Homework 5

CSC 152 / 252 – Cryptography
Due: 11:59pm, Wednesday, November 14
(Program may be turned in 72 hours late without penalty)

Read the document "Homework procedures" posted in Piazza resources for how to turn in your homework and policies on collaboration. A small sampling of your written homework and most of your programs will be graded over the semester. To minimize the chance that bad luck causes you a bad grade, do every problem to the best of your ability. Show your work. Ask questions if you need help.

What to turn in: Program file named exactly `hw5_siv.c` and a single file with your written solutions named `hw5.pdf`. Submit to `http://dbinbox.com/cscx52/xxxx` where `xxxx` should be replaced by your four-digit code. Verify submission using the DBInbox link sent to you earlier in the semester. Mis-submitted or mis-named files may receive no credit.

References: Chapters 8 and 10 from *Serious Cryptography*.

Written Problems:

1) Recall that the extended GCD algorithm goes like this.

```
When calculating the GCD of a and b, repeatedly do the following:
   Rewrite egcd(x, y) as egcd(y, r) where x = qy + r for some 0 <= r < y
   Solve for r:  r = x + (-q)y
   Substitute combinations of a and b for x and y, and simplify
```

At each iteration of the algorithm, you get a new r as a combination of a and b, which can be used in later iterations for substitution. The algorithm terminates when $r = 0$, meaning that y is the GCD.

Follow this process to find egcd(59,55) and format your intermediate results as seen in class.

2) What is $55^{-1} \bmod 59$. How do you know? In other words, complete and explain the steps needed to go from your answer of Problem 1 to the answer of this problem.

3) In a prior homework you needed to compute $(x^3 + x^2 + x + 1)^{-1} \bmod x^4 + x + 1$. Use the egcd algorithm to compute it.

4) Let $p = 367$ and $q = 373$ be randomly chosen primes. Use them to produce a public and private RSA key. When it comes time to pick e, choose the smallest value greater than 1 that qualifies. When it comes time to find an inverse, use the extended GCD algorithm to find it. Use your public key to encrypt 5, and show that your private key returns 5 when decrypting the result.

5) In class we saw an exponentiation algorithm that runs in time proportional to the log of the exponent. Follow that algorithm to compute $12^{13} \bmod 13$. Mod each of your intermediate values to keep them from getting too big. Format your computation like so, to compute $3^5 \bmod 10$:

$$\begin{aligned} 3^1 &= 3 \\ 3^{10} &= 3^2 = 9 \\ 3^{100} &= 9^2 = 81 = 1 \\ 3^{101} &= 3 \cdot 1 = 3 \end{aligned}$$

Begin with $base^1 = base$ and each time you square or multiply write a line showing the new exponenet in binary, the squaring or multiplying step carried out and simplified.

Programming:

P1) In class we saw that one goal of authenticated-encryption is the safe use of a single key for both encryption and authentication. We also learned of a mode called SIV which makes the reuse of a nonce when encrypting two plaintexts non-catastrophic. This is done by creating a "synthetic IV" for CTR-mode encryption that is a pseudorandom value determined by both the plaintext and the nonce. Only if both these things are repeated will the same IV be used for encryption. In fact, if it's known ahead of time that plaintexts can't repeat, the nonce can be omitted entirely.

In this programming assignment, you will use functions you have already written to implement an SIV mode. Here is what I'll call PCM-SIV (Poly-CTR-Mode-SIV) in pseudocode.

```
PCM-SIV-Encrypt(k: a 16 or 32 byte key
                n: a 0 to 16 byte nonce
                p: a plaintext of any number of bytes)
    keys = P52-BC(k, <48 bytes of 0xFF>)
    hashKey1 = readLE(keys[0..7]) >> 4          // Makes high 4 bits zero
    hashKey2 = readLE(keys[8..15]) >> 4         // Makes high 4 bits zero
    hash1 = write8bytesLE(Poly61(hashKey1,p)))  // Hash twice to reduce collision probability
    hash2 = write8bytesLE(Poly61(hashKey2,p)))  // Hash twice to reduce collision probability
    siv = P52-BC(k, hash1 || hash2 || nonce || 10*)
    c = P52-CTR(k, siv[0..15], p)
    t = siv[0..15]                              // The synthetic IV serves as authentication tag
    Receiver needs (c, n, t) to decrypt
```

A single key is used with every invocation of PC-BC, but the inputs to PC-BC are carefully chosen to disallow the same input twice. This means it's safe to use the same key for hash-key generation, SIV creation, and counter encryption.

The notation $s[a..b]$ is the bytes with indices a through b, inclusive. Concatenation is indicated "$||$". The read and write LE are there to make reading and writing Poly61's key and result easy on little-endian architectures. P52-CTR(k, n, p) is counter-mode encryption of p with key k and nonce/IV n. The counter block should begin with n followed by as many zeros as needed to make the block 48 bytes, and the resulting block should be incremented big-endian (ie, the far-right bytes gets incremented each time). P52-BC(k, x) is defined as $(k||0^*) \oplus \text{P52}(x \oplus (k||0^*))$.

Write a function with the following signature.

```
// Encrypt p into c and write authentication tag to t
void pcm_siv_encrypt(unsigned kbytes, unsigned char k[kbytes],  // Must be 16 or 32
                     unsigned nbytes, unsigned char n[nbytes],  // Must be 0 to 16
                     unsigned pbytes, unsigned char p[pbytes],
                     unsigned char c[pbytes], unsigned char t[16])
```

You may like the satisfaction of writing this entire function using your own functions. If you do so, copy your functions into the single file you submit and rename the functions something that won't conflict with the following names: poly61, P52_BC, P52_CTR, pcm_siv_decrypt. I will compile your submission with a file that defines all these, so if you use these names too, an error will occur. A library with these four functions will be made available on apollo.ecs.csus.edu for your use in testing and development (details to be posted in Piazza Q&A soon).

Compiling) Put your code into the file name(s) specified above and submit via DBInbox. Your file(s) should include the required function and should not include `main` unless specified to do so, should

not print anything unless specified to do so, should be appropriately documented, and should compile without warning or error when compiled using `gcc` or `clang` with compiler options `-std=c99 -Wall -Wextra -pedantic -c`. The only headers your file is allowed to include are the standard ANSI C headers (and `emmintrin.h` if needed).

As an extra check, I may compile and run your code using Clang's `-fsanitize=address` command line option which looks for your program writing to memory it should not write to. Apollo and Macs have Clang, so you may want to test your code on one of those machines. The best way to test your program is on Apollo or a Mac with the following command line:

```
clang -std=c99 -Werror -Wall -Wextra -pedantic -fsanitize=address
```

If your code compiles without problem, doesn't produce any error reports when running, and produces correct results, then your code is in good shape.