COS 226 Programming Assignment

Password Cracking

Write a program that uses a symbol table to break a widely-used password encoding scheme.

When a system's login manager is presented with a password, it needs to check whether that password corresponds to the user's name in its internal tables. The naive method would be to store passwords in a symbol table with the users' names as keys, but that method is vulnerable to someone getting unauthorized access to the system's table which would expose all of the users' passwords. Instead, most systems use a more secure method where the system keeps a symbol table that stores an encrypted password for each user. When a user types a password, that password is encrypted and checked against the stored value. If it matches, the user is allowed into the system.

For it to be effective, this scheme requires an encryption method with two properties: encrypting a password should be easy (since it has to be done each time a user logs on) and recovering the original password from the encrypted version should be hard.

Subset-sum encryption. A simple method for managing passwords works as follows: The length of all passwords is set to a specific number of bits, say N. The system maintains a table T of N integers that are each N bits long. To encrypt a password, the system uses the password to select a subset of the numbers in T and add them together: the sum (modulo 2^N) is the encrypted password.

The following tiny example for 5-bit keys illustrates the process. Suppose that the table has the following five 5-bit numbers:

```
0. 10110      0
1. 01101      0
2. 00101      1
3. 10001      0
4. 01011      1
```

For this table, the password `00101` would be encrypted as `10000`, since it says to use the sum of rows two and four in the table, and `00101 + 01011 = 10000`. Of course, in practice, the table would be much bigger, as discussed below.

Now, suppose you get access to the system table T and you also capture the user names and the corresponding encrypted passwords. This information doesn't get you into the system: to crack a password, you need to know a subset of T that sums to a given encrypted password. The security of the system depends on the difficulty of this problem (which is known as the *subset sum* problem). Obviously, the password length has to be set long enough to stop you from trying all possibilities, but short enough so that users can remember and type the passwords. In this assignment, you will see that passwords need to be longer than you might think. With N bits there are 2^N different subsets, so it would seem that 40 or 50 bits should be enough, but that is not the case.

Details. Rather than using numbers, it is typical to use some convenient translation from what users type to numbers. For this assignment, we use an alphabet of 32 characters (lowercase letters plus first six digits) in passwords, and encode them as arrays of 5-bit integers (`chars`) with 0 encoding `'a'`, 1 encoding `'b'`, and so forth. Thus, N = 5*C where C is the number of characters in the password.

To get started, you can use the code `encrypt.c`, which is the code that the system administrator would use to encrypt a user's password. It reads in the table T, then uses the bits of the password to select the words from the table to add together, and prints out the encrypted version. You can also use the tables `easy5.txt`, `easy8.txt`, `rand5.txt`, and `rand8.txt`. The first and third are for 5-`char` (25-bit) keys; the second and fourth are for 8-`char` (40-bit) keys. The first two are highly structured and are useful for debugging; the other two are randomly generated.

For example, with the constant `C` defined to `8` in the code, you should get the following result for the password `password`. The program prints out a proof that the encryption works, for your use in understanding the process:

```
% gcc encrypt.c -o encrypt
% encrypt password < rand8.txt
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000011

 1 gobxmqkt   6 14  1 23 12 16 10 19   00110011100000110111011001000001010100011
 2 qdrvjxwz  16  3 17 21  9 23 22 25   10000000111000110101010011011110110111001
 3 joobqxtz   9 14 14  1 16 23 19 25   01001011100111000001100001011110011111001
 4 xnoixmnk  23 13 14  8 23 12 13 10   10111011010111001000101110110001101010010
10 tcixtvem  19  2  8 23 19 21  4 12   10011000100100010111100111010100100011100
13 lqtsdtca  11 16 19 18  3 19  2  0   01011100001001110010000111001100010000000
15 zlptzlfp  25 11 15 19 25 11  5 15   11001010110111110011110010101100101011111
18 gmjuvyqw   6 12  9 20 21 24 16 22   00110011000100110100101011100010000010110
20 uoqrdhwp  20 14 16 17  3  7 22 15   10100011101000010001000110011111011001111
22 ltdkzndz  11 19  3 10 25 13  3 25   01011100110001101010110010110100011111001
23 btezrznq   1 19  4 25 17 25 13 16   00001100110010011001100011100101101100000
26 bujilqno   1 20  9  8 11 16 13 14   00001101000100101000010111000001101011110
27 qgaicljl  16  6  0  8  2 11  9 11   10000001100000001000000100101101001010011
28 yyefwcld  24 24  4  5 22  2 11  3   11000110000010000101101100001001011000011
30 gnvowyjk   6 13 21 14 22 24  9 10   00110011011010101110101101100001001010010
34 aynzobxh   0 24 13 25 14  1 23  7   00000110000110111001011100000110111001111
38 lxwewfhh  11 23 22  4 22  5  7  7   01011101111011000100101110001010011101111
39 aenipbjd   0  4 13  8 15  1  9  3   00000001000110101000011110000101001000011
   --------  ----------------------   -----------------------------------------
   vbskbezp  21  1 18 10  1  4 25 15   10101000011001001010000010010011001011111
```

If you wanted, you could use `bc` or another calculator to check that the columns sum to the digits corresponding to the letters in the decrypted password. The table `easy8.txt` makes this check easier, for debugging.

```
% encrypt password < easy8.txt
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000011

 1 aaaaaaac   0  0  0  0  0  0  0  2   00000000000000000000000000000000000000010
 2 aaaaaaae   0  0  0  0  0  0  0  4   00000000000000000000000000000000000000100
 3 aaaaaaai   0  0  0  0  0  0  0  8   00000000000000000000000000000000000001000
 4 aaaaaaaz   0  0  0  0  0  0  0 25   00000000000000000000000000000000000011001
10 aaaaabaa   0  0  0  0  0  1  0  0   00000000000000000000000000000010000000000
13 aaaaaiaa   0  0  0  0  0  8  0  0   00000000000000000000000000010000000000000
15 aaaabaaa   0  0  0  0  1  0  0  0   00000000000000000000000010000000000000000
18 aaaaiaaa   0  0  0  0  8  0  0  0   00000000000000000000010000000000000000000
20 aaabaaaa   0  0  0  1  0  0  0  0   00000000000000000010000000000000000000000
22 aaaeaaaa   0  0  0  4  0  0  0  0   00000000000000000100000000000000000000000
23 aaaiaaaa   0  0  0  8  0  0  0  0   00000000000000001000000000000000000000000
26 aacaaaaa   0  0  2  0  0  0  0  0   00000000000000100000000000000000000000000
27 aaeaaaaa   0  0  4  0  0  0  0  0   00000000000001000000000000000000000000000
28 aaiaaaaa   0  0  8  0  0  0  0  0   00000000000010000000000000000000000000000
30 abaaaaaa   0  1  0  0  0  0  0  0   00000000010000000000000000000000000000000
34 azaaaaaa   0 25  0  0  0  0  0  0   00000110010000000000000000000000000000000
38 iaaaaaaa   8  0  0  0  0  0  0  0   01000000000000000000000000000000000000000
39 zaaaaaaa  25  0  0  0  0  0  0  0   11001000000000000000000000000000000000000
   --------  ----------------------   -----------------------------------------
   b0onjjbh   1 26 14 13  9  9  1  7   00001110100111001101010010100000100111
```

Be sure to familiarize yourself with `encrypt.c` before proceeding. Note that it uses `key.h` and `key.c` to perform base-32 addition over C digit integers.

Brute force solution. Your first task is to write a brute force decrypt program `brute.c` that does the opposite of `encrypt.c`. Given an encrypted password and the table, it should find the password: the `C` characters that, when converted to an N-bit binary number, specifies the subset that sums to the N-bit binary number that converts to the `C` characters given as command-line argument. You should be able to easily break 5-char passwords, since there are only 32*32*32*32*32 (about 33 million) different possible subsets.

With the constant C defined to be 5, you should get the following result for the password `passw`:

```
% gcc encrypt.c key.c -o encrypt
% encrypt passw < rand5.txt
exvx5
```

```
% gcc brute.c key.c -o brute
% brute exvx5 < rand5.txt
i0ocs
passw
```

Note that there many not be a unique solution so your program should print out all of them. Note also that this approach won't work for longer passwords. For example, for 10-char passwords, there are 32^10 (over

1000 trillion) different possible subsets.

Symbol-table solution. Your next task is to write a faster decryption program (name your program `decrypt.c`) that is functionally equivalent to `brute.c` but fast enough to decrypt 10-char passwords in a reasonable amount of time (say, under an hour). To do so, use a symbol table. The basic idea is to take a subset S of the table T, compute all possible subset sums that can be made with S, put those values into a symbol table, then use that symbol table to check all those possibilities with just one lookup.

When you consider the scheme just sketched, several questions immediately come to mind: How big should S be? Precisely how is the one-lookup check going to work? Which symbol-table ADT is appropriate? Which algorithm would be best? Coping with these details is the substance of your work for this assignment.

You should debug your program for 6-`char`, then move up to 8 and 10-`char` passwords. Your goal, of course, is to be able to decrypt any password that was encrypted with `encrypt.c`, as in, for example:

```
% gcc decrypt.c -o decrypt
% decrypt vbskbezp < rand8.txt
koaofbmx
password
xvbyofnz
1p1ngsgg
```

This approach also breaks down as the password length increases, but it does sufficiently cut the work to crack passwords to render vulnerable systems that use this scheme.

Deliverables. Submit the two programs described above and a `readme.txt` that documents the approach that you used. When feasible, use your `decrypt.c` implementation to decrypt the following passwords that we encrypted with `encrypt.c` (using `rand8.txt`, `rand10.txt`, and `rand12.txt`).

```
xwtyjjin          h554tkdzti          uz1nuyric5u3
rpb4dnye          oykcetketn          xnsriqenxw5p
kdidqv4i          bkzlquxfnt          4l4dxa3sqwjx
m5wrkdge          wixxliygk1          wuupk1ol3lbq
```

Include in your `readme.txt` estimates of the amount of time that it would take your `brute.c` program to break an 8-character password and the amount of time *and* space that it would take your `decrypt.c` program to break a 12-character password.

Extra credit. It is annoying that several passwords may map to the same encrypted password. Create tables for which there is no such ambiguity when decrypting.