Topics in C programming useful for cryptography

C was designed by Dennis Ritchie in the 1970s as a systems programming language, meaning it was designed to be able to do a lot of the low-level operations that otherwise would need assembly language. With C you can do things like assign an arbitrary memory address to a pointer and then read and write bytes at any offset from that address. This is exactly the type of programming needed in cryptography, and is why C is the language of choice for this class.

Data datatypes:

In cryptographic programming, we manipulate data that resides in memory and looks like random bits. This means that the programs we write won't be manipulating numbers and characters like you are probably used to, but will instead manipulate arbitrary bit sequences.

Luckily the C header file `<stdint.h>` defines several types designed to make data manipulation easier. It defines the types `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`, which hold 8, 16, 32, and 64 bit quantities. Since memory on typical computers is "byte addressable" (meaning each different memory address refers to a single byte), we use uint8_t as the fundamental data type when interfacing with memory. That is, we will consider memory to be essentially a big array of uint8_t.

Technically a uintNN_t variable is an unsigned integer of NN bits, ranging from 0 to 2^{NN}-1, and an intNN_t is a signed integer of NN bits, ranging from -2^{NN-1} to 2^{NN-1}-1. In our programs we will use `int` for typical integer purposes like loop counters, and use the uintNN_t types for holding binary data. Only occasionally will we use uintNN_t or intNN_t types to represent anything else. This is consistent with industry practice; for example it is what Google recommends in its C++ style guide.

The main reason we use unsigned types for binary data is because it avoids "sign extension" (a feature that inserts extra 1 bits to maintain the sign of a signed integer when it is manipulated in certain ways).

Memory Buffers:

Cryptography works on regions of memory. For example, an application whose job is to encrypt a file will read the file into memory, encrypt the memory, and then write the encrypted memory back to a file. This means that the cryptography itself is focussed on manipulating memory.

In C, data in memory is accessed via the data's address in memory, which is typically held in a variable with a pointer type. For example `uint8_t *buf` is a variable named buf holding a memory address. Dereferencing `*buf` allows reading and/or writing the value that buf refers to in memory, and since buf's type is declared to point to uint8_t, it is a uint8_t that gets read or written.

Because students are much more comfortable with array notation than with pointer notation, I will typically use it to read and write to memory. When square brackets are applied to a pointer, the pointer behaves just like the base name of an array: `buf[0]` accessess the uint8_t buf points at, `buf[1]` accesses the next uint8_t in memory, `buf[2]` accesses the uint8_t after that, etc.

Note that pointers and arrays in C do not carry any length information, which means that if a function needs to know how long a memory buffer is, it must be passed as a separate parameter. Here's a function that shifts all the bytes in a memory buffer to an address one smaller but moves the original first byte to the end.

```
void rotate(uint8_t *buf, int num_bytes) {
    if (num_bytes > 1) {
        uint8_t temp = buf[0];
        for (int i=1; i<num_bytes; i++) {
            buf[i-1] = buf[i];
        }
        buf[num_bytes-1] = temp;
    }
}
```

Pointer arithmetic

If p is a pointer, then p[i] is the i-th element in memory, offset from the basename p. If you want the address of the i-th element in memory, offset from the basename p, use p+i. This is called "pointer arithmetic". Let's say we had the following definitions and that the memory used for "a" begins at address 0x1000 and the memory used for "b" begins at address 0x2000.

```
uint8_t a[] = {1,2,3,4,5};
uint16_t b[] = {1,2,3,4,5};
```

Then a[2] == 3 and b[3] == 4. Using pointer arithmetic, a+2 evaluates to the address of the item a[2] which is 0x1002 since each element of a is 1 byte. Since each element of b is 2 bytes, b+3 evaluates to the address of the item b[3] which is 0x2006.

Bitwise operations:

The operands `|, &, ^, ~` are "bitwise" or, and, exclusive-or, negation. This means that each bit is considered true (1) or false (0) and bits in the same relative positions are combined using the operation. For example, `a|b` logically or's the bits in `a` and `b`

```
uint32_t a = 5;     // 5 is 101 in binary
uint32_t b = 6;     // 6 is 110 in binary
uint32_t c = a|b;   // 101 or'ed with 110 bit-by-bit yields 111
prtinf("%u", c);    // Prints 7
```

The same process can be used with `&` or `^` to get the "and" or "xor" of two values (c would be 100 if we replaced `|` with `&` in the above snippet or 011 if we replaced `|` with `^`). `~c` negates all of the bits, turning 0 to 1 and 1 to 0.

Note: %u is the format specifier for 'unsigned int' which is what uint32_t usually is defined as. If you want to use the format specifier defined specifically for uintNN_t, you need to include `<inttypes.h>` and look up the right code.

The operations `<<` and `>>` shift bits. When the type being shifted is unsigned, vacated bits are replaced with 0 bits. Those bits being shifted beyond the size of the type are lost.

```
uint32_t a = 5;    // 5 is 101 in binary
uint32_t c = a<<2; // Shifting 101 left two positions yields 10100
prtinf("%u", c);   // Prints 20
uint32_t d = a>>2; // Shifting 101 right two positions yields 1
prtinf("%u", d);   // Prints 1
```

Shifts and bitwise operations can be combined to do all sorts of data manipulation. For example, `(a>>b) & 1` will evaluate to zero if the bth bit of `a` is zero and will evaluate to one otherwise (traditionally bit indices are numbered from the right). Likewise `a | (1<<b)` evaluates to a with its bth bit set to 1 and `a & ~(1<<b)` evaluates to a with its bth bit set to 0. You may need to simulate these operations on paper to understand how they work.

Exclusive-oring two buffers is a common thing to do in cryptography. Here's a simple routine to do it.

```
void xor_buf(uint8_t *dst, uint8_t *src1, uint8_t *src2, int num_bytes) {
    for (int i=0; i<numbytes; i++) {
        dst[i] = src1[i] ^ src2[i];
    }
}
```

Endian:

Intel computers move multi-byte data between memory and registers "little endian". It's called little-endian because a pointer to memory refers to the little end of the value being read (ie, the least-significant byte). When loaded from memory, the byte order is reversed to make the least-significant byte be on the right of the resulting register.

Let's say that `p` is a pointer to memory and the bytes in memory beginning at `p` are `0x01 0x23 0x45 0x67 0x89 0xAB`. Then if we dereference `p` to load one byte we get `0x01` in the target register. If we load a two byte value to a register little endian, the register will be `0x23 0x01`. A four-byte read little endian will result in `0x67 0x45 0x23 0x01`. The byte order is reversed by the hardware. Note that storing from a register to memory also causes byte-reversal when done little endian, so if a register has `0x67 0x45 0x23 0x01` and the register is stored to the memory address in `p`, then the four bytes beginning at `p` will be set to `0x01 0x23 0x45 0x67`.

This is transparent in most programs because programs usually load and store the same size data via any particular pointer. This means that when a value in a register gets stored to memory little endian and later gets reloaded little endian, the register is back to its original value. But, if data is placed in memory via another source, like a network or file read, then the data can end up in a register in backward order.

```
uint8_t memory[] = {0x01, 0x23, 0x45, 0x67};  // 0x01, 0x23, 0x45, 0x67 in memory
uint32_t *ptr = (uint32_t *)memory;   // put array's address into ptr
uint32_t x = ptr[0];                  // register x == 0x67452301 on little-endian
x = x+1;                              // register x == 0x67452302 on little-endian
ptr[0] = x;                           // memory is now 0x02, 0x23, 0x45, 0x67
```

Java and a few other computers use "big endian" reading and writing, which means that a pointer points to the most-significant byte of a multi-byte read (the "big end"). This means loads and stores don't reverse the bytes. Here's the same example but on a big endian computer.

```
uint8_t memory[] = {0x01, 0x23, 0x45, 0x67};  // 0x01, 0x23, 0x45, 0x67 in memory
uint32_t *ptr = (uint32_t *)memory;   // put array's address into ptr
uint32_t x = ptr[0];                  // register x == 0x01234567 on big-endian
x = x+1;                              // register x == 0x01234568 on big-endian
ptr[0] = x;                           // memory is now 0x01, 0x23, 0x45, 0x68
```

So, as you can see, the same C code will produce two different results if data is placed in memory one-byte at a time but then read four bytes at a time.

To avoid this problem we will write our code to be "endian neutral". We will only read data in the same granularity it was written. So, if something was written byte-by-byte to memory, such as from a file or network read, we will also read it byte-by-byte from memory. This means that if data was written byte-by-byte and we need to read it as a four byte quantity, we will read the four bytes separately and assemble them using bitwise operations into the desired byte pattern.

For example, if we explicitly want to load four bytes from memory into a uint32_t following the little-endian pattern we could do it like this:

```
uint32_t load_uint32_little(void *p) {
    uint8_t *p8 = (uint8_t *)p;  // make buffer behave like an array of bytes
    uint32_t a = p8[0];   // least significant byte of a is byte from p[0]
    uint32_t b = p8[1];   // least significant byte of b is byte from p[1]
    uint32_t c = p8[2];   // least significant byte of c is byte from p[2]
    uint32_t d = p8[3];   // least significant byte of d is byte from p[3]
    return (d<<24) | (c<<16) | (b<<8) | a;  // shift each byte into place
}
```

Taking a uint32_t variable and storing it little-endian can also be specified similarly.

```
void store_uint32_little(uint32_t x, void *p) {
    uint8_t *p8 = (uint8_t *)p;  // make buffer behave like an array of bytes
    p8[0] = (uint8_t)x;          // typecast keeps least significant byte
    p8[1] = (uint8_t)(x >> 8);   // typecast keeps least significant byte
    p8[2] = (uint8_t)(x >> 16);  // typecast keeps least significant byte
    p8[3] = (uint8_t)(x >> 24);  // typecast keeps least significant byte
}
```

On a little endian computer `load_uint32_little` could be written more simply as `return ((uint32_t *)p)[0]`, which casts p to the right type and then dereferences it to get the data. You might wonder why is this other way better. (1) It is endian neutral (ie, works on both big and little endian computers); (2) it is safe on computers that care whether p is "aligned" to a multiple of 4 (some computers can only load n bytes if the load address is a multiple of n); and (3) On a little endian computer this code compiles to assembly code that is just as efficient. Here is `load_uint32_little` compiled using gcc.

```
_load_uint32_little:
   movl  (%rdi), %eax
   ret
```

Typecasting pointers:

A pointer is a variable that contains a memory address and information about what type of data it points to. (Array basenames contain the same information and so pointers and array basenames can usually be used interchangeably.) If you want to treat a pointer as a pointer to a different type of data you typecast it. So, if we declare and define

```
uint8_t arr[] = {1,2,3};
```

then arr is a variable holding a memory address, the memory beginning at that address contains the bytes `0x01 0x02 0x03`. Since arr is declared to refer to uint8_t then by default accesses via arr will be uint8_t.

```
uint8_t x = arr[2];  \\ sets x to 3
uint8_t y = *arr;    \\ sets y to 1
```

If, however, we want to read a different type via arr we need to typecast it.

```
uint16_t *p = (uint16_t *)arr;     // Copies address arr into p
uint16_t x = *p;                   // sets x to 0x 02 01  (on little-endian)
uint16_t y = p[0];                 // sets y to 0x 02 01  (on little-endian)
```

It is good practice to avoid typecasting because it subverts the type system, but in cryptography it is often necessary because memory is written as sequence of bytes and we sometimes need to load data four bytes at a time.

Example File Processing:

Here is a program that opens a file named "input.dat" and prints the XOR of all the bytes in the file.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define BUF_BYTES 128

uint8_t xor_buf(uint8_t *buf, int nbytes) {
    uint8_t acc = 0;
    for (int i=0; i<nbytes; i++)
        acc ^= buf[i];
    return acc;
}

int main() {
    uint8_t buf[BUF_BYTES];
    FILE *f = fopen("input.dat", "r");
    if (f != NULL) {
        uint8_t acc = 0;
        int bytes_read  = fread(buf,1,BUF_BYTES,f);
        while (bytes_read > 0) {
            acc ^= xor_buf(buf,bytes_read);  // buf[0..bytes_read-1] just read
            bytes_read = fread(buf,1,BUF_BYTES,f);
        }
        printf("%X\n",acc);
        fclose(f);
    }
    return EXIT_SUCCESS;
}
```

Note that xor is associative and commutative, so just like you could add a, b, c, and d in any order you can xor them in any order too: (a+b) + (c+d) == a+b+c+d, for example. So does (a^b) ^ (c^d) == a ^ b ^ c ^ d. So, in the above application, chunks of bytes are xor'ed and those individual results are xor'd for the final result.

`fread(a,b,c,d)` reads c objects from file d, each object b bytes long, and places the result into the buffer pointed at by a. It returns how many objects were successfully read. When b is set to 1, fread reads c bytes and returns how many bytes were read. You can detect EOF by the return value being less than c.