Whitepaper

June 2017

Demystifying Full-Disk Encryption

A special case of applied cryptography

Jacob Thompson
Senior Security Analyst
Independent Security Evaluators LLC
jthompson@securityevaluators.com

Demystifying Full-Disk Encryption: A Special Case of Applied Cryptography

Jacob Thompson <jthompson@securityevaluators.com>

Independent Security Evaluators LLC
June 2017

Abstract
Transparent full-disk encryption uses techniques found almost nowhere else in cryptography, such as ESSIV and XTS-AES. Why must designers resort to building a custom cryptosystem rather than relying on standard techniques with typical security guarantees? This paper explores the constraints under which a full-disk encryption must operate, questioning the performance reasons for avoiding more standard cryptography yet finding them to hold. I introduce the reader familiar with cryptography but not the operation of disks to this problem, explain the high-level workings of ESSIV and XTS-AES, and review the attacks and limitations that these approaches face.

Background[1]

Security Guarantees of Common Cryptographic Protocols
Security practitioners commonly encounter applied cryptography as a part of standard protocols, such as Transport Layer Security (TLS) [1], Secure Shell (SSH) [2], and Internet Protocol Security (IPsec) [3], which provide encryption in transit, or Pretty Good Privacy (PGP) [4], Secure/Multipurpose Internet Mail Extensions (S/MIME) [5], and PKZIP Strong Encryption Specification [6], which provide encryption at rest. Because these protocols operate at upper network layers, within user space applications, or both, they have the flexibility to provide needed security guarantees beyond message confidentiality. Typical security assurances include (and a given protocol may provide all or only a subset of these):

- *Confidentiality*. Given a ciphertext message, only a party in possession of the correct decryption key (whether in raw form or derived from, e.g., a passphrase) may obtain the original plaintext.
- *Integrity*. Any modifications to a ciphertext message on its path from sender to receiver (or in the case of encryption at rest, during the time period at rest) can be detected. Such modifications could include random bit-flipping style corruption or more subtle attacks aimed at modifying a ciphertext such that it decrypts to a different yet still-valid plaintext [7]; regardless, the integrity assurance must cover both cases.
- *Authentication*. Two communicating parties may securely authenticate each other to ensure they are communicating with whom they intend rather than an impostor (e.g., as in a man-in-the-middle attack) even when communicating over a public, non-secure channel.

Common protocols provide "cipher suites" that adopt well-known and vetted security primitives in straightforward ways to provide these guarantees. It is typical to pair AES in CBC mode for confidentiality with HMAC-SHA256 for integrity, or alternatively, to use AES in GCM mode to provide confidentiality and integrity from a single primitive. RSA and elliptic curve cryptography (or public key cryptography in general) often provide authentication.

Given their need for versatility and flexibility, higher-layer protocols emphasize security assurances as opposed to efficiency in terms of space and time. They often use variable-length headers and complex formats like ASN.1 [8] for metadata. Each message may incur a constant space overhead (e.g., for an initialization vector) or a variable one (e.g., for padding a message to a multiple of a cipher's block size). There is no attempt to align messages with sector, page or other boundaries. Asymmetric algorithms (in particular) require a random number source for encryption/decryption operations, as in RSA blinding or Optimal Asymmetric Encryption Padding (OAEP), or they rely on probabilistic algorithms; such non-deterministic running time is inappropriate for time-critical areas such as interrupt handlers. These attributes prove to be challenges in implementing a performant full-disk encryption system.

[1] The author would like to acknowledge Ali Jad Khalil as one source of inspiration for this whitepaper.

Introduction

Implementation Challenges in Full-Disk Encryption

Software-only, transparent encryption at rest at the block device level provides some unique challenges. Hard disk reads/writes occur only in units of sectors (often 512 or 4096 bytes) [9]. To maintain performance any full-disk encryption system must maintain this sector-by-sector view of the disk, i.e., decrypting or encrypting a particular sector should require no disk activity beyond reading or writing the corresponding encrypted sector. Given the fixed size of disk sectors and the need to present the same sector size to the operating system that a plain unencrypted disk would have, the largest challenge of full-disk encryption is that there is nowhere to store any per-sector metadata.

Trade-offs in Prevalent Designs

Despite the dangers of developing custom cryptography, security researchers have judged that the restrictions surrounding full-disk encryption justify developing and using custom encryption modes catering to the limitations of disks. For example, rather than employing AES-CBC with HMAC-SHA256 or AES-GCM, which both require per-message space overhead for IVs, tweaks and authentication codes, full-disk encryption uses special encryption modes like CBC with encrypted salt-sector initialization vector (ESSIV) and XEX-based tweaked-codebook mode with ciphertext stealing (XTS-AES) [10]. These modes employ algorithms to map sector numbers to pseudorandom IVs to avoid storing them. They also jettison the integrity protection provided by trusted primitives like HMAC or GCM and instead provide no (in the case of CBC with ESSIV) or little (in the case of XTS-AES) integrity assurance.

Overview

The purpose of this whitepaper is to (1) review, through observation and simulation, the reasons why naïve attempts to adapt conventional cipher suites like AES-CBC with HMAC-SHA256 to full-disk encryption fail and (2) to review how ESSIV and XTS-AES work at a high level, consider their limitations, and provide insight into full-disk encryption products that use these modes. The target audience is security professionals with moderate experience with cryptography, including familiarity with cryptographic primitives, the operation of block ciphers, and common security failures involving them.

Performance Failure of the Naïve Approach

At their essence, full-disk encryption modes like ESSIV and XTS-AES involve trade-offs between best security practices, space, and performance. For example, they use unique per sector yet fixed IVs or tweaks, and they provide no cryptographically secure integrity protection. In exchange, volumes protected using full-disk encryption provide nearly 100% of their unencrypted capacities, and these modes operate quickly enough to be used to protect an entire file system or swap area. Suppose one is not so willing to relinquish standard designs without the data to prove they are not viable. To evaluate the trade-offs of ESSIV and XTS-AES, I first look deeper into the gains involved through using ESSIV versus a naïve attempt to adapt standard AES-CBC with HMAC-SHA256 to a full-disk encryption system.

Consider a transparent full-disk encryption system that, instead of making storage trade-offs, carves each 512-byte physical disk sector into a 16-byte initialization vector, a 464-byte ciphertext, and a 32-byte HMAC value. This allows the system to employ AES in CBC mode without padding, using unique IVs generated each time a sector is rewritten. It also provides cryptographically secure integrity assurance via the HMAC. Given the security of this naïve full-disk encryption system, now consider its performance. Such an encryption scheme would present a virtual block device consisting of 464-byte sectors to the file system and higher layers of the operating system. Whether such a device could work in practice is another issue; for example, Linux requires [11] filesystem blocks to be a power-of-two multiple of the sector size smaller than the page size (clearly, for 464-byte sectors, no such block size exists) but this experiment suffices for a brief experiment to evaluate performance.

Space Performance

Measuring the space cost of the naïve encryption system is easy. Consider a 1000 GB[2] disk. When used directly, such a disk has a capacity of 1000 GB, divided into 1,953,125,000 512-byte sectors. Now suppose the naïve encryption system overlays the device. To the filesystem and higher layers, the disk now consists of 1,953,125,000 sectors of 464 bytes. Then the disk has 906.25 GB of usable space for ciphertext, for a loss of 9.4%. This compounds that imposed at the file system and higher layers (e.g., due to metadata overhead, fragmentation and cluster size).

[2] Where 1 GB = 1,000,000,000 bytes, consistent with industry practice for disks.

Time Performance
To compare the time performance of the naïve encryption system versus an established one, I developed two sets of encryption/decryption programs: one for the naïve system and one for ESSIV. These are given in the appendix. Then, I tested the time taken for both approaches to decrypt 1 GB of data on a Linux system with two 2 TB mechanical disks in a RAID 1 configuration. The results are shown in Table 1.

While results may vary with performance optimizations, user space vs. kernel space (in which a real full disk encryption system would operate), solid-state disks, and so on, this is adequate for a basic comparison from which I can draw conclusions. The naïve approach is more than three times slower. While the HMAC check (which ESSIV does not provide) is partly to blame, I found that even with HMAC verification removed, the naïve approach still takes more than twice as long as ESSIV to decrypt the file. Beyond this, the main difference is that the naïve decryptor must perform I/O reads in increments of 512 bytes and I/O writes in increments of 464 bytes (due to the overhead of the IV and HMAC value) while the ESSIV decryptor performs both reads and writes in 512-byte increments. Thus the ESSIV approach performs all I/O in units of sectors while the naïve one does not. Clearly there is substantial benefit in doing so.

Approach	Time (real)	Time (user)	Time (system)
Naïve	21.221 s	20.007 s	1.677 s
ESSIV	5.539 s	4.301 s	1.243 s

Table 1. Time comparison for the naïve and ESSIV approaches to decrypt a 1 GB file.

ESSIV
Encrypted salt-sector initialization vector (ESSIV) full-disk encryption is a clever way to "salvage" CBC mode to operate in transparent full-disk encryption, where storing sector IVs is not feasible [10]. ESSIV is used by older versions of the Linux dm-crypt driver. Microsoft BitLocker on Windows 8.1 and earlier uses Elephant diffusion [12], similar in approach to ESSIV in that it "salvages" CBC mode, but in a more sophisticated way that better resists offline tampering as in the techniques I describe below.

Operation
ESSIV reconstructs a sector's IV on demand as follows, where *sector* is the 128-bit sector number, K is the data encryption key, E is the AES-256 encryption function, and H is a hashing algorithm such as SHA-256:

$$salt = H(K)$$
$$\mathrm{IV}(sector) = E_{salt}(sector)$$

The sector is then encrypted in CBC mode as usual using the per-sector IV generated above, and the data encryption key. There is no cryptographic integrity check. Because AES-CBC acts as a block cipher, it exhibits the avalanche effect in that inverting the value of any one bit in a block of ciphertext will, with high probability, flip 50% of the bits in the corresponding plaintext block. However, an attacker may control bits in the *next* block of plaintext by inverting bits in the previous block of ciphertext, albeit with the "corruption" of the tampered block of plaintext. Further, an attacker can move, duplicate, or delete ciphertext blocks. Targeted attacks against these properties of AES-CBC may leverage knowledge about the layout of the plaintext could have security-relevant consequences against AES-CBC when used in full-disk encryption.

Attack
Suppose an adversary has temporary, offline access to an ESSIV-encrypted volume, but never has access to the key. Could the attacker have a meaningful impact on the affected user's security? Yes—and not in the obvious way of reconstructing or guessing the key.

Instead, consider how the attacker could modify an encrypted executable to change its behavior to benefit the attacker later when the affected user boots and uses the encrypted volume. For simplicity's sake, consider the Python script given in Figure 1. As shown in Figure 2, this script is carefully formatted so that the text `12345` on line 4 lies entirely within one block (bytes 64-79) when this script is encrypted with AES-CBC, while the preceding block (bytes 48-63) lies entirely within an end-of-line comment. If an attacker has offline access to an AES-CBC-encrypted copy of this

script, the attacker may freely flip bits in the ciphertext block holding the text "`do the calculat`" to change the value of the 12345 constant on the next line. Figure 3 shows a sample result when such modified ciphertext is decrypted.

```
1: #!/usr/bin/python2.7
2: # -*- coding: latin-1 -*-
3: # do the calculation
4: x = 12345 + 67890
5: print x
```

Figure 1.Using the properties of CBC mode encryption, an attacker may change the behavior of an encrypted copy of this script without knowing the key.

```
0000000 23 21 2f 75 73 72 2f 62 69 6e 2f 70 79 74 68 6f  >#!/usr/bin/pytho<
0000016 6e 32 2e 37 0a 23 20 2d 2a 2d 20 63 6f 64 69 6e  >n2.7.# -*- codin<
0000032 67 3a 20 6c 61 74 69 6e 2d 31 20 2d 2a 2d 0a 23  >g: latin-1 -*-.#<
0000048 20 64 6f 20 74 68 65 20 63 61 6c 63 75 6c 61 74  > do the calculat<
0000064 69 6f 6e 0a 78 20 3d 20 31 32 33 34 35 20 2b 20  >ion.x = 12345 + <
0000080 36 37 38 39 30 0a 70 72 69 6e 74 20 78 0a        >67890.print x.<
```

Figure 2. When divided into 16-byte blocks, the fourth plaintext block (bytes 48-63) holds text within an end-of-line comment.

```
1: #!/usr/bin/python2.7
2: # -*- coding: latin-1 -*-
3: #ÊBÁË)PÂJ|èñ¤Úion
4: x = 23456 + 67890
5: print x
```

Figure 3. Modifying the ciphertext block holding the encrypted end-of-line comment corrupts the comment, but allows the attacker to control the next block bit-by-bit to change "12345" to read "23456."

Of course, in a real attack the adversary would likely target native machine code rather than a script and would need to be much more precise, but the same principles apply; see [13] for a detailed example. This is not the only way that data encrypted in CBC mode may be usefully tampered with due to the lack of integrity protection. An attacker may move, delete, or duplicate ciphertext blocks to varying effects; for an example, see [7].

ESSIV's inherent malleability through the direct use of CBC mode encryption, and the fact that Microsoft's Elephant diffuser is not sanctioned by standards bodies such as the National Institute of Standards and Technology (NIST) led to the supersession of these techniques by XTS-AES as the state-of-the-art in current full-disk encryption systems.

XTS-AES

XEX-based tweaked-codebook mode with ciphertext stealing (XTS-AES) is the successor industry standard to ESSIV, Elephant diffusion, and similar schemes that build on CBC mode encryption. Standardized by the IEEE [14] and NIST [15], XTS-AES is used by modern full-disk encryption systems, including LUKS, TrueCrypt, FileVault, and Microsoft BitLocker for Windows 10. While not providing cryptographically secure integrity protection, XTS-AES does aim to limit an attacker's tampering to only the ability to corrupt (via the avalanche effect) a single 16-byte block, while avoiding the bit-by-bit control over the next block and the ability to cut-and-paste blocks that arise in CBC.

Operation

XTS-AES encryption incorporates two AES keys, K_1 and K_2. K_1 is used for encrypting the sector number to compute per-block tweak values, while K_2 encrypts the actual data [14]. User-facing tools may simplify this to the seemingly impossible "512-bit AES" but the division of such a key into 256-bit K_1 and K_2 is the explanation.

XTS-AES is too complex to explain here other than at a high level. XTS-AES operates on a single sector at a time. First, XTS-AES divides the sector into a sequence of 128-bit plaintext blocks. For each plaintext block, the algorithm encrypts the sector number using K_1, and performs a Galois field multiplication with a polynomial that depends on the offset of the block within the sector. The result of this computation is a *tweak* that is unique to each sector-block pair. Second, the algorithm encrypts the exclusive-or of the plaintext block and the tweak using K_2. The exclusive-or of this output and the tweak is then computed to obtain a block of XTS-AES ciphertext to write to the correct position in the disk sector. If the sector size is not a multiple of the AES block size, ciphertext stealing is performed to ensure that the data do not expand after encryption.

Limitations

XTS-AES is designed to provide the best system given the trade-offs imposed by transparent, software full-disk encryption; it is not perfect. While the designers of XTS-AES strive to prevent bit-by-bit modification attacks such as the one demonstrated in ESSIV, still, XTS-AES incorporates no cryptographically secure integrity check. It only frustrates attackers by limiting their tampering ability to randomly corrupting 16-byte units of data on the disk, and under the right conditions this may still be leveraged to modify the behavior of executables.

There are other obvious limitations; most obviously, if a sector is written with some data, then modified, and later rewritten with the original data, it will contain the original ciphertext. That is, an attacker who has offline access to an encrypted disk at two different times could quickly determine which sectors have changed and draw conclusions from that observation.

Conclusion

As this whitepaper shows, the typical modular "cipher suite" approach to provide developers with easy-to-use cryptographic capabilities is not suitable for transparent full-disk encryption for space and performance reasons. The current state-of-the-art full-disk encryption scheme, with the support of IEEE and NIST, is XTS-AES. The selection of XTS-AES was not without controversy. Microsoft's Elephant diffuser provided the property that any tampering with a sector would corrupt the entire sector [12]; XTS-AES does not do so, limiting this anti-tampering property only to each 16-byte block. Still, Microsoft abandoned the Elephant diffuser in favor of XTS-AES for performance and NIST compliance reasons [16].

Beyond gaining a deeper insight into the inner workings of full-disk encryption, if there is one takeaway for the target audience, it is that XTS-AES is a "least worst" compromise solution to the problem of full-disk encryption, and should not be used in any other application where the limitations of transparent encryption do not apply. Ptacek provides additional insight [17] into that decision.

Of course, transparent full-disk encryption need not be implemented with standard hardware, nor must it be implemented totally in software. For example, hard drive manufacturers could introduce drives with 560-byte sectors, leaving room for a 16-byte IV, 512 bytes of data, and a 32-byte HMAC-SHA256 value. With this design encryption and authentication could be moved from the operating system to the drive firmware. Alternatively, hardware vendors could introduce a "mini" drive composed of the same number of sectors as a larger one yet possibly shrunk to 48 bytes; such a drive could be used to read and write IVs and HMAC values in parallel with the data on a separate drive.

This review of full-disk encryption has two main takeaways: (1) the structure and capabilities of disks limit the ability to apply standard cryptographic approaches to full-disk encryption, and (2) the algorithms designed to suit full-disk encryption incorporate inherent trade-offs and are not suited to any other purpose. Given these concepts full-disk encryption may be understood and the associated algorithms defended to the extent possible.

References

[1] T. Dierks and E. Rescorla, The Transport Layer Security (TLS) Protocol, RFC 5246, 2008.

[2] T. Ylonen, The Secure Shell (SSH) Transport Layer Protocol, RFC 4253, 2006.

[3] S. Kent and K. Seo, Security Architecture for the Internet Protocol, RFC 4301, 2005.

[4] J. Callas et al, OpenPGP Message Format, RFC 4880, 2007.

[5] B. Ramsdell and S. Turner, Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification, RFC 5751, 2010.

[6] PKWARE Inc., "XVI. Strong Encryption Specification," in *APPNOTE.TXT - .ZIP File Format Specification Version 6.3.0*, 2006.

[7] J. Thompson, "Tales of Fails and Tools for Message Integrity," BSides DC, 23 October 2016. [Online]. Available: http://www.securityevaluators.com/knowledge/presentations/integrity.pdf.

[8] International Telecommunication Union, Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), Rec. ITU-T X.690, 2015.

[9] Wikipedia, "Disk sector," 20 November 2016. [Online]. Available: https://en.wikipedia.org/wiki/Disk_sector.

[10] Wikipedia, "Disk encryption theory," 14 December 2016. [Online]. Available: https://en.wikipedia.org/wiki/Disk_encryption_theory.

[11] R. Love, "The Block I/O Layer," in *Linux Kernel Development*, Indianapolis, IN, Pearson Education, 2005, p. 236.

[12] N. Ferguson, "AES-CBC + Elephant diffuser," August 2006. [Online]. Available: http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/BitLockerCipher200608.pdf.

[13] J. Lell, "Practical malleability attack against CBC-Encrypted LUKS partitions," 22 December 2013. [Online]. Available: http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/.

[14] Institute of Electrical and Electronics Engineers, "IEEE P1619 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices," May 2007. [Online]. Available: http://grouper.ieee.org/groups/1619/email/pdf00086.pdf.

[15] National Institute of Standards and Technology, Special Publication 800-38E, 2010.

[16] M. Lee, "Microsoft Gives Details About Its Controversial Disk Encryption," The Intercept, 4 June 2015. [Online]. Available: https://theintercept.com/2015/06/04/microsoft-disk-encryption/.

[17] T. Ptacek, "You Don't Want XTS," 30 April 2004. [Online]. Available: https://sockpuppet.org/blog/2014/04/30/you-dont-want-xts/.

Appendix

Naïve Encryption Simulation

```
#include <stdio.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <openssl/hmac.h>
#include <openssl/rand.h>
#include <openssl/sha.h>
#include <string.h>

#define EKEY_LEN 16
static unsigned char ekey[EKEY_LEN] = { 0x47, 0x8c, 0xb4, 0x73,
                                        0xa7, 0x0c, 0x23, 0x0b,
                                        0x8f, 0x01, 0x2e, 0x89,
                                        0x2e, 0x3a, 0x73, 0x24 };

#define HKEY_LEN 16
static unsigned char hkey[HKEY_LEN] = { 0x19, 0x7b, 0x9f, 0xb4,
                                        0x52, 0x6e, 0x11, 0xde,
                                        0x29, 0xb7, 0xf8, 0x7e,
                                        0xa1, 0xb5, 0x38, 0x3e };

#define CTEXT_LEN (512 - AES_BLOCK_SIZE - SHA256_DIGEST_LENGTH)
#define REC_LEN 512
struct rec
{
   unsigned char iv[AES_BLOCK_SIZE];
   unsigned char ctext[CTEXT_LEN];
   unsigned char hmac[SHA256_DIGEST_LENGTH];
};

static void xprint (unsigned char *buf, unsigned int len)
{
   unsigned i;
   for (i = 0; i < len; ++i)
```

```
      fprintf (stderr, "%02x", buf[i]);
   fprintf (stderr, "\n");
}
int main (void)
{
   struct rec record;
   unsigned char buf[CTEXT_LEN];
   unsigned char iv[AES_BLOCK_SIZE];
   AES_KEY akey;
   HMAC_CTX hctx;
   unsigned int mdlen;

   /* initialization */
   HMAC_CTX_init (&hctx);
   AES_set_encrypt_key (ekey, EKEY_LEN * 8, &akey);

   while (fread (buf, CTEXT_LEN, 1, stdin) == 1)
   {
      RAND_pseudo_bytes (record.iv, AES_BLOCK_SIZE);
      memcpy (iv, record.iv, AES_BLOCK_SIZE);
      AES_cbc_encrypt (buf, record.ctext, CTEXT_LEN, &akey, iv, 1);
      HMAC_Init (&hctx, hkey, HKEY_LEN, EVP_sha256 ());
      HMAC_Update (&hctx, record.iv, AES_BLOCK_SIZE);
      HMAC_Update (&hctx, record.ctext, CTEXT_LEN);
      mdlen = SHA256_DIGEST_LENGTH;
      HMAC_Final (&hctx, record.hmac, &mdlen);
      fwrite (&record, REC_LEN, 1, stdout);
   }

   /* cleanup */
   HMAC_CTX_cleanup (&hctx);
   return 0;

}
```

ESSIV Encryption Simulation

```
#include <stdio.h>
#include <arpa/inet.h>
#include <openssl/aes.h>
#include <openssl/evp.h>
#include <openssl/hmac.h>
#include <openssl/rand.h>
#include <openssl/sha.h>
#include <openssl/md5.h>
#include <string.h>

#define EKEY_LEN 16
static unsigned char ekey[EKEY_LEN] = { 0x47, 0x8c, 0xb4, 0x73,
                                        0xa7, 0x0c, 0x23, 0x0b,
                                        0x8f, 0x01, 0x2e, 0x89,
                                        0x2e, 0x3a, 0x73, 0x24 };
static AES_KEY akey;

#define IKEY_LEN 16
static unsigned char ikey[IKEY_LEN];
static AES_KEY aikey;

#define CTEXT_LEN 512

static unsigned char *geniv (uint32_t sn)
{
   static unsigned char iv[AES_BLOCK_SIZE];
   unsigned char buf[AES_BLOCK_SIZE];

   memset (buf, 0, AES_BLOCK_SIZE);
   sn = htonl (sn);
   memcpy (buf + (AES_BLOCK_SIZE - sizeof sn), &sn, sizeof sn);
   AES_encrypt (buf, iv, &aikey);
   return iv;
```



```
}

int main (void)
{
   unsigned char ptext[CTEXT_LEN];
   unsigned char ctext[CTEXT_LEN];
   unsigned char *iv;
   unsigned int sn = 0;

   /* initialization */
   MD5 (ekey, EKEY_LEN, ikey);
   AES_set_encrypt_key (ekey, EKEY_LEN * 8, &akey);
   AES_set_encrypt_key (ikey, IKEY_LEN * 8, &aikey);

   while (fread (ptext, CTEXT_LEN, 1, stdin) == 1)
   {
      iv = geniv (sn++);
      AES_cbc_encrypt (ptext, ctext, CTEXT_LEN, &akey, iv, AES_ENCRYPT);
      fwrite (ctext, CTEXT_LEN, 1, stdout);
   }

   return 0;

}
```

