The University of Melbourne

School of Computing and Information Systems

COMP10002 Foundations of ~~Algorithms~~ Awesome

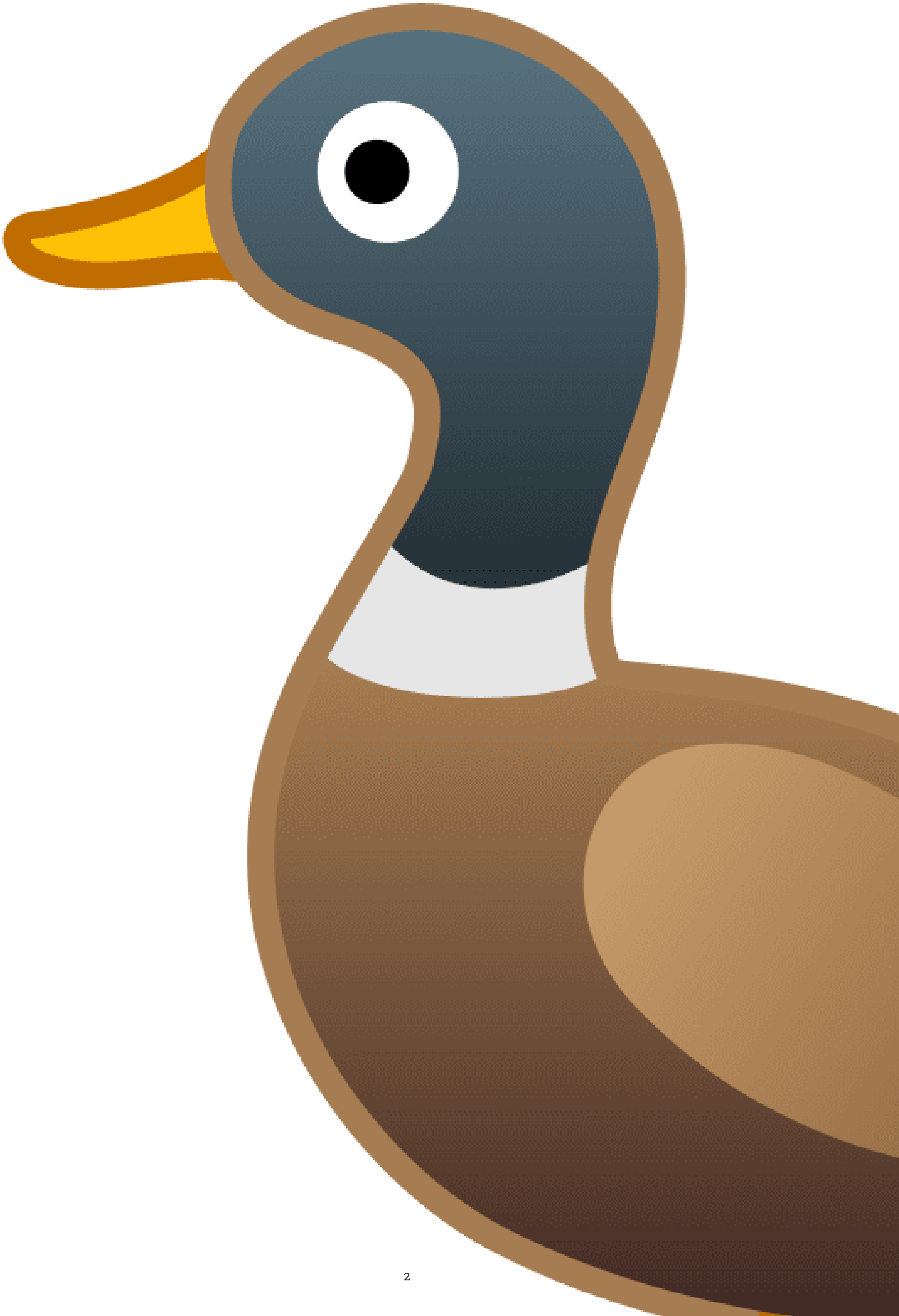
Semester 1, 2021

Assignment 1: The DUHK Attack [*]

Due 4PM, Tuesday 4th May, 2021

Version 1.1

[*]By Shaanan Cohney and Liam Saliba



Change Log

Version 1.1

- Typo: Capitalized some references where they were erroneously lowercase
- Typo: End of Section 4.1, “one call AES” → “one called AES”.
- Typo: End of Section 4.3, “oxC.”→ “`0xCC`” since 204 in base ten is `0xCC`.
- Typo: Locations that used the command `./assignment1` have been changed to use `./program1`
- Clarification: You do not need to read the research paper for this assignment, it is present only for enrichment and enjoyment. It also will not make the assignment easier if you do read it.
- Correction: Section 8, We did not previously provide sample output for the two input files. The latest version of the package contains two additional test files and two matching output files, the same ones used for Grok.

1 Learning Outcomes

In this assignment you will demonstrate your abilities functions, pointers, arrays, loops, and conditionals. The assignment covers the following components of the textbook: Chapters 1 to 7 inclusive, along with material from pages 230-235.

The assignment is intended to be *very challenging* and push your programming skills significantly further than Grok, as well as testing your algorithmic thinking. However, it is also intended to be fun, and a very close analogue to a real world task. The first time I coded the full version of the attack for the research paper took me a long time! You can anticipate that this assignment might also take quite a while, but the payoff will be in your new skills and learning.

The inspiration for this assignment is the research paper "Practical state recovery attacks against legacy RNG implementations" by Shaanan N. Cohney, Matthew D. Green, and Nadia Heninger, with more details at duhkattack.com.

Read all the instructions carefully and make sure you understand what is expected of you

2 Full Story (just for fun)

Watch the video here and read the accompanying description!

3 The (Short) Story

You have intercepted an encrypted message, and know some details about how it was generated. The message was produced by taking the xor of the original message with some key k_1.

This key, k_1, was produced using a pseudorandom number generator at times T_{11} to T_{19}.

However, while you know all the times at which the generator was ever used (T_0 to T_{19}), you only have access to outputs from T_9 and T_{10} (O_9 and O_{10}).

You reverse engineered the generator and found it used a second key, k_2.

Your goal is to reconstruct k_1, and use it to decrypt the original message.[a]

[a]When you succeed, you might notice missing punctuation in the message—we suspect the senders were just saving a few characters ;)

4 Background Material

The following additional material provides context for the assignment itself. You can also watch the videos uploaded to the LMS for a walkthrough of the assignment and the background material.

4.1 Generating Random Numbers

Computers don't know *how* to be truly random. While one can purchase a device to measure very unpredictable phenomena (like radioactive decays) and convert them into random numbers, most computers have no such hardware. Instead, computers rely on algorithms that produce random looking output that is very hard for an outsider to distinguish from output that is truly random. How hard? We say that no algorithm that runs in *polynomial time* (Big O of any polynomial) should be able to tell if the output is truly random or not with anymore than an minute probability. We call these pseudorandom number generators.

Importantly, the sequence of random numbers should also not be guessable. If you see prior output from the algorithm, they should be so "random" that you can't guess what will come next. This is important because we typically use a random generator to make random keys for encryption.

One well known algorithm to generate random numbers is the the following pseudorandom number generator (known as the 'ANSI X9.31 DRBG').

The generator itself relies on having a cryptographic function, normally one called AES that we discuss in a bit more detail in Section 4.5. Because of this choice, our assignment will process 128-bits at a time, the same amount as AES. We represent the encryption of message m under AES with k key as $AES_k(m) = c$.

The X9.31 generator operates as follows:

Step 1: Initialize the *state* of the algorithm with some pre-generated random array V_0 of size 128-bits (think of it as an array of 16 characters). This can be done by taking a value such as the time the computer was turned on, xor'd with other timing values from the startup process.

Then loop starting with $i = 1$ as many times as you need to generate sufficient output

Step 2: Generate an intermediate value $I_i = AES_k(T_i)$ where T_i is the current time

Step 3: Generate 128 bits of output by computing $O_i = AES_k(I_i \oplus V_{i-1})$

Step 4: Calculate the value that will be used in the next iteration $V_i = AES_k(O_i \oplus I_i)$

Eventually you'll end up with an array O that has as many blocks of random numbers as you need.

Note that the value V_i is the only variable from the previous iteration that is used in the next iteration. We therefore call V_i the *state* of the pseudorandom number generator. In this design, if an attacker can figure out the state, they can predict all future outputs.

The values we provide already incorporate the value of V_0, so you will not need to worry about it in the assignment.

4.2 The DUHK Attack

Unfortunately the ANSI X9.31 random number generator has a deadly flaw. If the following conditions hold, an attacker can guess the sequence of random numbers, and therefore know any key that's made up of those numbers:

The attack works as follows for $i = 1$ without loss of generality, assuming the following notation for AES *decryption* $AES_k^{-1}(c) = m$:

By rearranging Steps 2 to 4 for i and $i + 1$ we can get[1]:

$$AES_k^{-1}(AES_k^{-1}(O_{i+1}) \oplus AES_k(T_{i+1})) = O_i \oplus AES_k(T_i) \tag{1}$$

Now assume that our clever attacker:

1. Has a small number of good guesses for the key k
2. Has intercepted O_i and O_{i+1}
3. Knows the values of T_i and T_{i+1}—the times when the two outputs were generated

The attacker now makes lots of guesses for the key k. When they guess the right one, the left side and the right side of the equations will be equal! They now know the key.[2]

Because the attacker now knows the key they can compute the secret *state* of the generator using:

$$V_{i+1} = AES_k(O_{i+1} \oplus AES_k(T_{i+1})) \tag{2}$$

an equation in which they know the value of all the variables on the right hand side.

And, once the attacker knows the state of the generator, they can generate all the subsequent outputs O_{i+2}, O_{i+3} etc., assuming they also know the times at which those are generated.

4.3 Bits and Bytes

To carry out this attack you might want to learn a bit more about how we can store and manipulate all the values used for this sort of cryptography.

Remember from lecture that all values are ultimately represented inside a computer's memory as numbers of some sort. Every number can be written in different 'bases', like decimal (base-10) or hexadecimal (base-16) which we use for addresses, or binary (base-2).

[1]This works because we know the inverse of AES encryption is decryption so we can move an encryption term to the other side where it becomes a decryption term

[2]In the real attack, the adversary learns k by reverse engineering the code, but has decent guesses for the two times. The key k is typically too large to guess.

If we write a number in binary, it will consist of only zeroes and ones, with the rightmost digit corresponding to the one-place, the second-rightmost digit corresponding to the twos-place, then fours-place, eights-place, etc. ascending in powers of two.

This is directly analogous to the places in a base-10 number: the ones place, the tens place, the hundreds place.

For example in the number 204, the number is composed of $2 \cdot 100 + 0 \cdot 10 + 4 \cdot 1 = 2 \cdot 10^2 + 0 \cdot 10^1 + 4 \cdot 10^0$. We could also add a zero in the thousands-place, but the number wouldn't change.

We can write 204 in binary as $11001100 = 1 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$

Similarly the number 5 could be written as $101 = 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1$, or we could add some leading zeroes to the front and write 00000101

Thinking back to what we know about the char type, we know it can store up to 256 values (including zero). To write all these numbers in binary we would need eight ones and zeroes, each of which we call a 'bit', with eight bits making up a 'byte'. So, when we represent a char in binary, we will typically add zeroes to the front until all eight bits are represented.

This assignment will **not** require you to manipulate the bits of any individual number other than using the XOR operation on two *bytes* at a time.

We can do the same thing, just with base-16, instead of base-2, where along with the digits 0-9 we add the letters A-F to our repertoire. We call this the hexadecimal system, or hex for short.

This means after 9, we count A, B, C, D, E, F and then we're out of symbols, so we add a 1 to the next column (the 16^1s), and put a zero in the original column (the 16^0s). So $10 = 1 \cdot 16^1 + 0 \cdot 16^0$ in hex is actually the number 16!

Likewise we could write[3] 204 as $12 \cdot 16^1 + 12 \cdot 16^0 = \texttt{0xC} \cdot 16^1 + \texttt{0xC} \cdot 16^0 = \texttt{0xCC}$. Note that here I prefix hexadecimal numbers with `0x` to indicate to the reader, by convention, that the number is in base-16.

4.4 XOR

One of the operations that we care about more in cryptography is called exclusive-or.

Given two eight-bit numbers represented as a collection of eight bits $i_{0\ldots7}$ and $j_{0\ldots7}$, we can define operations that change the bits in different ways. One of these operations is XOR—short for exclusive-or—which we write using the symbol $\oplus$. We can define XOR in the following manner:

Let $l_{0\ldots7}$ be the output of the $\oplus$ operation.

For each index k from 0 to 7, if $i_k = j_k$ then $l_k = 0$, otherwise $l_k = 1$. We illustrate this in Table 1.

Table 1: Bitwise XOR operation

i_k	j_k	$(i_k \oplus j_k)$
0	0	0
0	1	1
1	0	1
1	1	0

In Table 2 we present the sample calculation of $00110011 \oplus 01010101 = 01100110$. To determine the result for each column, one can use Table 1 on the value in the same position in both of the rows.

It's called exclusive-or because applying the XOR operator bitwise, each bit returns a true value (1) if *only* (exclusively) one of the two input bits was 1.

Note this works for numbers of any bit-length, with the definition adjusted accordingly.

Table 2: Example of Bitwise XOR

	00110011
$\oplus$	01010101
=	01100110

In C we take the XOR of two values with the bitwise XOR operator `^`. For example to compute the XOR of two numbers a and b and place the result in c we write `c=a^b`, where a, b and c are of a type with an underlying integer representation (such as an integer, but including a char—which we recall is just a number in ASCII).

4.5 One-time pad and Basic Encryption

For the very last part of the assignment you'll need to know just a tiny bit more cryptography which we present here.

[3]Here I'm abusing notation to mix two bases in one equation for the purpose of illustration

Symmetric encryption (where both sides start sharing a secret key) obeys the following property: $D_k(E_k(m)) = m$. That is with a message m, encryption algorithm E and decryption algorithm D, both of which use a key k, the decryption-of-the-encryption should be the original message.

A one-time pad is a way of encrypting a message. Given a key that is as many bits in length as the message to encipher, this provides the best theoretical security guarantee of any cipher[4]. To encrypt a message m into a *ciphertext* c with key k the process is simple: for the i-th byte of the message, XOR it with the i-th byte of the key. That is, for all $i \in len(m)$:

$$c[i] = m[i] \oplus k[i]$$

The decryption process is similar: for the i-th byte of the ciphertext, XOR it with the i-th byte of the key. That is, for all $i \in len(c)$:

$$m[i] = c[i] \oplus k[i]$$

Why don't we use this technique everywhere? For the size of the messages that we transmit daily, we'd need to have already sent and stored keys just as long as the messages! This quickly gets unwieldy and infeasible.

Instead we typically use ciphers that aren't as strong but that can use a small key to secure messages that are for all practical purposes, secure. The most widely used (symmetric) cipher is called *AES* (the Advanced Encryption System).[5]

5 Your Assignment

Your solutions must be contained within a C file, `program1.c`.

Your code *must* compile as submitted or you may receive a zero on the assignment.

Your program is to read in input of the following format:

Input Line 1: Length of Encrypted Message in bytes, an integer l with a maximum value of 1024.

Input Line 2: Encrypted Message encrypted using a one time pad under key k_1, which is l characters in length. Each character is represented as a two-digit hexadecimal number in the file (see Lecture 10, minute 15). This corresponds to $b = \lceil \frac{l}{16} \rceil$ groups of 16-characters (blocks), where $\lceil x \rceil$ is the ceiling function. The numbers will not be prefixed with `0x`.

Input Line 3: Outputs O_9 and O_{10} from the random number generator, in groups of 16 characters, where each character is represented as a two-digit hexadecimal number in the file.

Input Line 4: Timesteps T_0 through T_{10} used to generate O_0 to O_{10}, in groups of 16-characters (blocks), where each character is represented as a two-digit hexadecimal number in the file.

Input Line 5: The first 1284 characters in the text of Sherlock Holmes (or some other novel!), the cipher book.

Your job is to read in the text of the book, and find which 16-character block has been used as the key k_2 for the random number generator.

You will then generate enough output from the random number generator to produce k_1, which is composed entirely of this random output.

Finally, you will decrypt the original message using the key k_1.

Along with this spec we are providing you with skeleton code to fill in, a set of libraries to accompany the scaffold and files containing the input data. See Section 6 for more details.

We now provide information about the stages in greater detail.

Stage 0: Reading the Input File

Your first task is to read in the input file (from `stdin`, the standard input), and parse each line correctly, as described above.

This is typically done (outside Grok) with a command similar to this one:

[4] As discussed in lecture, it was Claude Shannon who provided the first formal proof of this fact!

[5] Please do not use the techniques in this assignment to encrypt data in real life. While the algorithms used are very similar, there are a large number of caveats to learn about first.

```
./program1 < assignment1-input1.txt
```

When the program tries to read input, it receives the contents of the file. You can always open up a file in your text editor to see what's inside, and we encourage you to do so for this assignment.[6]

Do not use the facilities in C to open the input file within your code. While this is often good practice, we have not learned it yet in class. Therefore, your submission must take in input as described above.

You **must** store your results in the following variables, available as arguments in the function `stage0`:

- `int *ciphertext_length` which should be set by pointer to the length of the ciphertext (Line 1),
- `msg_t ciphertext` which should contain the ciphertext (Line 2),
- `block_t outputs[]` which should contains the two outputs from the random number we provide (Line 3),
- `block_t timesteps[]` which should contains all the timesteps (Line 4), and
- `book_t cipherbook` which should contain the text of the cipher book. (Line 5).

These values are passed to the `submit_stage0` function called in `main`. You will not be able to pass the tests on Grok if you leave extraneous `printf` statements in your code. This will present you with debugging output in the following format (truncated for brevity):

```
Stage 0
==========
Length of encrypted ciphertext (bytes): 128
Encrypted ciphertext, as hexadecimal bytes and ASCII: (below)
0x0000: 0c 6a 03 63 c3 7c 93 d6  48 89 81 27 6b d3 86 bb  ·j·c·|··H··'k···
                                    ...
0x0070: 00 74 cc 54 64 74 58 0e  5f 2f 86 4f 21 da 92 e7  ·t·TdtX·_/·O!···

Outputs, as hexadecimal bytes and ASCII: (below)
  O_ 9: a9 90 e4 e7 f0 d7 1d 0f  d3 67 62 ba 1e 65 8f 84  ·········gb··e··
  O_10: 85 99 d6 a1 93 c5 30 01  e0 ec 1b 69 ed 0e 20 e8  ······0····i·· ·

Timesteps, as hexadecimal bytes and ASCII: (below)
  T_ 0: 01 3c 6c 60 52 86 02 00  01 3c 6c 60 52 86 02 00  ·<l`R····<l`R···
  T_ 1: 01 3c 6c 60 37 9a 02 00  01 3c 6c 60 37 9a 02 00  ·<l`7····<l`7···
                                    ...
  T_19: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ················

Cipherbook: (below, wordwrapped to 80 chars)
To Sherlock Holmes she is always THE woman. I have seldom heard him mention her
under any other name. In his eyes she eclipses and predominates the whole of her
                                    ...
 ifi
```

Your code's output should **exactly match** the sample output given for the input file.

Stage 1: Stripping Punctuation

Produce an array that consists of only the alphanumeric values from the book's text on Line 5.

You may **not** use any functions from `string.h` or `ctype.h`.

You **must** store your results in the following variables, available as arguments in the function `stage1`:

- `book_t cipherbook[]` which should (after your completed implementation) contain only the *alphanumeric* values from the book, and
- `int *book_len` which should be set by pointer to be the length of the cipher book after you have removed all non-alphanumeric values.

These values are passed to the `submit_stage1` function called in `main`. This will submit your result for stage 1 for grading, as well as giving you a debug output with the format (truncated for brevity):

[6] Some files store data intended to be read only by a computer, so a human may not be able to understand and read every file on a computer.

```
Stage 1
==========
Punctuation stripped cipherbook length: 1024
Punctuation stripped cipherbook, as hexadecimal bytes and ASCII: (below)
0x0000: 54 6f 53 68 65 72 6c 6f  63 6b 48 6f 6c 6d 65 73  ToSherlockHolmes
0x0010: 73 68 65 69 73 61 6c 77  61 79 73 54 48 45 77 6f  sheisalwaysTHEwo
                                ...
0x03f0: 68 65 68 6f 6d 65 63 65  6e 74 72 65 64 69 66 69  hehomecentredifi
```

Your code's output should **exactly match** the sample output given for the input file.

Stage 2: Guessing the key k_2

You will need to implement a loop to iterate through the stripped book you generated in stage 1, in blocks of 16-bytes. For each block, generate both sides of eq. (1). above and check if both sides match. When both sides match, you have discovered the correct set of 16 bytes that comprises k_2

You **must** store your result in the following variable, available as arguments in the function `stage2`:

- `block_t key2` which should contain the 16 characters that make up k_2.

This value is passed to the `submit_stage2` function called in `main`. This will submit your result for stage 2 for grading, as well as giving you a debug output with the format:

```
Stage 2
==========
Key k2: 76 65 72 65 78 63 65 6c  6c 65 6e 74 66 6f 72 64  verexcellentford
```

Your code's output should **exactly match** the sample output given for the input file.

For this stage you will need to use the `AES` functions we provide, whose use we describe in ??.

Stage 3: Generating the key k_1

You will need to generate outputs O_{11}, O_{12}, ..., O_k until you have generated n bytes of output where n is the length in characters of the encrypted message on Line 2. Remember each O is 16-bytes in length.

Write a function that implements the random number generator, and use it to generate output of length n. This output is k_1. To do so you will need to write code that performs Steps 1 to 4 listed in section 4.1.

This will require you to first calculate V_{10}, the initial state of the generator, which you should be able to compute from O_{10}, I_{10} and k_2 using eq. (2).

You **must** store your result in the following variable, available as arguments in the function `stage3`:

- `byte_t key1[]` which should contain the n characters that make up k_1.

These values are passed to the `submit_stage3` function called in `main`. This will submit your result for stage 3 for grading, as well as giving you a debug output with the format:

```
Stage 3
==========
Key k1, as hexadecimal bytes and ASCII: (below)
0x0000: 58 2f 50 37 97 39 c0 82  1c cc d2 73 3f 96 d5 ef  X/P7·9·····s?···
0x0010: 78 06 f3 fb d5 64 04 a3  5c 6f 2a bd 98 66 49 f1  x····d··\o*··fI·
                                ...
0x0070: 54 31 9f 00 30 31 0b 5a  0b 6a d5 1b 75 9f c1 b3  T1··01·Z·j··u···
```

Your code's output should **exactly match** the sample output given for the input file.

Stage 4: Decrypting the original message

Iterate over both k_1 and c the encrypted message taking the xor to produce the original unencrypted message and solve the caper!

You **must** store your result in the following variable, available as arguments in the function `stage4`:

- `byte_t plaintext[]` which should contains the decrypted message.

These values are passed to the `submit_stage4` function called in `main`. This will submit your result for stage 4 for grading, as well as giving you a debug output with the format:

```
Stage 4
==========
Decrypted ciphertext, as hexadecimal bytes and ASCII: (below)
0x0000: 54 45 53 54 54 45 53 54  54 45 53 54 54 45 53 54  TESTTESTTESTTEST
0x0010: 54 45 53 54 54 45 53 54  54 45 53 54 54 45 53 54  TESTTESTTESTTEST
                                       ...
0x0070: 54 45 53 54 54 45 53 54  54 45 53 54 54 45 53 54  TESTTESTTESTTEST

Decrypted ciphertext, as plaintext: (below)
TESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTT
ESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTESTTEST
```

Your code's output should **exactly match** the sample output given for the input file.

6 Provided Code

We are providing you with a file that contains some pre-existing code to both aid in grading, and also to help you structure your assignment.

You must use the scaffold code for the assignment, else you will have differences in your output and will face severe mark deductions.

The scaffold begins with an authorship declaration. **You must fill this in with your full name, student number and date**.

6.1 Type definitions

The scaffold provides a set of type definitions: `book_t`, `byte_t`, `block_t` and `msg_t`. These are aliases for their defined types. For example, `block_t`, representing a block of 16 bytes of data is defined with:

```
#define BLOCKSIZE 16                  // The length of a block
typedef unsigned char byte_t;         // A byte (8 bits)
typedef byte_t block_t[BLOCKSIZE];    // A cipher bitset (block) (16 bytes)
```

`block_t` can be used anywhere where you would use a `unsigned char[16]`. For the rest of the implementations, see the scaffold code.

6.2 The `aes.h` library

The scaffold includes the library `aes.h` with the line `#include "aes.h"`. It provides the simple AES implementation functions:

`AES_encrypt(unsigned char message[16], unsigned char key[16], unsigned char output[16])`

`AES_decrypt(unsigned char ciphertext[16], unsigned char key[16], unsigned char output[16])`

Clearly, a variable of type `block_t` could be used in the arguments of the AES functions.

Do not modify any code in `aes.h` or `aes.c`. It will be replaced with a fresh version in grading.

6.3 The `a1grader.h` library

The scaffold includes the library `a1grader.h` with the line `#include "a1grader.h"`.

It provides access to the following functions:

```
void submit_stage0(int cipher_length, msg_t ciphertext, block_t outputs[N_OUTPUT_BLOCKS],
                block_t timesteps[N_TIMESTEPS], book_t cipherbook);
void submit_stage1(book_t stripped_book, int book_len);
void submit_stage2(block_t key2);
void submit_stage3(byte_t *key1);
void submit_stage4(msg_t plaintext);
```

These are called automatically in the scaffold code at the end of each stage of the assignment, for printing the outputs you saw in 5, as well as submitting those stages for assessment during grading. **Make sure your values for submission are stored in the variables that are passed to these functions**.

The library also provides the helper function `void hexdump(byte_t a[], int len)`, a debug function that outputs a byte array in the format you saw in each of the stages in section 5. You are free to use this function in debugging your code, but we recommend removing those debug printouts when you submit your final work.

We have provided you an implementation for this library, in `a1grader.c` for the **development version** of these functions. For grading, we will use the **grading version** of `a1grader.c`, which you will not be able to see, specifically for grading your work.

Do not modify any code in `a1grader.h` or `a1grader.c`. It will be replaced with a fresh, but modified version for grading.

The scaffold code includes the functions from these files by referencing their header files:

```
// Provides the functions AES_encrypt and AES_decrypt (see the assignment spec)
#include "aes.h"
// Provides functions to submit your work for each stage.
#include "a1grader.h"
```

If you wish to see how any of these functions are implemented you may open and view `aes.c` and `a1grader.c`.

6.4 The `read_hex_line` helper function

We provide the `read_hex_line` function to help you read the input data. It reads a line of input from the command line, and converts pairs of digits that represent numbers in base-16 (from 0 to 255, or equivalently `0x00` to `0xFF`), into our custom `byte_t` type[7].

```
/* Read in two chars at a time and converts to a number, stopping after max_count or at
↪  a newline
 * Returns the number of *bytes* read, and the last character in last_char
 */
int read_hex_line(unsigned char output[], int max_count, char *last_char)
```

7 Developing and testing your program

If you are developing on Grok, all of the files including the scaffold (where you will complete the assignment code), an implementation for `aes.c` and the sample grader `a1grader.c`, and their header files (`aes.h`, `aes.h`) will be automatically included.

You may compile and run as normal there.

If you plan to develop on your own machine, you will need to make sure you have **all** of these files downloaded, with their correct names, including the sample test files named `assignment1-input1.txt` and `assignment1-input2.txt`. They should all be within the same directory. You can then compile your program with the following command:

```
clang -Wall -std=c11 -pedantic program1.c aes.c a1grader.c -o program1
```

If you are using the gcc compiler, replace `clang` with `gcc` in the above line.[8]

Your code will be compiled with the same command as above for grading, so make sure it compiles without warnings and behaves as expected! Programs that do not compile will receive serious mark deductions!

You must not remove or modify any of the provided code, as doing so will confuse our grading tools.

Note: The test files we're using on Grok do not contain the same secret message contained in the provided input files. You can get full points either way, but it's more fun to actually decrypt the real messages. We do this because Grok shows you the desired output, and we want to hide the actual secrets!

[7]This is just a typedef for an `unsigned char`, which stores values from 0 to 255

[8]`clang` and `gcc` are largely interoperable—if something weird happens, please post a question on Piazza.

Style

Style will be graded as part of each section. While there is no official style guide, we require that your code be readable, appropriately documented, and use abstraction where appropriate (functions and typedefs).

We also provide the following set of indicators that we will use when evaluating your code for style, along with explanations for **why** we care about some of them.

7.0.1 Readability

- bad choice for function names
- bad choices for variable name
- use of typedefs without the suffix '`_t`'
- excessive commenting
- insufficient commenting (generally at least one comment per function to explain what the function does)
- inconsistent bracket placement
- inconsistent indentation
- lack of whitespace (blank line between functions or code blocks)
- multiple lines >100 chars

7.0.2 Adherence to common conventions

- `#defines` not in upper case (to make them stand out)
- use of typedefs without the suffix '_t'
- multiple lines >100 chars (used to prevent line-wrapping when reading the code in a terminal, historically was 80 characters)
- no authorship statement (for our class, student name and id at the top of program)
- use of external code without attribution

7.0.3 Efficiency and Maintainability

We are also concerned with the substantive design of your code. You should effectively structure your code so that it is reasonably memory/time efficient.

- excessive use of magic numbers (makes them hard to change later)
- absence of function prototypes (forces reliance on ordering of code, and makes it hard to know what's in a file)
- excessive use of global variables (leads to bugs)
- insufficient use of functions (makes code hard to maintain and understand)
- functions too long or too complex
- overly complex algorithmic approach
- unnecessary duplication/copying of data
- avoidance of typedefs
- overly complex function argument lists (this does not mean having lots of arguments alone, this means designing them so that more than necessary are needed)

When given the choice between following these guidelines and making your program worse—make your program better. Put in a comment for us so we know it was an active decision, which will help avoid misunderstandings when grading.

If you are looking for a style guide to follow we recommend the following resources:

- The CS50 style guide is appropriate for this course generally
- Google's C style is fairly popular
- Large open source projects such as Linux and GNU also have sophisticated style guides

We do not set a fixed style as C has no agreed on style—part of learning to code in C is developing an instinct for good formatting and documentation.

8 Submission Instructions

This assignment is worth 15% of your final mark for the subject.

You need to submit your code in Grok Learning (`https://groklearning.com`) for assessment.

Submission will NOT be done via Canvas. Instead, you will need to:

1. Log in to Grok Learning using your student login details.
2. Navigate to the Assignment 1 module of our subject COMP10002 2021 S1: Foundations of Algorithms.
3. Place your code in the `program1.c` tab window.
4. Compile your code by clicking on the `Compile` button.
5. Once the compilation is successful, click on the `Mark` button to submit your code. (You can submit as many times as you want to. *Only the last submission made before the deadline will be marked.*)
6. Two sample tests will be run automatically after you make a submission. Make sure that your submission passes these sample tests.
7. Two hidden tests will be run for marking purpose. Results of these tests will not be available until after the submissions are marked.

You can (and should) submit both **early and often** – to check that your program compiles correctly on our test system, which may have some different characteristics to the lab machines and your own machines.

To receive any points for the coded stages your code must compile.

We will test your code with inputs (and novels) other than that provided as a sample. Please ensure you don't hard-code any values from the input file.

You will be given sample input files:

- `assignment1-input.txt`
- `assignment2-input.txt`
- `grok-input1.txt`
- `grok-input2.txt`

and sample output for the two Grok inputs: `grok-output1.txt` and `grok-output2.txt`. You can test your code on your own machine with any one of the files using following command and compare the output with the matching file:

```
./program1 < assignment1-input.txt    /* < feeds data from the file into the program */
```

For the input files without matching output, you will know you have solved the task when the result of Stage 4 is a message written in English. The is no grade penalty for not testing on the two files without output, but doing so will allow you to read the secret messages within!

Note that we are using the following command to compile your code on the submission system (we name the source code file `program.c`).

```
clang -Wall -std=c11 -pedantic program1.c aes.c a1grader.c -o program1
```

The flag "`-std=c11`" enables the compiler to use a more modern standard of the `C` language—`C11`. To ensure that your submission works properly on the submission system, you should use this command to compile your code on your local machine as well.

9 Academic Honesty

This assignment is covered by the University's academic honesty policy, and your grade is predicated on meeting all the expectations under this policy.

You may discuss your work with others, but what gets typed into your program must be individual work, **not** from anyone else. Do **not** give (hard or soft) copy of your work to anyone else; do **not** "lend" your memory stick to others; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "no" when they ask for a copy of, or to see, your program, pointing out that your "no", and their acceptance of that decision, is the only thing that will preserve your friendship.

ALL code you submit must be your own work, the following exceptions notwithstanding:

1. If you wish to use code from an external source such as Stack Overflow, you must post privately on Piazza to validate the acceptability of the code, and must also cite it appropriately in your code. This includes code in the public domain for which you would ordinarily not be required to include a citation.
2. You **MAY** use any code provided as part of the official course materials (book, slides, lecture, etc.)

A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. As usual, if you're smart enough to outsmart our software, you're smart enough to do the assignment and actually learn something. See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **4PM, Tuesday 4th May, 2021** will lose penalty marks at the rate of 2 marks per day or part day late. Late submissions after 4PM, Friday 7th May, 2021 will **not** be accepted. Students seeking extensions for medical or other "outside my control" reasons should email Dr. Cohney. If you attend a GP or other health care professional as a result of illness, be sure to take a Health Professional Report form with you (get it from the Special Consideration section of the Student Portal), you will need this form to be filled out if your illness develops into something that later requires a Special Consideration application to be lodged. You should scan the HPR form and send it in connection with any non-Special Consideration assignment extension requests.

This Ends the Required Components of the Assignment. Read Ahead for More Background, and Bonus Stage.

Where can I learn more about the DUHK attack?

These resources are targeted above a first-year level, but might still be of interest, particularly the articles from mainstream media.

- Check out the official website for the research at duhkattack.com
- Watch the talk given by baby Dr. Cohney (before he was a Dr.)!
- The work was written about in Security Intelligence and Bleeping Computer among other places.
- Spot the reference on Wikipedia.

Bonus Stage: Implementing the Real DUHK Attack! Bonus 1.5 points

This stage is entirely optional. It is left as a **significantly difficult** exercise to challenge students looking for something much harder, that takes them outside of the course material. Please ensure you have successfully completed, tested and perfected all other stage of your assignment before attempting this stage. We do not necessarily anticipate **any** successful completions.

In the real DUHK attack, the key is known, but the timestamps are not. Further, you can only validate the correctness of your guesses by comparing against some cryptographic value generated from taking a function over the output from the pseudorandom number generator.

For this bonus stage, you may not expect any tutor support (though you can ask questions, you are not guaranteed a reply), and similarly should not expect partial credit.

- Submit a separate file `bonus.c` that that reads in an input file in the below format and outputs O_{11}
- Submit a valid input file `bonus-input.dat`! If time permits, we will release our own sample input.

Input Line 1: A 128-bit key k used for AES, equivalent to 16 unsigned chars, where each char is represented as a two-digit hexadecimal number in the file (two actual characters).

Input Line 2: O_9 and O_{10} from the random number generator, in groups of 16 unsigned chars, where each char is represented as a two-digit hexadecimal number in the file (two actual characters).

Input Line 3: T_7 and T_8 used to generate O_7 to O_8, in groups of 16 unsigned chars, where each character is represented as a two-digit hexadecimal number in the file (two actual characters).

Input Line 4: An integer representing the length in characters of the input on the next line

Input Line 5: A value y under the definition $y = g^x \mod p$, where p is the prime number 23, g is[9] 5, and x is O_{11} treated as a large integer.

Note that the value on Line 5 is one side of the set up for Diffie-Hellman key exchange, as described in Lecture 6 (approximately minute 32).

Timestamps should be precise to within 1 microsecond. Each timestamp value t_i is generated by a call to the library function `do_gettimeofday()`, which returns a 64-bit representation of the time. As the timestamps used by the random number generator as 128-bits, the full value T_i is computed as the concatenation of t_i with itself (64-bits of t_i, and then the 64-bits of t_i again).

You may assume that the distance between timestamps T_i and T_{i+1} is an average of 150 microseconds, with a standard deviation of 30 microseconds.

Note: To solve the bonus stage, you will need to use a library that can handle very large numbers, such as the GNU MP big number library.

[9] Chosen to be a *primitive root modulo p*