6.857 Homework	**Problem Set 2**	# 2-1 - MAC Attack
Victor Williamson		
Victor Chan		
David Feldman		March 2, 2009

a) This MAC procedure is not a good one. It is possible for an adversary to recover K using merely the original message M and MAC(K,M), both of which are sent in the clear and are readily available to the adversary.

During creation of the MAC we let:

$$V_0 = K$$

for each i = 1,2,...,n

$$V_i = encryptAES(M_i, V_{i-1})$$

to produce

$$MAC(K, M) = V_n$$

It is very simple to reverse this procedure. We would again pad M and break it into n 128-bit blocks. We then use AES decryption as follows:

$$V_{i-1} = decryptAES(M_i, V_i)$$

where

$$V_n = MAC(K, M)$$

as received from the sender, and produce

$$K = V_0 = MAC(M_1, V_1)$$

Note that

$$xxxAES(K, B)$$

denotes the encryption or decryption of a 128-bit plaintext block B with a 128-bit key K using the AES algorithm.

b) Adding a checksum to this MAC will not make it any better. This adds no protection to the MAC because the XOR of the message blocks does not prevent the key from being retrieved. It just increases i by 1 in the reversal step. Since an attacker is able to retrieve K, they will still be able to create fake messages and the correct MAC's for these messages. It would be trivial to add an additional "correct" checksum to the fake message. Logically, adding a checksum defined by XORing all of the message blocks together adds no additional information, since all of the messages are sent unsecured, and a listening adversary could just as easily compute the checksum himself.

6.857 Homework	**Problem Set 2**	# 2-2 - Truncated-Key AES
Victor Williamson		
Victor Chan		
David Feldman		March 2, 2009

We found (K_{29},K'_{29}) to be (0x0214ced4, 0x1353827b) in hexadecimal. First we note that meet in the middle reduces the computation time from 2^{2n} to 2×2^n. This works by encrypting over K values and decrypting over K' values until an intermediate value matches. Next we determine what is feasible. We choose JAVA over Python because of greater speed and performance. A Toshiba laptop running Windows Vista with 3.00 GB of RAM on Intel Core Duo 1.83 GHz was used to set benchmarks. The JAVA JRE 1.6 was able to compute up to $s = 2^{23}$ before crashing with out of memory. For s greater than 23 we iterate multiple times to compensate and run the garbage collector to free up memory between iterations. For example, for $s = 24, 25, 26$ we iterate 2, 4, and 8 times respectively. The program consists of two phases. The encryption phase uses the iteration's keys to encrypt P_s to intermediate values and the decryption phase decrypts C_s trying all $K' < 2^s$ and checks against stored intermediate values. As s increases, the majority of the time per iteration is spent in the decryption phase. The program spends about 8 seconds per million decryptions. For 2^{32} the program would take approximately 9.5 hrs. for decryptions on each of 512 iterations which reaches over 203 days, for 2^{30} it takes 2.38 hrs. for each of 128 iterations reaching close to 13 days. Our best bet is 2^{29} where the decryption phase takes approximately 1.19 hrs. on each of 64 iterations totaling nearly 3.2 days. For 2^{29} the program successfully found K, K' at iteration 20 in roughly 1.5 days.

Our program uses constant memory and the limiting factor is time. The Java Virtual Machine maxes out at 1500M of memory, and this is chosen for every s. Note we only consider $s \geq 23$. Our code stores 64 bytes for each of 2^{23} intermediate value entries because JAVA stores single bytes as 4 bytes. For cost in time we require 2^{23} encryptions and 2^s decryptions for each of $2^s/2^{23}$ iterations.

Memory =

$$1500M \geq C2^{23}M \times 64M = C2^{29}M \approx C512M$$

Time ≈

$$O(\frac{2^s}{2^{23}}(2^{23} + 2^s)) = O(2^s + 2^{2s-23})$$

Note that a lack of memory does result in noticeable performance losses, since if we had significantly more memory (enough to store ciphertext,key pairs for every possible key in the first stage of the meet-in-the-middle attack) we would not need to divide up our key-space and run multiple rounds, bringing the runtime strictly down to $O(2^{s+1})$. This would help increase the s-value for which computation is feasible, although not by much: at large s-values time constraints still make the computation infeasible, even with unlimited memory.

JAVA Program

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map.Entry;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class q2 {

private static int s = 29;
private static Cipher cipher;

/**
 * @param args
 * @throws FileNotFoundException
 */
public static void main(String[] args) throws Exception {
cipher = Cipher.getInstance("AES/ECB/NoPadding");

String keyf = "C:/Users/victorw/Documents/6.857/group7/k-"+s+".bin";
String keypf = "C:/Users/victorw/Documents/6.857/group7/kp-"+s+".bin";
FileOutputStream fosk1;
  FileOutputStream fosk2;
  File file;
  try{
    file = new File(keyf);
    fosk1 = new FileOutputStream(file);
    file = new File(keypf);
    fosk2 = new FileOutputStream(file);
  }catch(FileNotFoundException fe){
    System.out.println("could not open output files for writing, exiting");
    return;
  }
byte[] ciphertext;
byte[] plaintext;
String filenameC = "C:/Users/victorw/Documents/6.857/group7/ciphertext-"+s+".bin";
String filenameP = "C:/Users/victorw/Documents/6.857/group7/plaintext-"+s+".bin";
file = new File(filenameC);
long length = file.length();
ciphertext = new byte[(int) length];

FileInputStream fb = new FileInputStream(file);

fb.read(ciphertext);

file = new File(filenameP);
length = file.length();
```

```
fb = new FileInputStream(file);
plaintext = new byte[(int) length];
fb.read(plaintext);

Integer i = 0;

System.out.println("The length of ciphertext " + ciphertext.length);
System.out.println("The length of plaintext " + plaintext.length);
System.out.println("Math.pow(2, " + s + ") = " + Math.pow(2, s));
double iters = Math.pow(2, s)/Math.pow(2, 23);
outer:
for (int j = 0; j < iters; j++) {
  System.out.println("PHASE j = " + j);
  HashMap<String, byte[]> keylist = new HashMap<String, byte[]>();
  System.gc();
  int startval = j;
  boolean overflow = false;
  // encrypt first

  for (int x = startval; x< Math.pow(2, s); x += iters) {
    byte[] key = getkey(x);
    if (x % 10000000 == 0) {
      System.out.println(x);
    }
    if (x == startval && !overflow) {
      overflow = true;
    } else if (x == startval && overflow) {
      break;
    }

    SecretKeySpec skeySpec1 = new SecretKeySpec(key, "AES");

    // Instantiate the cipher
    cipher = Cipher.getInstance("AES/ECB/NoPadding");
    cipher.init(Cipher.ENCRYPT_MODE, skeySpec1);
    byte[] encrypted = cipher.doFinal(plaintext);

    keylist.put(new String(encrypted), key);
  }

  System.out.println("ENCRYPTION PHASE COMPLETE");
  overflow = false;
  for (int x = 0; x < Math.pow(2,s); x++) {
    if (x % 10000000 == 0) {
      System.out.println(x);
    }
    if (x == 0 && !overflow) {
      overflow = true;
    } else if (x == 0 && overflow) {
      break;
    }

    byte[] key = getkey(x);
    SecretKeySpec skeySpec2 = new SecretKeySpec(key, "AES");
    cipher.init(Cipher.DECRYPT_MODE, skeySpec2);
    byte[] decrypted = cipher.doFinal(ciphertext);

    String decStr = new String(decrypted);
    if (keylist.containsKey(decStr)) {
      byte[] storedKey = keylist.get(decStr);
      System.out.println("Found a potential match");
      SecretKeySpec skeySpec1 = new SecretKeySpec(storedKey,"AES");
      cipher.init(Cipher.ENCRYPT_MODE, skeySpec1);
      byte[] middle_stage = cipher.doFinal(plaintext);
      skeySpec2 = new SecretKeySpec(key, "AES");
      cipher.init(Cipher.ENCRYPT_MODE, skeySpec2);
      byte[] complete = cipher.doFinal(middle_stage);

        if (byteArraysEqual(ciphertext, complete)) {
          try {
            fosk1.write(storedKey);
            fosk2.write(key);
          } catch (IOException e) {
            System.out.println("Could not write keys");
          }
          System.out.println("Match is valid: k = " + getInt(storedKey));
          System.out.println("\nk’ = " + getInt(key));
          break outer;
        } else {
          System.out.println("Match is no good");
        }
      }
    }
    System.out.println("DECRYPTION PHASE COMPLETE");
  }
  System.out.println("DONE");
}

private static byte[] getkey(int value) {

  byte[] b = new byte[16];
  b[12] = (byte) (value >>> 24);
  b[13] = (byte) (value >>> 16);
  b[14] = (byte) (value >>> 8);
  b[15] = (byte) value;

  return b;

}
```

```
private static int getInt(byte[] key){
  int r = 0;
  r += key[12]*Math.pow(256,3);
  r += key[13]*256*256;
  r += key[14]*256;
  r += key[15];
  return r;

}

private static boolean byteArraysEqual(byte[] a, byte[] b) {
  if (a.length != b.length)
    return false;

  for (int i = 0; i < a.length; i++) {
    if (a[i] != b[i])
      return false;
  }

  return true;
 }
}
```

Python verify keys program

This program is used to validate K_{29} and K'_{29} in Python.

```
import Crypto.Cipher.AES as AES
import struct
import copy

def testKeyFromFile(plain,cipher,key,keyPrime):
    fp = open(plain,"rb")
    p = fp.read()
    fp.close()

    fc = open(cipher,"rb")
    c = fc.read()
    fc.close()

    fk = open(key,"rb")
    k = fk.read()
    fk.close()

    fkp = open(keyPrime,"rb")
    kp = fkp.read()
    fkp.close()
    aes1 = AES.new(k)
    aes2 = AES.new(kp)

    mid = aes1.encrypt(p)
    final = aes2.encrypt(mid)

    print pretty_print_byte_string(k)
    print pretty_print_byte_string(kp)
    print pretty_print_byte_string(p)
    print pretty_print_byte_string(mid)
    print pretty_print_byte_string(c)
    print pretty_print_byte_string(final)

    if final == c:
        print "keys worked"
    else:
        print "keys failed"

def testKey(plain,cipher,key,keyPrime):
    fp = open(plain,"rb")
    p = fp.read()
    fp.close()

    fc = open(cipher,"rb")
    c = fc.read()
    fc.close()

    aes1 = AES.new(ints_to_string(key))
    mid = aes1.encrypt(p)

    aes2 = AES.new(ints_to_string(keyPrime))
    final = aes2.encrypt(mid)

    print pretty_print_byte_array(key)
    print pretty_print_byte_array(keyPrime)
    print pretty_print_byte_string(p)
    print pretty_print_byte_string(c)
    print pretty_print_byte_string(final)
    if final == c:
        print "keys worked"
    else:
        print "keys failed"

# pretty_print functions from aes_utils.py
```

6.857 Homework	**Problem Set 2**	# 2-3 - Ben's Block Cipher
Victor Williamson		
Victor Chan		
David Feldman		March 2, 2009

(a) Consider a given key, (p,S,q). To be more concrete, let us first define our permutations p and q to be lists of numbers of length 128 such that each number in 0...127 is represented once (it is a one-to-one mapping). If a 128-bit input is fed into a permutation, say p, then the output P[input] is defined such that output[k] = input[p[k]] for k in 0...127. Let us define S as a one-to-one mapping from $[0, 1]^8 \Rightarrow [0, 1]^8$.

Now take p and chop it into groups of 8 bits, starting from the lsb (bit zero). Since S only operates on groups of 8 bits, we can permute these groups without changing the mappings being processed by the s-box. If we similarly chop q into 8-bit chunks and permute those in the exact same way that we permuted p, then the output bits of S will end up in the same location as in the original key. A 128-bit permutation is composed of 16 groups of 8 bits, and thus we can permute p and q in a byte-wise fashion to produce 16! equivalent keys without modifying S.

Now that we have permuted the bytes, let us consider the bit-ordering inside the bytes. A short example might be illustrative: if, in every byte-sized chunk of p, we switch the first bit (i.e. bit zero) and the last bit (bit 7), then we can produce an equivalent key solely by swapping the first and last bits of every input key in the mapping defined by S. To generalize, we can permute the 8 bits that comprise each byte of p in any way that we would like and still end up with an equivalent key, so long as this 8-bit permutation is the same in each byte of p, and so long as we permute the bits of the byte-sized input keys of S in the same way as we permuted the bytes in p.

Similarly, we can permute the 8-bit output values of S in any way that we would like and obtain equivalent keys, so long as every output value is permuted in the same way and so long as we do the same permutation on every byte in q.

These 3 ways to produce equivalent keys, p/S_{in} byte-swapping, p/S_{in} bit-swapping, and q/S_{out} bit-swapping, do not overlap. Thus, since there are 8! ways to permute 8 bits, we haved found $16! * (8!)^2$ equivalent keys to (p,S,q).

(b) It's important to note that there are many equivalent keys. Thus we can arbitrarily pick certain properties, and then make everything else work out to be consistent.

Here is a way to use a plaintext attack to break this cipher:

Encrypt 0x0h and 0x1h and compare the ciphertexts. We know that S is 1-to-1, and so bits in these 2 ciphertexts should differ in at least 1 and in at most 8 positions. Now encrypt the other 127 plaintexts where there is a single one bit (i.e. 0x2h, 0x4h, 0x8h, 0x10h,etc). We can similarly compare these ciphertexts to the 0x0h ciphertexts, and mark at which 8 positions the bits differ. Since each bit is part of 1 of the 16 bytes, there should be 16 groups of 8 ciphertexts where the differing bits overlap. Hopefully each group of 8 should in total have 8 bits that differ from the 0x0h ciphertext. It is possible, albeit unlikely, to have less (the minimum is 4). If this is the case, we would look to see which bits in the input triggered differing bits in the output, and then encrypt more plaintexts where we set multiple of these input bits at a time to 1, looking for more differences from 0x0h until we have 8 bits in each group.

Thus what we have determined is which bits belong together in a byte after p but before q.

We do not know the ordering the different bytes after p, just which bits are part of a single byte. However, note from part a that if we swap the ordering of the bytes produced by p, we can create an equivalent key merely by doing a similar swap on q. Thus, we select an arbitrary ordering of the 16 bytes we found, and use this ordering to begin to partially define p and q.

We also do not yet know how the bits are ordered inside each of the 16 bytes (we just know which bits belong to which byte). Once again, recall from part a that if we alter the inner ordering we can make an equivalent change in the s-box. Now consider the ciphertexts we produced previously for 0x0h,0x1h,0x2h,etc. These ciphertexts can be divided into 9 distinguishable types. After the s-box but before q, 0x0h produces 16 copies of the 0x00h byte mapping, while the others produce 15 copies of the 0x00h byte mapping and 1 copy of the mapping from either 1,2,4,8,16,32,64, or 128 decimal. The later 8 types (i.e. not 0x0h) all have 16 members, representing a common position in the byte. Due to the key equivalence, we can arbitrarily select one of these types to represent the 1st bit, another the 2nd bit, another the 3rd bit,..., and the last the 8th bit. Given these bit groupings, we have fully established p.

We can similarly use key equivalence to establish q as follows. Look at the 0x0h ciphertext. Count all of the ones and all of the zeros and divide each count by 16. We now know how many ones and how many zeros are in the output value defined by S for the input 0x00h. These counts are the 'unique' information; the ordering is arbitrary since we can then adjust q to be consistent. Thus we just pick some bit ordering and let it be the official mapped value for 0x00h (perhaps the ordering created by looking at the 1st byte and reading the bits left to right). We already knew the ordering of the bytes coming out of the s-box (we chose it earlier), and we now know the identity of the byte coming out of the s-box for the input 0x00h. Thus we can just match up the bits inside the bytes to the corresponding bits in the 0x0h ciphertext. This completely defines q.

Since we have established p and q, we can easily enumerate each key,value pair in the s-box by looking at the plaintext/ciphertext pairs for 0x00h-0xFFh, tracing from the input to just before the s-box using p and from the output to just after the s-box using q.