There is a new Advanced Encryption Standard... Now What?

John Black
University of Nevada at Reno (USA)
jrb@cs.unr.edu
www.cs.unr.edu/~jrb

1 Some Background

In 1976 the National Bureau of Standards, now the National Institute Standards and Technology (NIST), adopted an algorithm called the "Data Encryption Standard" (DES) as a federal information-processing standard for protecting information. DES works like this: you choose a random 56-bit key K, and then compute the function $C = \mathrm{DES}_K(P)$ for a 64-bit "plaintext" P. The "ciphertext," C, is also 64-bits long. The idea is that P is transformed into some "random" string C so that P is hidden from users who lack a copy of K. (This is a very informal definition; cryptographers use a much more precise definition which we'll skip here.) An algorithm of this type is called a "block cipher" because its inputs and outputs are of the same fixed size; this size is called the "block size" and for DES it is 64-bits. Block ciphers like DES can be used as a building block for more complex tasks like encrypting and authenticating arbitrary messages, for example.

Since the DES key is 56-bits, there are 2^{56} possible keys. In 1976 this was a lot of keys, but these days one can build a machine for a few hundred thousand dollars which will exhaustively search through all 2^{56} keys in just a few hours. Therefore we have long been in need of a replacement for DES. NIST, aware of this need, launched a process in 1997 aimed at adopting a new block cipher, dubbed the "Advanced Encryption Standard" (AES) which would supplant DES as a federal standard. The AES would have a block size of 128-bits, longer keys (NIST specified that the algorithm must accept keys of 128, 192, and 256 bits), resist all known attacks, be efficient in hardware and software, be patent-free, and run well on both 8-bit architectures (like some smartcards) or on 32-bit commodity processors. Submissions were received from RSA, IBM, and other groups. After years of scrutiny by the cryptographic community, an algorithm called Rijndael (pronounced "rhine-doll") was signed into law as the new AES. (See `www.nist.gov/aes` for more information.)

Rijndael was invented by two young Belgian cryptographers, Joan Daemen and Vincent Rijmen. To some, Rijndael seemed a risky choice. Why? All other submissions to NIST used the traditional "Feistel Network" structure (or variations of it), which was well-known and trusted. DES had also used this structure. But Rijndael was an evolutionary descendant of "Square" (see DDJ 10/99??) which is based on a very different idea: the algorithm treats the input as a matrix and it transforms this matrix by shifting rows, mixing columns, and renaming bytes via a table, all in some prescribed order. This novel design gave Rijndael several advantages over many of its competitors: it is simple, efficient, and quite elegant. (And, some would add, unpronounceable.)

2 Using AES: Information Privacy

Now that we have AES, what do we do with it? Well, since it's a block cipher, we can build higher-level primitives using it, just as we did with DES. Let's start with a simple example. If we want to hide 128 bits of information stored in some file, we might use this approach: choose a random 128-bit key K (AES allows 192-bit and 256-bit keys as well, but we'll use 128-bit keys). Next compute $C = \mathrm{AES}_K(P)$ by calling the AES routine with key K and your 128-bit plaintext P and capturing the return value in a 128-bit ciphertext C. To decrypt we run AES "backwards" with the same key K, handing it C and we recover M.

This is a very simple usage of AES and it is already possible to go astray: for example if your random-number generator is not very good, the key K may be predictable to some adversary who is trying to break your encryption (a famous case of this occurred in 1995 when Ian Goldberg and David Wagner broke the encryption on Netscape's browser by attacking the weak number-generation method used; see the 1/96 issue of DDJ at `http://www.ddj.com/articles/1996/9601/9601h/9601h.htm`). And what do you do next with P and C? You could erase P from your hard disk and leave C as the only record, but probably your operating system does not "zero out" the disk space after P is erased, so an "unerase" program or disk analysis tool might be able to recover P without even looking at C! Or perhaps you assume that your computer is secure, but you wish to share P with someone by sending C over a (potentially insecure) network channel. But this opens an entirely new can of worms: how do you get your randomly-chosen K to the person on the other end of the network?

We're going to ignore all of these issues. We'll assume that some good random-number generator is used to generate K (see `http://www.cs.berkeley.edu/~daw/rnd/index.html` for a collection of recommended methods), and we'll also assume that K is held by both parties on opposite ends of a network connection. Readers familiar with public-key cryptography and certificates know how this is typically done in practice, but we won't discuss it here. (Of course if you are just encrypting items on your disk, you have no key-management worries other than hiding your key in a safe place!)

So let's say you've used the method above to encipher P and have obtained C. Is this what cryptographers mean when they say "encryption?" Well, no. The word "encryption" carries with it a very strong connotation among cryptographers, and though there are several definitions currently out there, our method satifies none of them. One obvious drawback to our method is that P must be 128 bits, but in the real world our messages will often have other lengths. A more subtle drawback is that if we "encrypt" P twice with the same K, we get the same C each time. This is bad because an adversary who sees C transmitted twice over the network will know that the message P is being repeated. Despite the fact he may not be able to determine what P is, we still dislike the fact he can detect its retransmission.

We will fix each of these problems one at a time; let's start with the first one.

ECB MODE. So how do we encrypt messages whose lengths are not 128-bits? In this case one typically uses a "Mode of Operation," an algorithm which uses a block cipher as a building block. (See `www.nist.gov/modes`.)

NIST blessed several modes of operation back when DES was adopted. The simplest is the Electronic Code Book (ECB), which is the straightforward extension of the method introduced above: take a message M of *any* length; pad M by adding a '1' bit and then as many '0' bits as needed to bring M's length up to the next multiple of 128 bits; break M into 128-bit blocks $M_1, M_2, \cdots M_m$; then encrypt each block by $C_i = \mathrm{AES}_K(M_i)$ for each $i \in [1, m]$. Finally, output $C_1, \cdots, C_m$ as the ciphertext (see Figure 1). To decrypt, we just reverse this process.

This method doesn't address the retransmission problem mentioned above, but does it offer good security? It must, right? NIST endorsed this mode as a standard, so it must be good!

In actuality, most researchers regard ECB as a poor way to encrypt because of the retransmission problem. Let's examine an example of how this problem might be exploited. Assume we are using ECB mode

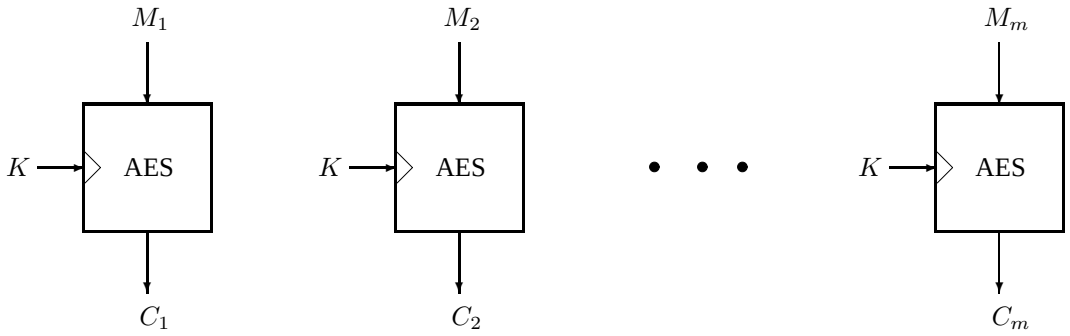

Figure 1: In ECB mode, we encipher each block of the plaintext message M independently using the same key K.

encryption with 128 bit blocks, transmitting several messages with the same AES key K. Let's say we have a message "Meet her at the Albuquerque Fair." Then if we were to encrypt with ECB it might look like this:

```
   Message:  M e e t   h e r   a t   t h e       A l b u q u e r q u e   F a i r
     ASCII: 4d6565742068657220617420746865      416c627571756572717565 2046616972
Ciphertext: 8293f911ab100910ec6643acf1f970      91ff6a8194d5d3c27ebb182990a1b451
```

Now, in a later message which begins with "Albuquerque Fair...", we would recognize the ciphertext as a match with an earlier ciphertext block we had seen. This is more information than we would like to give away! (Normally we would like to divulge only the length of the plaintext and the fact that we are sending a message, nothing more. ECB mode gives away much more!)

In fact, this retransmission problem is exploited by people attempting the ubiquitous "cryptogram" puzzles often found in Sunday newspapers: the most commonly-found letter in English text is "E," and since every "E" is translated to the same ciphertext character, puzzle solvers can exploit this. (Of course in the newspaper's block cipher the block size is not 128 bits but more like 5 bits so frequency attacks are far more feasible.)

So let's not use ECB for encryption.

IF NOT ECB, THEN WHAT? In September 2000, NIST held a workshop to decide which Modes of Operation should be included in a new federal standard. In early 2002, NIST issued a draft standard which included ECB mode, despite suggestions to drop it. But another mode, never before standardized by NIST, appeared in the draft as well: "Counter Mode" (CTR Mode). CTR mode encryption has been known for a long time: first formally introduced by Diffie and Hellman in 1979, it's based on ideas much older. Here's how we might implement it: as part of our set-up, we choose some counter value ctr where $0 \leq ctr < 2^{128}$. Then, to encrypt a message M, we xor M with the first $|M|$ bits of $\text{AES}_K(\langle ctr \rangle) \parallel \text{AES}_K(\langle ctr+1 \rangle) \parallel \text{AES}_K(\langle ctr+2 \rangle) \cdots$. (Here $|M|$ means the length of M in bits and $A \parallel B$ means the concatenation of strings A and B.) The ciphertext becomes (ctr, C). In other words, the ctr value is not kept secret: we allow anyone to read it along with C.

To decrypt ciphertext (ctr, C) we compute the xor of C and the first $|C|$ bits of the same string $\text{AES}_K(\langle ctr \rangle) \parallel \text{AES}_K(\langle ctr+1 \rangle) \parallel \text{AES}_K(\langle ctr+2 \rangle) \cdots$. Often we refer to C itself, rather than (ctr, C), as the ciphertext. Note that decryption is the same as encryption with the plaintext M and ciphertext C interchanged (see Figure 2).

If we just encrypted i blocks with a counter value ctr, we would normally use the value $ctr + i$ as the counter value for our next encryption so that a ctr value is not repeated.

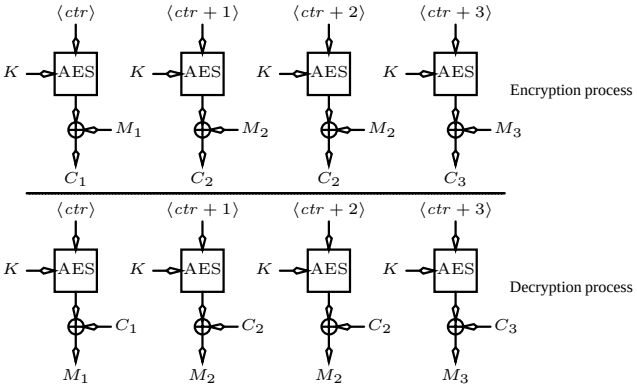

Figure 2: Encryption and decryption process in counter mode.

CTR mode encryption satisfies the requirements for strong cryptography assuming it is properly implemented, using a strong block cipher such as AES. It handles messages of arbitrary length, and solves the retransmission problem: two messages containing the same block will always result in different ciphertexts. The biggest potential for error when using CTR mode is in re-using the counter: security quickly vanishes when counter values are re-used and it is thus imperative to avoid this. This applies to all ctr values for each block of the message; in other words, we should *never* submit the same input to AES throughout the life of our AES key K. (Once K is refreshed to a new value it is, of course, fine to re-use counter values used under the old K. But changing K is often expensive, so we would like to re-use it many times while just changing ctr for each message.)

Although the security of CTR mode can be rigorously demonstrated, doing so would take a while. Intuitively, enciphering the counter values produces "random looking" outputs. It is well-known that if we could get truly random strings as outputs, then xoring these strings with any message would achieve *perfect* security. Of course the pseudorandom strings we get from AES in this context are not truly random, but if AES is a good block cipher then they are "pretty close" to random.

Beyond the fact that it is provably-secure, CTR mode enjoys several other advantages, one being that it is very efficient: in a parallel environment we can compute the many AES invocations simultaneously since the input-blocks do not require knowing the output from any other AES invocation. Also, if we know the counter value ranges in advance, we can compute the AES values in anticipation of encrypting or decrypting messages and thus save time later on. And you probably noticed that we never needed to decipher with AES even when decrypting C (unlike with ECB mode). This means we don't need to implement the "backwards" mode of AES which means less code (and less silicon for hardware implementations).

CTR mode is a fast and secure method to encrypt, and looks likely to soon be ratified by NIST as a new standard mode.

3 Authentication

As several prominent security experts have indicated, encryption without authentication is an incomplete solution to the information-security problem. So let's see how we can add authentication to our system.

First, what is "authentication?" Let's say that you are connected to your on-line broker's web site and you indicate that you'd like to transfer $1,000 from account #108331 to account #109991. Let's further assume that the web connection is using CTR mode encryption as described above. Well, we know that no eavesdropper will be able to understand what you're doing: after all, it's encrypted! But what if an adversary

attempts to *alter* the ciphertext as it passes through the internet? This could be very bad for you, especially if he succeeds in changing the destination account number to something he can exploit!

For example, say the adversary owns account #109944. He knows that the destination account number is always sent at the end of the final block of the transmission. Let's say he sees the string `...af54` at the end of the final block, and he knows you're attempting to transfer money to account #109991, which ends in ASCII `...3931`. He knows that `af54` is the xor of `3931` with the key material, so if he just transmits `af54` $\oplus$ `3931` $\oplus$ `3434` = `a251`, he will effectively change your account number to his, and thus redirect your $1,000 to an account he controls. (Here the symbol $\oplus$ means xor.)

How did he manage that? Did our encryption system fail? No, it didn't, only because encryption, as it's usually defined and implemented, does not promise to maintain the integrity of the ciphertext message. It is a commonly-held fallacy that encryption ensures message integrity; don't believe it.

So now that we know there's a problem, how do we solve it? The good news is that there are a wealth of algorithms called "Message Authentication Codes," or "MACs," which do precisely what we want: given a message M we compute the "MAC tag" t and send it along with M to our on-line broker. The recipient then computes the MAC tag on the received message M' and if it matches the received tag t' he accepts the message as authentic; if not, he rejects it as an attempted forgery. And why couldn't an adversary alter the message M and just compute a new MAC tag himself? Because, like encryption above, there is a key K shared between you and your broker which the adversary does not possess. Computing the tag requires knowing K. Of course the adversary might alter M and then *guess* a correct tag for the forged message M but, for a good MAC, his chances are very slim.

Notice in the discussion above we said nothing about the privacy of M. We might just send M in the clear, without encryption, but this is rarely done. In settings like SSL, the cryptography used in your web browser, we want both at the same time. We can achieve this by simply using a good encryption scheme, like CTR mode with AES, and then applying a good MAC to the resulting ciphertext. In other words, M might actually be (ctr, C) as output from the CTR mode encryption algorithm. It is normally *extremely* important the the key used for encryption be completely independent from the K used for our MAC if we are to obtain a secure scheme here. But generating and maintaining two keys is perhaps less convenient than we might like, and the method is slow compared to the latest schemes for doing both encryption and authentication simultaneously.

4 OCB Mode

People had long been searching for a mode of operation which achieved both encryption and authentication simultaneously, but each attempt was subsequently broken. In August 2000, Charanjit Jutla of IBM announced the first correct scheme. Soon thereafter two teams of researchers also announced their own schemes: one team led by Phillip Rogaway of UC Davis produced the "Offset Counter Block," or "OCB" mode. Another team led by Virgil Gligor from the University of Maryland, proposed the "XCBC" scheme (and a few related derivatives). In August 2001, NIST held a second workshop to examine these modes and others. (See `http://csrc.nist.gov/encryption/modes/workshop2`.) Let's take a closer look at one of them: the OCB mode.

OCB is a bit harder to describe than the modes we've just seen. Principally because it accomplishes a challenging objective, encryption and authentication simultaneously, and also because it has been highly optimized. Nonetheless we'll sketch how it works; more details, the research paper, and reference code written in C can all be found at `www.cs.ucdavis.edu/~rogaway/ocb`.

Suppose we have a plaintext P we would like to send across the web in such a way that it's encrypted and authenticated. Once again, we'll assume our partner shares some secret random key K. The OCB mode will use K as the key to AES, as usual.

Before we begin, we'll need a "nonce." A nonce is some value which is guaranteed not to repeat throughout the life of the key; the counter value in CTR mode served this purpose above, but there is no restriction stating that the nonce must be updated sequentially. And, as before, all security is lost if the nonce is repeated, so we must ensure this does not occur.

Call the nonce N. We then compute $L = \mathrm{AES}_K(\mathrm{CONST})$ and $R = \mathrm{AES}_K(N \oplus L)$ and store these away. And finally, we we must produce some "offsets" based on L and R. These offsets will seem very mysterious perhaps, unless you sift through the paper to see why these steps are taken (if you want to impress your boss, tell him OCB generates offsets from multiples of L, in Grey-code order, in a Galois field). First we fill an `Lm[]` array:

```
Lm[1] = L;
for (i=2; i <= 16; i++) {      // message can have at most 65536 blocks
    Lm[i] = Lm[i-1] << 1;
    if (msb(Lm[i]) == 1)
        Lm[i] ^= 0x87;
}
```

Here, `msb()` is a function which returns the most-significant bit of its argument. Note that the above is pseudo-code: these values are typically 128 bits and most conventional computers cannot handle them directly as integers. Next we form the offsets in the `Z[]` array:

```
Z[1] = L ^ R;
for (i=2; i <= 65536; i++)
    Z[i] = Z[i-1] ^ L[ntz(i)];
```

Here, `ntz()` is a function which returns the number of trailing zero bits of its argument (so `ntz`(2) = 1, for example). And finally, we're ready to present the algorithm!

As before, we break M into 128-bit blocks $M_1, M_2, \cdots M_m$. Then to encrypt M_i we compute the ciphertext $C_i = \mathrm{AES}_K(M_i \oplus Z[i]) \oplus Z[i]$. And that's it, for the most part. There are some details regarding the processing of the final block of M, but these can be found on the web site or in the paper.

Now that we have the ciphertext $C_1, C_2, \cdots C_m$, we still need to compute an authentication tag to obtain an authenticated message. To get the tag, we compute $\mathrm{AES}_K(M_1 \oplus M_2 \oplus \cdots \oplus M_m \oplus Z[m])$. We are done! We now send the encrypted, authenticated message to our partner by transmitting N, $C_1, C_2, \cdots C_m$, and the authentication tag. When he receives these data, he recovers $M_1, M_2, \cdots M_m$ by reversing the algorithm, using N and K. Then he recomputes the tag and ensures that it matches the tag which was sent. Note that we have omitted some details, such as how to handle partial final blocks and how to produce smaller authentication tags, but this is the gist of the algorithm.

Like the other modes, OCB is provably secure. It can be rigorously demonstrated that "breaking" OCB is tantamount to breaking AES itself. Also, OCB is very efficient: like CTR mode, it is fully parallelizable. And many of the steps we did separately above (such as generating the offsets) can be done more efficiently and on-the-fly. In fact, the work required to process each block of message ends up being very little: a few array accesses and some `xor`s, along with an AES invocation.

Perhaps you've heard about the recent devestating attacks on the first IEEE 802.11 wireless LAN standard's security, call WEP (Wireless Equivalent Protocol)? In the newest version of 802.11, OCB now appears as the algorithm for encryption and authentication. It makes sense: OCB is very fast and is accompanied by proofs of security. It is unlikely to ever be broken.