CSE 565 Sample Solution to Homework 1

As was previously announced, there are extra credit points (10 points) for providing code in question 4 using low-level AES built-in functions such as in C. We distributed these extra points across questions 4 and 5 so that they are for providing the code and also running it.

1. (9 points total; 3 points each) For part (a), documents containing sensitive personal data (such as salary, social security number, etc.) would be a good answer. For part (b), it could be documents containing financial information (such as tax documents, bank statements, etc.). For part (c), documents that have to meet a strict schedule (such as publishing a newspaper, weather forecast for a departing ship or plane, etc.) are a good answer.

2. (6 points)

   (a) The problem with the code is that it doesn't comply with the principle of fail-safe defaults. If the program runs into an unexpected issue and the call to `IsAccessAllowed` results in an incorrectly formed returned value which is not `ERROR_ACCESS_DENIED`, access will be automatically granted.

   (b) To avoid the flaw, we need to modify the logic to default to denying access.

```
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ACCESS_ALLOWED) {
  // grant access
} else {
  // deny access and notify the user
}
```

3. (30 points total)

   (a) (10 points) Using the formulas in the book, we obtain the following for ciphertext $C = (L_2, R_2)$:

   $$L_1 = R_0$$
   $$R_1 = L_0 \boxplus F(R_0, K_0, K_1, \delta_1) = L_0 \boxplus (((R_0 \lll 4) \boxplus K_0) \oplus ((R_0 \ggg 5) \boxplus K_1) \oplus (R_0 \boxplus \delta_1))$$

   $$L_2 = R_1 = L_0 \boxplus (((R_0 \lll 4) \boxplus K_0) \oplus ((R_0 \ggg 5) \boxplus K_1) \oplus (R_0 \boxplus \delta_1))$$
   $$\begin{aligned} R_2 = L_1 \boxplus F(R_1, K_2, K_3, \delta_2) &= R_0 \boxplus (((R_1 \lll 4) \boxplus K_2) \oplus ((R_1 \ggg 5) \boxplus K_3) \oplus (R_1 \boxplus \delta_2)) = \\ &= R_0 \boxplus (((L_0 \boxplus (((R_0 \lll 4) \boxplus K_0) \oplus ((R_0 \ggg 5) \boxplus K_1) \oplus (R_0 \boxplus \delta_1)) \lll 4) \boxplus K_2) \oplus \\ &\oplus ((L_0 \boxplus (((R_0 \lll 4) \boxplus K_0) \oplus ((R_0 \ggg 5) \boxplus K_1) \oplus (R_0 \boxplus \delta_1)) \ggg 5) \boxplus K_3) \oplus \\ &\oplus (L_0 \boxplus (((R_0 \lll 4) \boxplus K_0) \oplus ((R_0 \ggg 5) \boxplus K_1) \oplus (R_0 \boxplus \delta_1)) \boxplus \delta_2)) \end{aligned}$$

   (b) (15 points) Suppose that we request a ciphertext on message consisting of all zeros, i.e., $M = (L_0, R_0) = \{0\}^{64}$. Since adding a 0 to any value or XORing a 0 with any value does not modify that value, we obtain that in this case $L_2 = R_1 = K_0 \oplus K_1 \oplus \delta_1$. Because the value of δ_1 is known, we easily can recover $K_0 \oplus K_1$ from L_2.

   Note that recovering similar information about K_2 and K_3 is not as easy. That is, we can recover $(((K_0 \oplus K_1 \oplus \delta_1) \lll 4) \boxplus K_2) \oplus (((K_0 \oplus K_1 \oplus \delta_1) \ggg 5) \boxplus K_3)$ using the same plaintext, but searching for a message that will produce $L_2 = R_1 = 0$ will require many trials.

   (c) (5 points) By adding two more rounds to the block cipher, the same information about the key can no longer be recovered due to more complex transformation on the plaintext and the key.

4. (20 points total)

   (a) (10 points plus 5 extra credit points) One possible solution using C is as follows:

```
 int i;
__m128i aes_key, message, cipherblock; // original key and block
__m128i tmp, key[11];
// key expansion
 key[0] = aes_key;
 tmp = _mm_aeskeygenassist_si128(key[0], 0x1);
 key[1] = key_ops(key[0], tmp);
 tmp = _mm_aeskeygenassist_si128(key[1], 0x2);
 key[2] = key_ops(key[1], tmp);
 tmp = _mm_aeskeygenassist_si128(key[2], 0x4);
 key[3] = key_ops(key[2], tmp);
 tmp = _mm_aeskeygenassist_si128(key[3], 0x8);
 key[4] = key_ops(key[3], tmp);
 tmp = _mm_aeskeygenassist_si128(key[4], 0x10);
 key[5] = key_ops(key[4], tmp);
 tmp = _mm_aeskeygenassist_si128(key[5], 0x20);
 key[6] = key_ops(key[5], tmp);
 tmp = _mm_aeskeygenassist_si128(key[6], 0x40);
 key[7] = key_ops(key[6], tmp);
 tmp = _mm_aeskeygenassist_si128(key[7], 0x80);
 key[8] = key_ops(key[7], tmp);
 tmp = _mm_aeskeygenassist_si128(key[8], 0x1b);
 key[9] = key_ops(key[8], tmp);
 tmp = _mm_aeskeygenassist_si128(key[9], 0x36);
 key[10] = key_ops(key[9], tmp);

 // encryption
 tmp = message;
 tmp = _mm_xor_si128(tmp, key[0]);
 for (i = 1; i < 10; i++)
   tmp = _mm_aesenc_si128(tmp, key[i]);
 cipherblock = _mm_aesenclast_si128(tmp, key[10]);
```

   The code above makes use of the following function:

```
__m128i key_ops(__m128i tmp, __m128i tmp1) {
  __m128i tmp2;

  tmp1 = _mm_shuffle_epi32(tmp1, 0xff);
  tmp2 = _mm_slli_si128(tmp, 0x4);
  tmp  = _mm_xor_si128(tmp, tmp2);
  tmp2 = _mm_slli_si128(tmp2, 0x4);
  tmp  = _mm_xor_si128(tmp, tmp2);
  tmp2 = _mm_slli_si128(tmp2, 0x4);
  tmp  = _mm_xor_si128(tmp, tmp2);
  tmp = _mm_xor_si128(tmp, tmp1);
```

```
  return tmp;
}
```

(b) (10 points) The expected answer is that for the AES instruction set to be accessible from a particular programming language, there should be support for it in the form of built-in functions or cryptographic libraries. C has built-in functions that closely map to the available hardware instructions, while others (such as python and Java) allow programmers to call hardware accelerated implementations of AES, but without access to low-level functions. The cryptographic libraries themselves may be written in a different language (i.e., python's implementation of hardware accelerated AES can be written in C). So overall there needs to be programming language support for this functionality.

5. (25 points total)

(a) (10 points) Answers vary. Using a cryptographic library for this functionality is expected to be a few lines of code and involve key generation (for the desired key type and bitlength) and encryption in the ECB mode.

(b) (15 points plus 5 extra credit points) In general, single-block AES encryption is quick and for encrypting only a single block in a program, other overhead associated with running a program will dominate. For that reason, the assignment is asking to iterate over 1000 encryptions. Many languages (even interpreted languages such as Java) have efficient implementations of cryptographic operations, and the runtime of cryptographic operations is often comparable across different languages.

If you ran two different implementations that use AES-NI, the runtime is expected to be comparable, with the difference stemming from non-cryptographic operations. If you have one AES-NI-enabled implementation, while the other was not, after enough iterations of the encryption algorithm you should see the difference in performance. Encrypting one block using AES-NI uses only a few CPU cycles and is most efficient when the key is fixed (key expansion is noticeably slower than a single-block encryption). For that reason, encrypting 1,000,000,000 blocks using AES-NI takes on the order of seconds.