCrypt uses a new real random key in place of the original disk key in the backup of an image file. This methodology solves the previously described security problem entirely.

5. Conclusion

A new ciphertext-only attack that can be mounted easily by anyone who has access to encrypted volumes using identical keys, has been identified and described by us. Likelihood for all OTFE (on-the-fly encryption) software packages to be susceptible to this kind of attack is close to 100%. Tests have shown that the majority of all commercially available OTFE (disk encryption) programs are susceptible to this attack.

All versions of TurboCrypt from 2008 and later contain two additional mechanisms that protect users so that security of their data will never be compromised as long as users follow the simple rule to let TurboCrypt create backups of their volume image files.

Developers of other OTFE products will most probably follow and implement similar countermeasures with time. To our knowledge is the described method free of patents and the author can confirm that he hasn't applied for protection of this intellectual property.

For more information: http://www.pmc-ciphers.com