More secure than AES - mode FMC

Stefan Westerfeld <stefan@space.twc.de>

FMC 1.0 - 21.11.2009

Abstract. In this document, we describe **mode FMC (full message chaining mode)**, a cryptographic mode which is built using AES-256 as block cipher and SHA-256 as hash function, and a newly developed algorithm called entropy propagation. When applied to a message, mode FMC provides encryption and authenticates the contents and possibly a data chunk called public data – that is not encrypted - simultaneously. Thus it is an **Authenticated Encryption with Associated Data (AEAD)** scheme. Mode FMC encryption was designed to be more secure than AES, in the sense that for some successful attacks against AES that could be discovered some time in the future, mode FMC encrypted data should still be secure.

Motivation

Mode FMC was developed for an application called dpim. For this application, the user enters highly confident data (such as his login/password for various sites), and this data is stored on an untrusted server, encrypted. The server never knows the decryption key, so as far as security is concerned, only the client encrypts and decrypts data.

How do we encrypt this data, so that we are sure that the server will not be able to decrypt it? AES seems to be the de facto standard for encrypting data. It is recommended by the NIST as block cipher. However the security of AES remains an open question. The algorithm has been reviewed or analyzed by many cryptographers, and there are no known weaknesses that are exploitable in our scenario.

However, we are encrypting highly confidential data (if a users login/password list is compromised, then a lot of harm can be done) , and we store the encrypted data on untrusted computers as part of the protocol. On the other hand, we don't need the encryption to be fast, since we expect that there is not too much data to handle.

To put it in other words: we can afford to do a lot more than plain AES-CBC encryption. But we want the extra amount of time spent doing encryption to result into better resistance against cryptographic attacks. Thus AES-FMC encryption was developed as a slower but more secure way of encrypting data.

Design goals

The term FMC = full message chaining indicates that in our encryption mode each part of the plaintext affects all parts of the ciphertext. So unlike CBC mode where changing the end of the plaintext only affects the end of the ciphertext, in FMC mode we require that changing the end of the plaintext affects all of the ciphertext – the beginning, the middle and the end. The chapter on entropy propagation describes how this design goal is implemented. We make this property part of the design goals, because we believe that by making each part of the ciphertext depend on as many variables as possible will make it harder to uncover the plaintext without knowing the key.

Mode FMC was designed to have true 256-bit security. This means that a 256-bit key is used to encrypt the data, and there is no other (known) method of decrypting the data, than using the key. Of course, an attacker which does not know the key can try every single key, which will lead to a

successful guess after trying half of the possible keys.

Mode FMC was designed to resist partially known plaintext (in case you know a part of the plaintext of the encrypted message), known plaintext and choosen plaintext attacks. Mode FMC encrypted data should remain secure even if AES alone does no longer have true 256-bit security, or if AES becomes vulnerable to – for instance – known plaintext attacks.

A 256-bit security level means:

Suppose you have a computer/processor that tries 10^9 keys per second and uses only 20W of energy (which is fairly optimistic), then in one hour, we would try $3.6 * 10^{12}$ keys, with an energy consumption of 20 Wh or 0.02 kWh. Assuming that 1 kWh costs 0.12 EUR, this produces a cost of 0.0024 EUR.

Number of keys tried	Power consumption	Price
$3.6 * 10^{12} \sim 2^{41}$	0.02 kWh	0.0024 EUR
2^{64}	102483 kWh	12297.80 EUR
2^{128}	$1.89 * 10^{24}$ kWh	$2.26 * 10^{23}$ EUR
2^{196}	$5.58 * 10^{44}$ kWh	$6.69 * 10^{43}$ EUR
2^{255}	$3.21 * 10^{62}$ kWh	$3.86 * 10^{61}$ EUR
2^{256}	$6.43 * 10^{62}$ kWh	$7.72 * 10^{61}$ EUR

The world wide energy consumption per year is 125 PWh. Since 1 PWh = 10^{12} kWh, the average energy needed to recover the key would be $3.21 * 10^{62} / 10^{12} = 3.21 * 10^{50}$ PWh. So the key recovery would take $3.21 * 10^{50}$ PWh / 125 PWh = $2.57 * 10^{48}$ years (compare this to the age of the universe of $13.7 * 10^9$ years), if all of the energy available world wide would only be used for breaking the key.

Thus, a security level of 2^{256} is more than enough for storing confidential data. The only thing we need to ensure is that there are no shortcuts which could speed up the search for the key.

Entropy propagation

Some records from a password list with the fields site, login, password could look like that (we already added padding to the next 16-byte boundary (AES uses 128-bit blocks):

- `www.mymail.com#Homer.Simpson@mymail.com#secret99`
- `www.mymoviebox.com#HSimpson#xzz39vv############`
- `\23456781234567/\23456781234567/\23456781234567/`

This means that if these records are stored using AES-CBC using Key K, and Initial Vectors IV_A and IV_B the following plaintext pairs can be guessed by an attacker:

Plaintext	Ciphertext
P_0 = („`www.mymail.com#H`“ xor IV_A)	$C_0 = E_K(P_0)$
P_1 = („`omer.Simpson@mym`“ xor C_0)	$C_1 = E_K(P_1)$
P_2 = („`www.mymoviebox.c`“ xor IV_B)	$C_2 = E_K(P_2)$

If a way to perform a known plaintext attack against AES becomes known (that is more effective than brute force key search), then a list of pairs (P_0, C_0), (P_1, C_1), (P_2, C_2)... could be sufficient to deduce the key, and thus be able to decrypt the missing and valuable information: the passwords stored in the records.

The idea of entropy propagation is to use the fact that the attacker knows some of the plaintext of each record, but not all of it. Another way to put this is the amount of bits that need to be guessed per byte of the record, for the first record:

```
www.mymail.com#Homer.Simpson@mymail.com#secret99
\23456781234567/\23456781234567/\23456781234567/
     0 bits          0 bits          40 bits
```

We don't estimate the password with 8 bits per char (in this case 64 bits would need to be guessed for the 8-char password), because guessing the password could be speed up using a word list. In any case, if the message consists of the parts

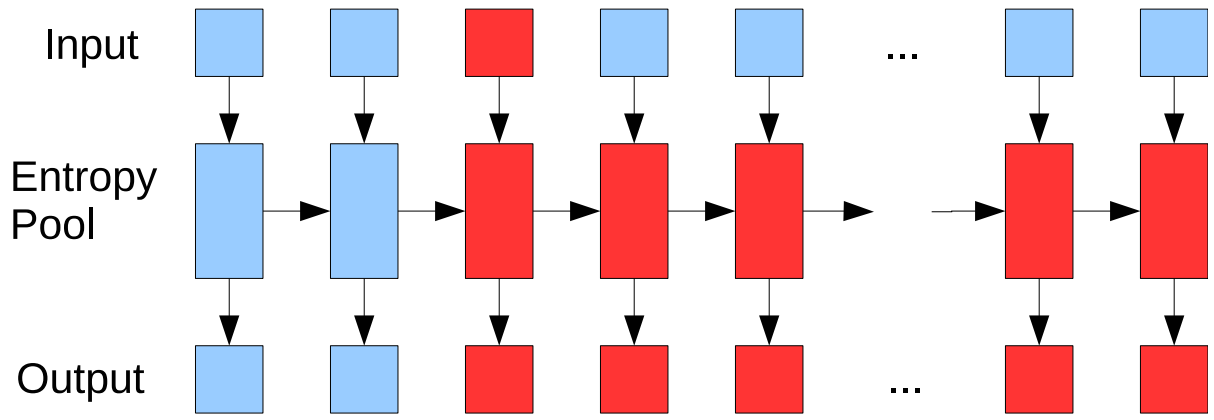
A = „`www.mymail.com#H`“
B = „`omer.Simpson@mym`“
C = „`ail.com#secret99`“

we can make life significantly harder for somebody trying a known plaintext attack by storing

A' = A xor C = `0x161e1b000e1600421a094d110a191a71`
B' = B xor C = `0x0e04095c4d3c044e03160c1c25194054`
C' = C = „`ail.com#secret99`“

This transformation will make it harder to obtain plaintext / ciphertext pairs, because although A and B are known to the attacker A' and B' are not fully known, because the attacker does not know C. Of course simply assuming the last block is always unknown to the attacker will not work in the general case. So we'll now present a transformation that also makes blocks that are known to the attacker dependant on blocks that are not known to the attacker, however not requiring knowledge which blocks are known to the attacker and which are not.

Forward entropy propagation works by processing the data forward:



A 256-bit entropy pool is initialized with a well known value (256 zero bits). For each block of the

input message three steps are performed:

- the `output[0..15]` is computed as: `input[0..15] xor pool_state[0..15]`
- `pool_state[0..15]` = `output[0..15]`
- `pool_state[0..31]` = `diffuse (pool_state[0..31])`

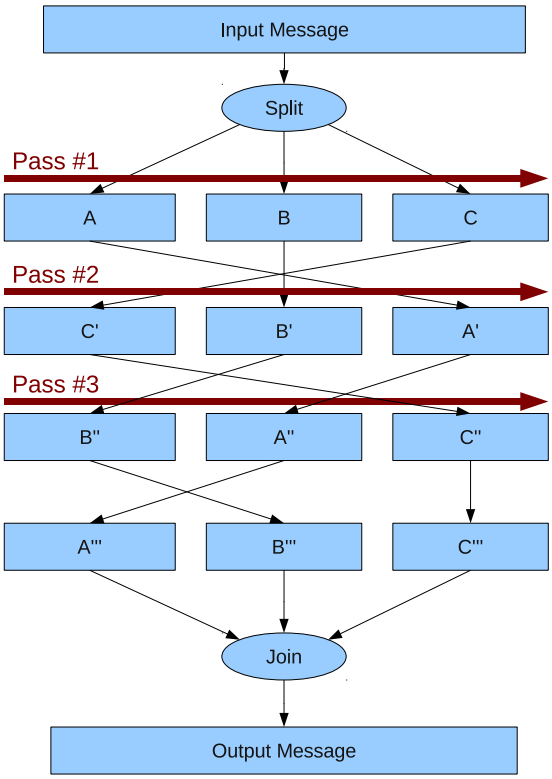
So if an attacker knows fully the blocks marked in blue in the input, he also knows the first two blocks of the output (since the entropy pool state can be fully predicted from the inputs). However, with the first red block, the entropy pool state is no longer known to the attacker. In fact, if the attacker needs to guess 40 bits, to guess the portion of the third block that he does not know, then there are 2^{40} possible states the entropy pool could have. So even if the attacker knows the rest of the message fully, the output of the entropy propagation continues in one of 2^{40} ways. All output blocks after the third are thus red as well.

It should be easy to see that by reordering the blocks and performing more than one pass of entropy propagation we can get rid of the first two known blocks in the above scenario and produce 2^{40} possible sequences which depend on the possible values the third block could have. Thus, although we're working with the assumption that the attacker knows almost the complete message, entropy propagation maps this message to an unknown message, using the knowledge that the attacker does not have, without knowing which blocks are or are not known to the attacker.

We will later see that it makes sense to introduce extra blocks which equally likely could contain any possible value (2^{128} possibilities per block). By doing so, an attacker can no longer know any block of the output of the entropy propagation, even if he knows the whole „plaintext“ (except for the extra blocks introduced during encryption). This is the reason we say that mode FMC is resistant against known plaintext and chosen plaintext attacks.

It may seem as doing one forward pass of entropy propagation and one backward pass (reversing the order of the blocks) is enough to make every block depend on every other block. However, an example will reveal that this does not produce optimal results. If the attacker knows all blocks except for the last block, and the first half of the last block (so he knows 64 out of 128 bits), then the forward pass will not change the amount of data known to the attacker. The forward entropy propagation will effectively be reduced to a XOR with a constant for all known blocks. The last block will then also be XORed with the state of the entropy pool accumulated up to this point, which is 100% known by the attacker. So after forward entropy propagation, the first blocks remain known, and the first half of the last block remains known.

Now we reverse the order of the blocks. The first half of the first block is known. The second half of the first block is unknown. All other blocks are known. The state of the entropy pool is zero. Another forward entropy propagation will leave the first block as it is (as there is no entropy in the entropy pool, yet), and create 2^{64} possible continuations with randomized versions of the other blocks. So after the „backward propagation“ the attacker still knows the first 64 bits of the first block. To put it in a different way: if the goal of entropy propagation is to make every bit of the output dependant on every bit of the input, then a two-pass forward-backward propagation is not enough, because in our case the bits within the last (first) block do not depend on the other bits within the first block. However, with three passes over the data, the goal of making every bit of the output depend on every bit of the input can be reached.



The graphic shows the complete algorithm. A message is first split into three parts A, B and C. Each of the parts contain only complete blocks (splitting at any position that is not evenly dividable by 16 bytes does not happen). If the message cannot be divided into three parts with the same number of blocks, A gets one extra block, and if there is some data left, B gets one extra block. So a message of 6 blocks would be split into (A = 2, B = 2, C = 2) blocks, a message of 7 blocks into (A = 3, B = 2, C = 2) blocks and a message of 8 blocks into (A = 3, B = 3, C = 2) blocks.

After splitting the input message into three parts, one forward entropy propagation called **pass #1** is performed over (A, B, C) and results in (A', B', C'). This is equivalent to running forward entropy propagation over the input message.

We reorder the parts, so that the next input for the entropy propagation – **pass #2** - is (C', B', A'). Note that although we reverse the order of the parts, this is not the same as reversing the whole message, because we do not change the order of the data within C' (or another part).

The output is again reordered so that the final and last pass, **pass #3** operates on (B", A", C"). At the end of the last pass, the original order is restored, and the output message is obtained by joining A'", B'" and C'".

To see that the operation does in fact produce a result in which a change in one arbitary input block influences every output block, we first assume that the change takes place in A. If one block in A changes, then the entropy pool state will be different after processing this block in pass #1. This means that the entropy pool state will be different at the start of B. So every block in B' will be changed. And – for the same reason every block in C' will be changed.

So if we assume that a modification takes place in A, B' and C' will be changed. If we assume that a modification takes place in B, the entropy pool state after processing B be will be different, so every block in C' will be changed. So from the point of view of minimizing the number of blocks that are changed after pass #1, the best possible position is to change the last block of C. Every other change will result in the last block of C' being changed as well, but also a number of blocks elsewhere.

By showing that if the last block of C changes, every output block changes, we also show that if an arbitary block changes, every output block changes. Suppose only the last block of C changes. Then the output of pass #1 will not change except for the last block in C'. And even this block will only change in those bits that were different in the input, since the entropy pool has the same state, because no blocks before the last block in C were different.

In pass #2, the entropy pool state will be different after processing C', which means that the unchanged input A' will result in a completely different A" and the unchanged input B' will result in a completely different B". Or in terms of blocks, after pass #2 the last block of C" and all blocks of A" and B" are changed.

Finally in pass #3, since A" is completely different, so is A'", since B" is completely different so is B'", and since the entropy pool at the beginning of processing C" is completely different (because it depends on A" and B"), C'" is completely different. So a small change in any of the input blocks will change every output block, or to make a statement about single bits: every output bit depends on every input bit, since changing one input bit will change approximately half of the output bits.

The entropy pool „diffusion“ function

To implement entropy propagation as described above, after adding new data to the entropy pool, a diffusion function is applied. Since blocks are only added to the first 16 bytes of the entropy pool, the diffusion function should mix all 32 bytes, so that the data is spread over all pool bytes. However, this process should not loose any information; so the pool state after diffusion should still contain all 256 bits of entropy than before the diffusion. We ensure this property by making the diffusion function bijective, or to put it in another way: the diffusion function diffuse (state) is invertible, so that for all possible states: state = $diffuse^{-1}$ (diffuse (state)).

We build our diffusion function with some of the primitives used by AES (Rijndael).

The S-Box

The Rijndael S-Box is an invertible function S_{RD} which substitutes every byte with another byte. The implementation can simply be a 256-byte lookup table, where the contents of the table are computed with the same algorithm an AES implementation might use.

Mixing function

Similar to AES's MixColumns, we specify an invertible function which mixes four bytes (a_0, a_1, a_2, a_3), resulting in four bytes (b_0, b_1, b_2, b_3):

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

Note that multiplication takes place in AES's finite field. The mixing function can – together with the S-Box – be implement using table lookups. For details, consult AES/Rijndael specifications.

Permutation of the state array

Mixing affects four consecutive bytes. Its output depends on all these bytes. A permutation is used so that the next mixing step will include one byte of different four byte groups. That way the output of two mixing operations will depend on 16 bytes.

So in the permutation we simply group the state bytes by their modulo 4 remainder.

(e_0 e_4 e_8 … e_{28} e_1 e_5 e_9 … e_{29} e_2 e_6 e_{10} … e_{30} e_3 e_7 e_{11} … e_{31})

Round constant addition

The round constants (A, B, C) are computed by encrypting the plaintext 0x00 0x01 0x02 … 0x59 with the 256 bit AES key „FMC8FMC8FMC8...FMC8“. They are XORed to the state.

Diffusion function steps

Here are the steps executed for the diffusion function:

1. add round constant A to the 32 byte array
2. apply S-Box on each byte
3. apply mixing function to each four byte block
4. permutation
5. add round constant B to the 32 byte array
6. apply S-Box to each byte
7. apply mixing function to each four byte block
8. permutation
9. add round constant C to the 32 byte array
10. apply S-Box to each byte
11. apply mixing function to each four byte block

Note that there is no extra permutation at the end. As each step has an inverse operation, the whole function is invertible.

Strongly non-separable decryption

For some encryption schemes, like AES-CBC, it is possible for an adversary to guess an encryption key, and try to decrypt a single block. If that block decrypts to useful plaintext (for instance to all printable characters), the key is probably right, otherwise another guess needs to be made. We call encryption schemes that allow decrypting a part of the message individually **separable**. If an adversary is forced to decrypt the whole message, to see whether the key is right, we call the encryption mode **strongly non-separable**. This distinction is made in [1], and our encryption scheme is in fact similar to the one presented there, and is **strongly non-separable**.

The reason that our scheme was designed to have that property is that by requiring strongly non-separable encryption, we also exclude what we'll call **local search**. Local search means that the adversary takes one small piece of the ciphertext (usually one block), and analyzes only this part of the message until he figures out the corresponding plaintext. For this to be successful, there needs to be a local constraint, such as the plaintext consists of printable characters, or stronger, the plaintext consists of a piece of english text. Local search can also be performed in separable encryption schemes if the plaintext of one block is known. Depending on the strength of the block cipher there might be ways to combine some local constraints to recover the key.

So what we want to enforce is **global search**, that is, we want that analyzing a part of the ciphertext alone will not reveal any useful information about the plaintext. Only by analyzing the whole message the plaintext can be revealed. The motivation for making our mode strongly non-separable is not so much that it will take longer to try one key. Even if the message has a length of 2^{20} blocks (16 MB), the amount of energy required for brute force search is only changed by a factor of approximately 10^6. Given the already exorbitant amount of energy/time required for brute force search, this is only a small improvement.

The reason for designing the mode non-separable is that by doing so, we make it harder to predict what is a reasonable result for decrypting the first (or any other) block. Ideally the answer to the

question what would constitute an appropriate result of decrypting the first block would be: anything. Thus a local search for the key used to encrypt the first block would be impossible.

So how to achieve non-separable encryption? The idea is simple (and a similar algorithm is presented in [1]). Here are the steps:

- $I = E_R(P)$ encrypt the plaintext with a random key K_R
- $H = SHA256\ (I)$ compute H as SHA256 sum of the ciphertext I
- $C = E_S\ (I \parallel H \text{ xor } R)$ encrypt I and (H xor R) with the secret key K_S

Why is this strongly non-separable? Suppose we decrypt only one block of C. The result could be a block of I. But we do not have the key K_R. So we cannot proceed to decrypting the plaintext. All we have is one block of basically random data (since the plaintext was combined with R, and R was designed to have each of the possible 2^{256} values with equal probability). Only if we decrypt *all of* C we can compute the H = SHA256 of I, and by XORing the last 2 blocks of the decrypted C with H we can recover P. So the mode is **strongly non-separable**.

The details of the actual FMC specification given below are somewhat different, but the property of making **local search** impossible remains.

Key derivation

Since the security level of mode FMC is 256-bit, the encryption uses one master key of 256 bit. However, during encryption, two subkeys key K and key S are used; these are both derived from the 256-bit master key. The key derivation works like this:

```
K = SHA-256 (master_key || „key encryption key“)
S = SHA-256 (master_key || „secret key“)
```

By using this key derivation K and S are different, and should not be related by sharing any obvious property. So if an attacker can compute one of the keys through an attack, for instance key S, it is not possible to deduce the corresponding key K, since SHA-256 is a one-way function.

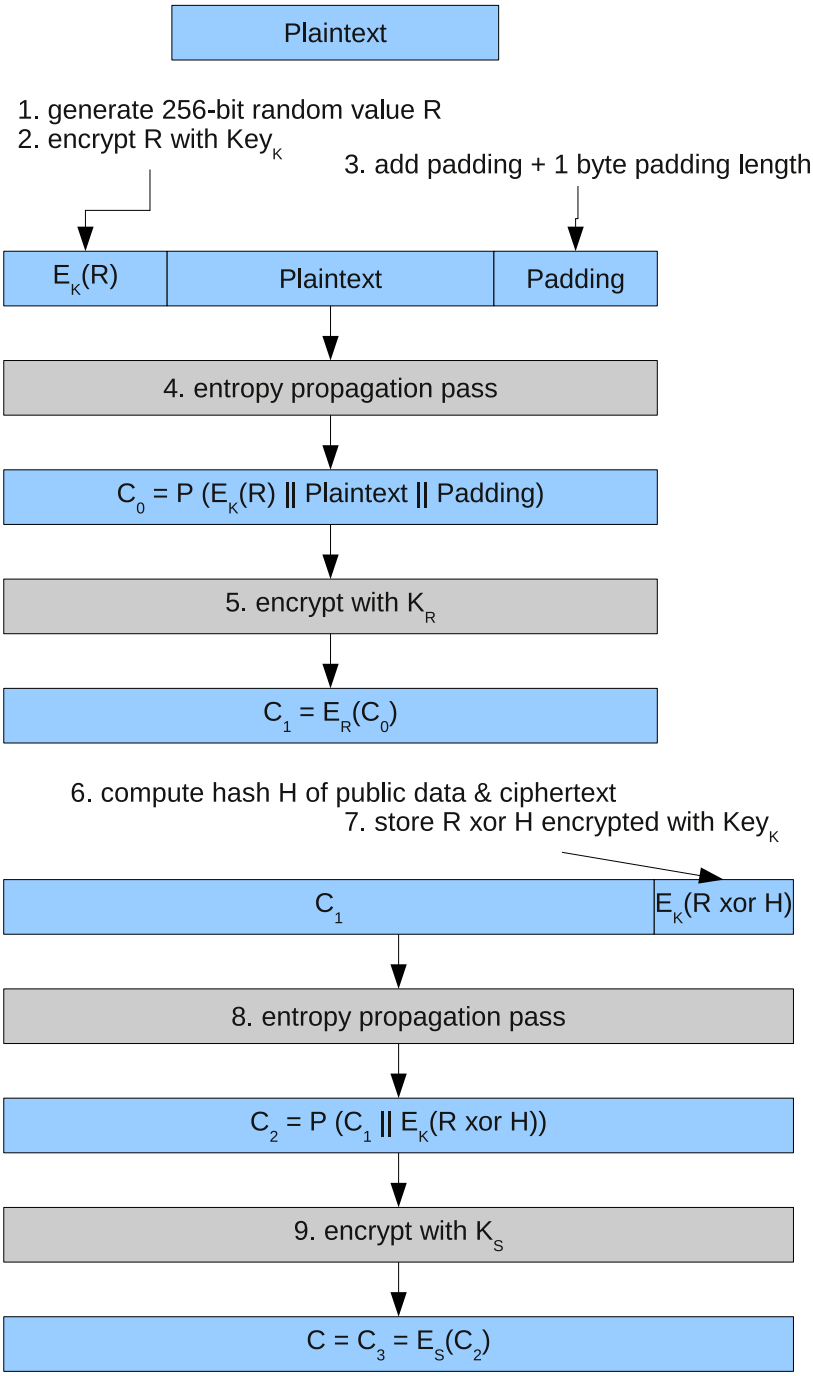
Of course it is theoretically possible to generate all (K, S) pairs in 2^{256} steps, by deriving them from all possible 256-bit master keys, but practically this is impossible as the calculations about power consumption show.

FMC encryption description

The first step of FMC encryption is to generate a 256-bit random value R. This value is encrypted with key K. This and all following encryptions are performed in AES-256-CBC mode which means the encryption is performed using AES with a 256-bit key in cipher block chaining mode. Initial vectors are not used.

Padding is appended to (E_S(R) || Plaintext): first a number of random bytes, and as a last byte, the number of padding bytes including this number byte. Padding should always append at least 8 random bytes, to ensure that some randomness is found at the end of the (E_K(R) || Plaintext || Padding).

Then, an entropy propagation is performed over this data, followed by encryption which results in $C_1 = E_R(C_0)$.



The hash H is computed like this: D shall be „the public data“ || 0xff, so if the public data is 0x20 0x20, D shall be 0x20 0x20 0xff. Then, D and C1 are XORed together bytewise S = (D_0 xor $C1_0$, D_1 xor $C1_1$, D_2 xor $C1_2$, …). The shorter of the two is padded with 0x00 bytes so that D and C_1 have the same length. Over this string S a SHA-256 hash is computed.

The value (R xor H) is encrypted with key K, and the concatenated value (C_1 || E_K (R xor H)) is used as input for another entropy propagation.

Finally, the output of this entropy propagation C_2 is encrypted with key S, producing the ciphertext C = C_3 = E_S (C_2).

We have listed a few properties of this encryption scheme in earlier chapters. So here we'll just state that FMC encryption is **strongly non-separable**: since every block of C_1 can only be decrypted using the key K_R, which is different for every FMC encryption, R needs to be computed from the value (H XOR R). Since computing H requires knowing all of C_1 and all of D, C_1 must be fully decrypted to recover R. Since (H XOR R) is stored encrypted with key K, to perform a successful decryption one would need to know key K and key S.

Another property is that FMC encryption **resists known plaintext** and **choosen plaintext** attacks. This is the case because by choosing a plaintext (or by knowing a plaintext) one cannot know or choose the random value R. This means that as soon as the entropy propagation is applied to (E_K(R) || Plaintext || Padding), the result is unknown to the attacker, and can not be reconstructed without guessing R. Since R is a 256-bit value, and since R can be every possible 256-bit value, E_K(R) can also be every possible 256-bit value. But this means that the input of the entropy propagation contains two blocks which are not known by the attacker, and thus the result of the entropy propagation will be unpredictable for the attacker.

FMC decryption description

Besides undoing all steps performed during encryption, decryption also validates the data. Tampering with encrypted data or decrypting data that was not encrypted by mode FMC with the same master key will be detected. The first step in decrypting is therefore the validation of the length. Since there is at least one block of content, and two blocks are added for storing $E_K(R)$ at the inner level, and two blocks for storing $E_K(R \text{ xor } H)$ at the outer level, any message smaller than 5 16-byte blocks is rejected as invalid.

After checking this,

$$C_2 = D_S (C_3)$$

can be computed using AES with the 256-bit key S. All decryptions use AES-256-CBC, that is AES using a 256-bit key in cipher block chaining mode, without initial vector. Then, by undoing the entropy propagation

$$C_1 \,||\, E_K(R \text{ xor } H) = U (C_2)$$

can be obtained. By hashing C_1 and the public data (as described in the encryption), H can be computed, and R xor H can be obtained by decrypting the blocks that were encrypted with key K. After obtaining R in this way,

$$C_0 = D_R (C_1)$$

can be computed. Then

$$E_K (R) \,||\, \text{Plaintext} \,||\, \text{Padding} = U (C_0)$$

can be computed by undoing the entropy propagation. What remains to do is checking correctness. Since we have checksummed C_1 and the public data with the result H, and computed R as the difference between a stored value of H xor R and H, there are two cases. Either our computed H and the original H match. Then we should have obtained the correct value of R, and we are sure that neither public data nor C_1 was tampered with. Or we have computed an incorrect R*, and should reject the message.

To check which of the two cases occured, we decrypt the first two blocks, which contain $E_K (R)$. If our R matches with the value, then it is highly likely that the message is authentic. Thus we remove the padding and return the plaintext. If we have an R* which is different from the R, the message is not valid. There is a small chance that if public data and/or C_1 was modified we still find that the R value matches. However, this small chance has a probability of 2^{-256}, which means that its so unlikely that it will not happen in practice.

Quantifying the improvement over AES

From the discussion of mode FMC encryption it is should by now be plausible to believe that mode FMC is **stronger than AES**. But even so, how much stronger exactly is stronger? Here is an idea on how to quantify this. We claim that if AES is broken, then FMC encrypted data will remain secure. So to observe the behaviour FMC has when working with a broken cipher, one could step by step reduce the number of rounds is the AES-encryptions mode FMC uses. What if we used a 13-round AES? 12-round AES? 10-round AES? … 1-round AES?

What if we didn't even use AES at all, but did a XOR with the key as only encryption (of course to model a 128-bit block cipher, we would only use the first 128 bit)?

If we knew that the best we could to is breaking 4-round-AES mode FMC – for instance – would lead to a nice security margin of 10 (because we usually work with full = 14 round AES). This would give us a number to quantify how much stronger mode in fact FMC is.

We'll consider the simplest case here, defining a block cipher XOR-128, which we will use for K and S encryptions, which consists of a bytewise XOR with the first 128 bit of the key K or the key S. Usually XOR-128 is a quite weak cipher. If you encrypt one kilobyte of english text with XOR-128, then a probability function

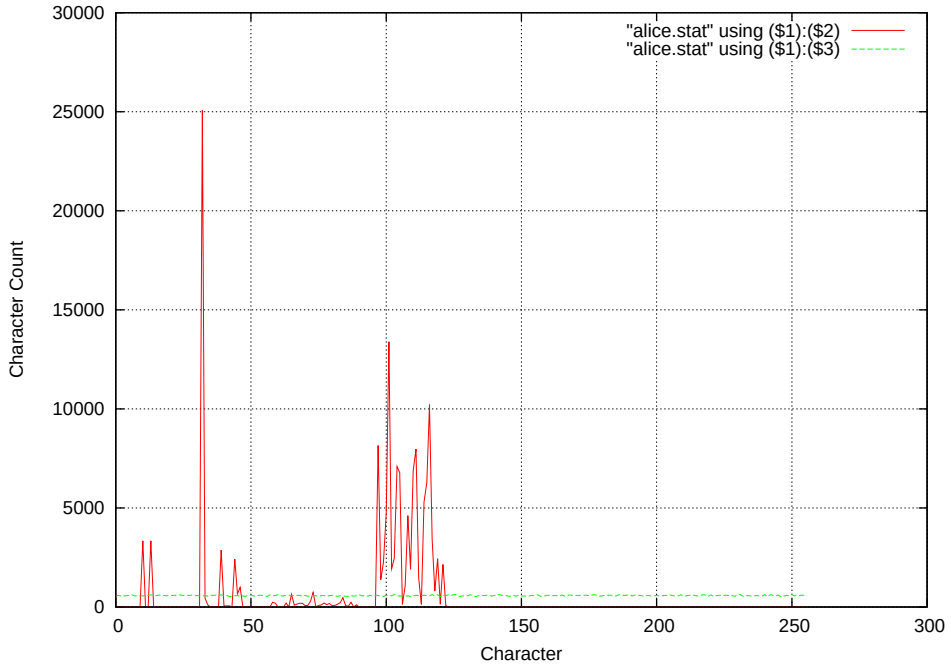
$$p: Z_{256} \rightarrow [0,1) \text{ with } p(l) = p(\{\text{the probability of letter } l \text{ in english text}\})$$

can be used to compute a score that for the first key byte having the value v:

$$score(\{\text{first key byte is v}\}) = p(data[0] \text{ xor } v) * p(data[16] \text{ xor } v) * \ldots * p(data[1008] \text{ xor } v)$$

The math is the same for other key bytes. So by using a statistic of how likely each letter is, we can find likely key bytes. Reconstructing the key from this information is relatively easy.

However, this method does not work for XOR-128-FMC encryption. Entropy propagation has the property that even if a plaintext block had statistical properties that would allow retrieving the key, after entropy propagation all characters occur approximately with the same probability. Here is a statistic of the text „Alice in Wonderland“ by Lewis Carroll:



The red line is the statistic collected from the raw text (plain ascii). The character 32 (Space) seems to be extremely frequent and characters between 97 and 122 (the lower case letters 'a' – 'z'). On the other hand there are many characters that do not occur in this text, like any character greater than 127. So if a guessed key byte xor a data byte results in such a character, we can conclude that we guessed the wrong key byte.

The green line is the statistic after running entropy propagation over the text. Here the frequency of all the letters is approximately the same, and it is no longer obvious how to find a key byte. Are there other properties (like for instance two-letter combinations, which in the plaintext for english would be for instance „th“, which is frequent) that can be exploited?

File	Size (bytes)
alice.txt	147788
alice.txt packed with bzip2	42792
alice.txt.ep – entropy propagated alice.txt	147792[1]
alice.txt.ep packed with bzip2	148872

As a simple test of whether there are patterns in a piece of data, we use bzip compression. As you can see, the normal english text can be compressed nicely, because english text is predictable to a certain degree. This regular structure allows us to reconstruct the key from XOR-128 encrypted ciphertext. However after alice.txt has been subjected to entropy propagation, there are no patterns that bzip2 could identify and use to reduce the length of the file. On the contrary: it grows a bit.

So it seems that there are no obvious patterns in the output of an entropy propagation pass, and thus the method of scoring key bytes we used above does not help us for XOR-128 mode FMC encrypted data.

There are two more properties that make it hard to recover key S from ciphertext. One is that there is no obvious way to test a key hypothesis like { the second key byte has the value 42 }. When we were facing XOR-128 on raw text, we could simply xor each 16^{th} data value with 42, starting at the second data value, and look at the resulting characters. If these were distributed according to a distribution similar to the distribution of english plaintext, then the hypothesis would be true, otherwise false. Characters that never occur in english plaintext like 0x04 or 0xf4 make some key bytes entierly impossible.

But what if we're looking at the output of an entropy propagation pass? Which – in mode FMC – contains mostly data that is the output of another entropy propagation pass XOR-128 encrypted with a random key R? Since any byte is approximately equally likely, we can't test a hypothesis like { the second key byte has the value 42 } by looking at every 16^{th} data value alone. Any value is equally likely, so we can not gain information on whether our hypothesis is correct in this way.

It seems as if we need to undo the entropy propagation step to figure out useful information. The intersting property of inverse entropy propagation is: all output bits of the inverse entropy propagation depend on each input bit. So can we test a hypothesis about the second key byte *alone*?

No. If we just XOR the 2^{nd} data value, 17^{th} data value, … 1009^{th} data value with a hypothetic second key byte (maybe 42), and then undo the entropy propagation, then every byte of the output will be incorrect. Why? Because by not XORing the other data bytes with their correct key bytes, we have changed many bits in the input of the inverse entropy propagation. And changing just one bit already destroys all plain text. So to test the hypothesis that the second key byte is 42, we're forced to also choose the first key byte, the third key byte, … and all choices need to be correct, before the inverse entropy propagation will yield the correct plaintext.

Which brings us to the second difficulty: if we had guessed the 128-bit key correctly, and could undo the entropy propagation, all that we could see would be the XOR-128 with key R encrypted output of another entropy propagation pass. Now there is $E_K(H \text{ xor } R)$ at the end of this text, but without knowing key K (another 128-bit value), reconstructing R would be impossible. So we could try guessing K or R, and then use inverse entropy propagation to finally see the plaintext. However, we would then need to guess a total of 256 bits, which makes the attack as slow as brute force.

1 This is slightly bigger than the original file because for entropy propagation the size of the input data needs to be dividable by 16

At this point we could conclude that we have a security margin of 14 rounds of AES encryption, because XOR-128-FMC is impossible to break. However, as mode FMC has not been exposed to public review until now, its perhaps better to state that there is no known exploit on any version of mode FMC with a simplified block cipher, and time will tell whether such an exploit becomes known. However, it seems that there is *no trivial exploit* even for XOR-128-FMC, which has a block cipher that is a lot easier than the one AES-FMC uses.

Performance considerations

Although mode FMC was designed as slow-but-secure encryption, during development the dpim application with its requirements was used as guide on how fast exactly mode FMC needs to be. The dpim application requires storing a user's todo list, passwords, phone numbers, addresses, email addresses and some other stuff in a database. These records are also versioned, which means that the if a friend moved elsewhere (new phone number, new address), some old records would still remain in the database. The maximum number of these records was estimated to be 20000 records (although it would be likely that this limit is too high for almost anybody). Further the number of bytes per record estimated to be 256. A time limit of one second for reading the database results in a performance requirement of 20000 record decryptions / second.

The measured data is (AMD Phenom(tm) 9850 Quad-Core Processor, running at 2.5 Ghz, 32-bit):

```
Testing decrypt performance:
 => 19688.1388 records / sec
 => 496.0157 cycles per byte

Testing AES performance:
 => 161904.1075 records / sec
 => 60.3173 cycles per byte
```

Although plain AES would be significantly faster, the mode FMC decryption is fast enough for the dpim application.

Summary

We presented mode FMC, an AEAD scheme based on AES, designed with the goal to be more secure than AES. A number of components in the FMC algorithm, such as the strongly non-separable design and the entropy propagation result in an algorithm that should resist known plaintext and choosen plaintext attacks better than AES alone does. One way of quantification of the extra security was discussed, and would require somebody to come up with an attack against an mode FMC with round reduced AES. We also discussed that although mode FMC can be implemented in a way that offers reasonable performance for the application we had in mind, plain AES is significantly faster, so whether or not using mode FMC is a good idea requires knowing how much data you'll need to process.

References

1. Ronald L. Rivest, All-or-nothing Encryption and The Package Transform