5

Lab 3: Password Cracking

In this assignment, we will explore the space-time tradeoffs of some data structures used to crack passwords. Because stolen password databases are a real problem, most reasonably secure password database implementations do not store passwords in plaintext form.

Your task is to explore several schemes used to recover passwords from password databases, a process often referred to as *password cracking*.

This assignment is split into two parts. In part 1, you will implement and generate a cracked password dictionary. In part 2, you will implement and generate several variations on precomputed hash tables, including rainbow tables. In both parts, you will attempt to recover plaintext passwords from an encrypted password database.

5.1 *Required Reading*

Please read "Why Stolen Password Databases are a Problem" and "Trading Time for Space. Both readings are available on the course website.

5.2 *Requirements*

Collaboration. This is an individual assignment. All of the code you submit must be written exclusively by you. You are welcome to collaborate with a classmate to understand the assignment, and to discuss how a solution should work at a high level, but you *must not share code*. Rule of thumb: if you are looking at code on someone else's screen, it's an honor code violation.

Language and Libraries. You solution must be written using C. Only use the built-in C libraries and the `libmd` library, which contains an implementation of the MD5 hash algorithm. Do not download any additional libraries. If you are at all uncertain about which libraries are OK and which are not, please ask me. You are welcome to use any code I give you as a starting point.

To install `libmd` on your Raspberry Pi, type:

```
$ sudo apt install libmd-dev
```

Common environment. If you wish, you may develop this code on your own machine, but please be sure to test it on your class Raspberry Pi before submitting. If you develop on a machine different from your Raspberry Pi, there *will* be differences, and some potential differences mean that your code may not build at all. All assignments will be graded using the Raspberry Pi.

If I can't build your code, I can't grade it. That will likely have a negative effect on your grade.

Stack Overflow. You are permitted to refer to Stack Overflow for help, but you *must not under any circumstances copy the code you see there*. If you find a helpful Stack Overflow post, you must attribute the source of your inspiration in a comment at the appropriate location of your code, and you must provide a URL for me to look at. Unattributed code will be considered an honor code violation.

Instructions for Compiling and Running. You must supply a file called `BUILDING.md` with your submission explaining how to:

1. compile your program,[1] and
2. how to run your program on the command line.

[1] Hint: I should be able to just type `make`.

Reflection questions. This assignment asks you to answer a few questions. You must supply the answers to these questions in a file called `PROBLEMS.md`.

Starter code.
The starter code contains the following files:

File	Purpose
`epassword.db`	An encrypted password database.
`database.c`	Library for reading the `epassword.db` database.
`database.h`	API for `database.c`.

You will also be given a zipped, encrypted database, `encrypted_db.txt`.

5.3 *Inputs and Outputs*

The file, `encrypted_db.txt`, is a password database of the following form:

$\texttt{username}_1\texttt{,pwhash}_1$
$\texttt{username}_2\texttt{,pwhash}_2$
...
$\texttt{username}_n\texttt{,pwhash}_n$

where $\texttt{username}_i$ is an alphanumeric user name, and where $\texttt{pwhash}_i$ is a 32 digit hexadecimal number (i.e., 16 bytes), representing a password hash.

Since we are exploring the scenario where you possess a stolen password database, you will have direct access to the database file.

Passwords are hashed using the MD5 cryptographic hash algorithm[2]. Password plaintexts, which are not stored in the password database, are composed of the following characters: `0`–`9` and `A`–`F`, and are *exactly 4 characters long*. This file is the input to your program.

[2] https://en.wikipedia.org/wiki/MD5

In both parts 1 and 2, you will be decrypting this database. Your goal in both cases is to produce as output a "cracked" password database of the form:

$\texttt{username}_1\texttt{,password}_1$
$\texttt{username}_2\texttt{,password}_2$
...
$\texttt{username}_n\texttt{,password}_n$

Your decrypted database *must be sorted by username*.

To ensure that your code is working correctly, here are some sample plaintext passwords for a few users in the dictionary:

```
dbarowy,BA1D
ihowley,F00D
wjannen,CAFE
```

5.4 *Part 1: Dictionary Attack*

In this part, your job is to crack the `encrypted_db.txt` database using a dictionary attack. You should be able to call your program from the

command line like so:

```
$ ./dictattack <encrypted database> <decrypted database>
```

where `<encrypted database>` is the path to your encrypted database, `epassword.db`, and `<decrypted database>` is the path where you want the decrypted database to be written.

Your code should have a `dictattack.c` file containing a `main` method. You should also create a library called `crackutil.c` that comes with a `crackutil.h` file.

I describe the pieces you must implement below in order to assemble your dictionary attack. I leave unspecified how these pieces fit together, but if you understand dictionary attacks, the correct way to connect the pieces will be obvious.

5.4.1 *Plaintext generator*

A dictionary attack needs a way of generating all possible plaintexts. Create two files called `crackutil.h` and `crackutil.c`. In `crackutil.h`, insert the following function signature, and in `crackutil.c`, implement it:

```
void genPlaintext(char *dst, int n);
```

where `dst` is a pointer to a string buffer and `n` is a number between `0` and `65535`. `genPlaintext` should write a 4-character plaintext into `dst` using the set of characters described above (see "Inputs and Outputs"). You may implement this function any way you want, but you need to be sure that the function is capable of generating all possible plaintexts using our scheme. One such scheme might produce a mapping from inputs to outputs like so:

```
0       0000
1       0001
2       0002
...
331     014B
...
65535   FFFF
```

Note that the set of valid password plaintext characters just happens to be the same set of characters used when printing a number in hexadecimal format.[3] In fact, if you look carefully at the sample mapping above, an algorithm that reproduces it should suggest itself.

Always document your functions so that others can understand them without reading your implementation. Here is a suggested comment for this function:

```
/**
 * Generates the nth plaintext.
 *
 * @param dst  A char buffer of length 5.
 * @param n    A number between 0 and 65535.
 */
```

You may find the `snprintf` function helpful for this step. See `$ man 3 snprintf` for details.

[3] https://en.wikipedia.org/wiki/Hexadecimal

5.4.2 *Cipher function*

A dictionary attack must be able to run the same cryptographic hash function that a password scheme uses to hash plaintexts. Since cryptographic functions are not usually secret, we will assume that we know

what function our targeted password system uses. For this assignment, will assume that the MD5 cryptographic hash function is being used.

The MD5 algorithm is in the `libmd-dev` package. [4] This library is slightly cumbersome to use, so instead of using it directly, I provide a straightforward wrapper function.

[4] See section 5.2 for installation instructions.

First, be sure to include the appropriate `libmd` header:

```
#include <md5.h>
```

A widely-held principle in computer security is that mechanisms should be fundamentally secure. In other words, knowing *how* they work should not prevent them from being effective protections. Consequently, all widely-deployed cryptographic algorithms are developed in full public view. Relying on secrecy as a security mechanism is often derisively called "security through obscurity," and it should be avoided because once an attacker learns your secret, your defenses evaporate.

Then, put the following function in your `crackutil.c`. Don't forget to update `crackutil.h` with the appropriate function signature.

```
/**
 * Hashes password using MD5.  Assumes that password
 * is exactly PTLEN-1 chars and that hash is a pointer
 * to an array of length MD5_DIGEST_LENGTH.
 *
 * @param password A string to hash.
 * @param dst      A pointer to an array to store the hash.
 */
void  hash(char* password, uint8_t* dst) {
  MD5_CTX ctx;
  MD5Init(&ctx);
  MD5Update(&ctx, (uint8_t*)password, PTLEN-1);
  MD5Final(dst, &ctx);
}
```

A `uint8_t` is an unsigned, 8-bit integer. Remember that one byte is represented by 8 bits in most modern computer hardware, so a `uint8_t` is also a byte.

The `password` argument to the `hash` function is a pointer to a plaintext password string, and the `dst` argument is a pointer to an `uint8_t` array long enough to hold a 16-byte MD5 hash. The `md5.h` header defines the constant `MD5_DIGEST_LENGTH`, which is the correct length of the `uint8_t` array to use for `dst`.

Note that I leave it up to you to define `PTLEN` which represents the length of the plaintext buffer, `password`. How long do you think `PTLEN` should be? Put a preprocessor `#define` in your `crackutil.h` to define this constant, like so:

```
#define PTLEN <some number>
```

If you type `$ man 3 md5`, you will see that the MD5 documentation refers to "message digests". A *digest* is another name for a ciphertext produced by a hash algorithm.

Finally, `libmd` is a shared library, which means that you need to provide `gcc` with a linker flag. The linker flag for `libmd` is `-lmd`. Remember, *linking*, which is the step your compiler takes when it joins library files together with your program source code, happens when you are producing the final program binary. The program binary is the one that contains your `main` method.

Bonus: if you read the `man` pages for the MD5 functions used in the `hash` function, you will discover that while my implementation is correct, it is somewhat inefficient. If you want to *optionally* push your knowledge further, try using them as the documentation suggests *instead* of using my wrapper function.

5.4.3 *Pretty printing*

You will likely want to print your hash values out at various points in the development of `dictattack`. One reason to do this is to verify that your MD5 hashes are correct. For example, the plaintext `000F` should hash to an MD5 that prints out as `45632A2B09337E7FC4415AAF9E098491`.

Conventionally, we print an MD5 value as a 32-digit hexadecimal number. This length makes sense because an MD5 hash is an array of 16 `uint8_t` values. Since one byte can be represented by two hexadecimal digits, $16 \times 2 = 32$, meaning that we expect a 32-digit hexadecimal string as output.

Write an implementation for the following function signature, and add it to `crackutil.c`. Be sure to update your `crackutil.h` header.

```
void  hashToString(uint8_t* hashbuf, char* dst);
```

The argument `hashbuf` is a pointer to your MD5 in array form, and `dst` is the destination buffer for your pretty-printed MD5 string. You are strongly encouraged to add a `#define` representing the correct MD5 string length to your `crackutil.h`.

5.4.4 *Database reader*

A dictionary attack must be able to read in a stolen password database. Write a method with the following signature and add it to `crackutil.c` and `crackutil.h`.

```
list_t* readPasswords(char* path) {
```

The argument `path` is a path string, like `"epassword.db"`. The return value is a linked list of password data, where a list node is defined in `database.h` as a `list_t`:

```
typedef struct node {
  pwent_t data;
  struct node * next;
} list_t;
```

And, for completeness, where the list node's data item is a kind of `struct`, called `pwent_t`, defined as:

```
typedef struct pwent {
  char username[ULEN];
  char password[PWLEN];
} pwent_t;
```

You are encouraged to use the `read_pwdb` function from `database.h`.

Since `readPasswords` returns a linked list, I can access individual entries either by searching for them using `list_find` from `database.h`, or by traversing the list as follows:

```
list_t* db = readPasswords("some_database.db");
list_t* finger = db;
```

```
while(finger->next != NULL) {
        printf("username: %s\n", finger->data.username);
        printf("passhash: %s\n", finger->data.password);
        finger = finger->next;
}
```

Finally, if you use `read_pwdb`, be aware that it allocates memory using `malloc`, which means that somewhere in your program, you will need to `free` the data structure it returns. Think carefully about how to `free` it.

5.4.5 *Hash table*

The C standard library on most UNIX machines come equipped with an implementation of a hash table called `hsearch`. You can learn about this hash table by typing `$ man 3 hsearch`, which includes sample code. You will use the `hsearch` database to create a dictionary for this lab.

There are some important caveats about the `hsearch` implementation that you should be aware of.

- You can have at most *one* hash table at a time. Consequently, you never are given the ability to save a pointer to this data structure.
- You create a new database by using the `hcreate` function. `hcreate` takes a parameter for the maximum size of the table. For performance reasons, you should set it to be 25% larger than the maximum number of elements than you expect to store in the table.
- Both storing and retrieving from the table use the same function, `hsearch`. The behavior of this function depends on the `action` argument, either `ENTER` or `FIND`.
- Elements cannot be deleted from the table.
- When you store in the hash table, what is stored is a *copy* of a pointer to a `key` and a *copy* of a pointer to a `data` item.
- The type of the `key` is always a string pointer, namely a `char *`.
- The type of the `data` is always a `void *`, which essentially means "a pointer to something." If you stored a string pointer in the `data` field, you will need to cast it, e.g., `(char *)`, when you read it out.
- Finally, `hdestroy` only deallocates the keys in the table, not the data. When deallocating, you will need to carefully consider how to deallocate both keys and values.

Finally, because of the last item above, I suggest that when storing data in your table, that you store copies of key and data values. For example,

```
ENTRY e, *ep;
e.key = strdup(key);
e.data = strdup(value);
ep = hsearch(e, ENTER);
```

Remember to verify that the `hsearch` function is successful. The `man` page explains how to check for success.

5.4.6 *Dictionary-based cracking algorithm*

Your `main` method should systematically call `genPlaintext` and, for each plaintext generated, call your `hash` function to generate a ciphertext. Every pair of plaintext and ciphertext should be stored in a hash table, otherwise known as a *dictionary*. Since our hash table implementation requires `char *` as keys, you will need to call your `hashToString` function to convert the has to a string. After generating this table, you will read entries from the `epassword.db` encrypted file, look up the hashed password in your dictionary, and decrypt it. Each user and their decrypted password should be printed out in the form:

$\texttt{username}_1\texttt{,password}_1$
$\texttt{username}_2\texttt{,password}_2$
...
$\texttt{username}_n\texttt{,password}_n$

Finally, your algorithm must ensure that the output, which you should call `password.db`, is in alphabetical order. Since `epassword.db` is already in the correct order, you just need to preserve this order.

5.5 *Part 2: Trading Time for Space*

Dictionary attacks are an effective tool when time and space are not an issue.[5] Unfortunately, distributing dictionaries can be cost-prohibitive even for password schemes with only modest complexity. Precomputed hash chains and rainbow tables address this problem, making cracked password databases smaller. They work by trading extra time to perform a lookup for reduced space used by the data structure.

[5] For example, you are a government-level attacker who can devote supercomputing resources to solving the problem.

In this part, you will write an implementation that hashes with a configurable "table type." Your implementation should be callable on the command line like so:

```
$ ./hashchain <encrypted database> <decrypted database> <type> <width> <height>
```

where

`<type>` is `exhaustive`, `random`, or `rainbow`; `<width>` is the width of the hash chain table (in other words, the length of the hash chain);

`<height>` is the number of hash chains generated; `<encrypted database>` is the path to your encrypted database; and `<decrypted database>` is the path where you want the decrypted database to be written.

Your code should have a `hashchain.c` file that contains a `main` method. You are encouraged to reuse your code you developed for your dictionary attack in this section, and I encourage you to add new helper functions to your `crackutil.c`.

5.5.1 *Reduction function*

An attack using precomputed hash chains requires a so-called *reducer*, a function that maps ciphertexts to plaintexts. Note that a reducer does not compute the hash inverse; in general, computing hash inverses is infeasible. Instead, the purpose of a reducer is to select a new plaintext (using a ciphertext) so that hashes can be "chained" together. Reducers are used as a kind of space-saving mechanism, allowing us to store only the *starting point* and *ending point* of a hash chain.

Add a function with the following signature to your `crackutil`:

```
void reduce(uint8_t* ciphertext, int index, char* buf);
```

where `ciphertext` is the ciphertext to reduce, `index` is a number that selects a reduction function, and `buf` is a pointer to a buffer to store a plaintext. When you call `reduce`, it should return a plaintext.

There are many ways to reduce a ciphertext, but the most important criterion is that the reducer must be able to produce any possible plaintext given its input domain (all possible ciphertexts). For example, one such implementation might produce the following mapping from ciphertext to plaintext:

`reduce(AA338257F792484CAEB90FC3D8A708AF, 0, ...)` → `AA33`
`reduce(57BE0A3E4E7DF1C975A5B1FCAAB8CF6B, 0, ...)` → `57BE`
`reduce(C90874550C415765F8B15B45E4F64A9E, 0, ...)` → `C908`

Changing the index parameter might produce the following:

`reduce(AA338257F792484CAEB90FC3D8A708AF, 1, ...)` → `A338`
`reduce(57BE0A3E4E7DF1C975A5B1FCAAB8CF6B, 1, ...)` → `7BE0`
`reduce(C90874550C415765F8B15B45E4F64A9E, 1, ...)` → `9087`

5.5.2 *Precomputed hash chain (PCHC) table*

In this part, you will generate a precomputed hash chain (PCHC) table. To generate a PCHC table, you will need to reuse your `genPlaintext` function from Part 1. Write a table-generating function that has the fol-

lowing signature:

```
int genTable(tabletype_t type,
             int width,
             int height,
             char** keys);
```

where `type` is the following C `enum`

```
typedef enum tabletype {
        EXHAUSTIVE,
        RANDOM,
        RAINBOW
} tabletype_t;
```

where `height` is the number of chains to be generated, where `width` is the number of reductions applied in a given chain, and where `keys` is a pointer to an array that stores the hash table's keys for later deallocation.

The function should return the number of chains inserted into the table.

Remember from the reading that a PCHC table maps plaintexts to plaintexts. Ciphertexts are not stored in the table at all!

As in part 1, I suggest using the `hsearch` hash table implementation, where the `key` is a plaintext starting point and the `data` is a plaintext ending point. Because `hsearch` cannot store duplicate keys, you will be limited to storing only one chain for a given starting point. Therefore, some chains will need to be discarded. I leave it to you to decide whether you should discard the new chain or the old chain. Either way, decryption rates will be lower than if you use a data structure that does not discard chains. For the purposes of assignments, discarding chains is fine, but if you want an extra challenge, try designing an alternative data structure.

You must be able to generate the following table types:

- `EXHAUSTIVE`. Generate a PCHC table of size `width` $\times$ `height` by systematically enumerating all possible plaintexts. If `width` $\times$ `height` $< |P|$, where P is the set of all possible plaintexts, then just enumerate the first `width` $\times$ `height` passwords.

- `RANDOM`. Generate a PCHC table of size `width` $\times$ `height` by randomly selecting plaintexts.

- `RAINBOW`. Generate a rainbow table of size `width` $\times$ `height` by randomly selecting plaintexts.

Note that the only difference between an ordinary precomputed hash chain table and a rainbow table is how reducers are applied. In an ordinary table, one applies a fixed reducer (e.g., `reduce(ciphertext, 0, ...)`) at every step in a chain. In a rainbow table, one applies a *different reducer for every reduction step in a chain*.

For example, the first reduction might be called with `reduce(ciphertext`$_0$`, 0, ...)`, the second reduction might be called with `reduce(ciphertext`$_1$`, 1, ...)`, the third reduction might be called with `reduce(ciphertext`$_2$`, 2, ...)`, and so on, up to `reduce(ciphertext`$_{w-1}$`, n, ...)`, where w is the width of the table.

5.5.3 *Convert from `char*` ciphertext to `uint8_t` array*

At some point during this lab, you will need to convert from the base-64 encoded hash strings stored in the encrypted password database to the `uint8_t` arrays that the MD5 values that your `hash` and `reduce` functions use. To do this, you will have to think back to CSCI 237 a bit. Here's the signature of the function you should write.

```
void hashFromString(char* ciphertext, uint8_t* dst)
```

where `ciphertext` is a hexadecimal string like `14456DED73AF945CE2B3AFF7260D4B34` and `dst` is an array of `uint8_t` values big enough to store the numeric representation of a hash.

This problem is not hard if you break it down into pieces. The most important piece of information is that each pair of hexadecimal digits encodes a single byte. Recall that a `uint8_t` is a byte.

For example, the hexadecimal string `B4` is the decimal number `180`. We handle each hexadecimal character—one *nibble*—at a time. The low-order hexadecimal nibble `4` is the decimal value `4`. The high-order hexadecimal nibble `B` is the decimal number 11×16. To find the combined value, add the two numbers together: $\texttt{B4} = 4 + 11 \times 16 = 180$.

If you've implemented this step correctly, you should be able to compute a "round trip" of a base-64 encoded hash string through your `hashFromString` and `hashToString` functions. The starting and ending strings should be the same. For example,

```
char *ciphertext = "14456DED73AF945CE2B3AFF7260D4B34";
uint8_t ct[CTNUMBYTES];
hashFromString(ciphertext, ct);
char ciphertext2[HASHHEXLEN];
hashToString(ct, ciphertext2);
printf("'%s' is '%s'\n", ciphertext, ciphertext2);
```

The above should print out:

```
'14456DED73AF945CE2B3AFF7260D4B34' is '14456DED73AF945CE2B3AFF7260D4B34'
```

5.5.4 *PCHC table lookups*

To lookup a decryption, you will need to supply the following lookup function:

```
void lookup(char* ciphertext,
            tabletype_t tt,
            int width,
            int height,
            char* buf);
```

where `ciphertext` is a ciphertext string, `tt` is the table type, `width` is the table width (or chain length), and `height` is the table height (or number of chains).

The function should return `true` if a decryption was found, otherwise it should return `false`. You can use `bool` values in C by including the following header:

```
#include <stdbool.h>
```

The algorithm for performing a PCHC lookup is discussed in the "Trading Time for Space" reading. Note that lookups for rainbow tables work differently than for vanilla PCHC tables, because searching a chain for a ciphertext involves not just hashing and reducing, but hashing and reducing using the same sequence of reductions used to originally construct the table. If this does not make sense to you, I strongly recommend simulating a rainbow table lookup on paper (perhaps with some help from the reference implementation) until you see why.

5.5.5 *Generating a cracked password database*

Finally, as in Part 1, your `main` method should read the password database (reusing your `readPasswords` function), generate a table of the requested type (using `genTable`), and then should attempt to decrypt all of the password hashes stored in the database (using the `lookup` function), writing out the ones it can decrypt to a file.

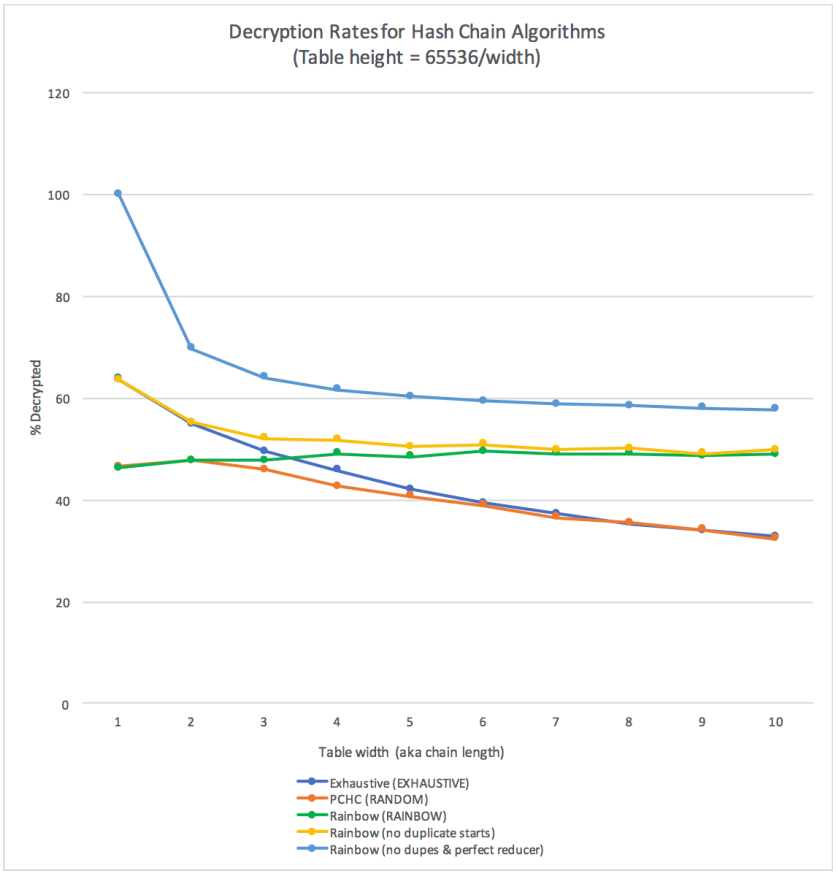
Your implementation should report the following two statistics:

1. the number of hash chain collisions (i.e., the number of hash chains with the same endpoint), and

2. the number of successful decryptions.

You should expect that your code will be tested against both the supplied database and a database of my choosing. Note that any technique based on precomputed hash chains is *unlikely to be 100% successful* at decrypting all of the hashes, because collisions are hard to avoid. Nonetheless, if your lookup fails close to 100% of the time, something is wrong with your code.

The following chart, generated using my own code, should give you a sense of the kinds of decryption rates you can expect with a correct implementation.



5.6 *Reflection Questions*

Provide answers to the following questions in a file called `PROBLEMS.md`.

1. The password scheme in Part 1 has 65,536 possible passwords. How many passwords would an *up-to-8 character* alphanumeric (uppercase and lowercase) scheme have, assuming that the empty password is disallowed? Explain your derivation.

2. Using your own implementation as a benchmark, how long do you estimate that it would take to generate a dictionary for the scheme in the previous question?

3. How many [mega/giga/tera/peta] bytes would it take to store a password dictionary for such a scheme assuming that password fields are always 8 bytes (where entries shorter than 8 bytes are NULL-padded) and where password hashes are 16 byte MD5 hashes? For simplicity, ignore the existence of hash collisions.

4. Given your answer to the previous question, what are the drawbacks when using your dictionary attack implementation for a password scheme like the one discussed in the previous questions? Think about the compute resources you actually used when performing your attack (CPU, RAM, disk; hint: where did you store data structures as you carried out your attack?). How might you modify your dictionary attack implementation to address the limitations you identify?

5. Why are we unable to decrypt all of the passwords in Part 2? Do you think a different reducer would help?

5.7 *Bonus*

Compute the success probability formula found at the top of page 6 in the paper "Making a Faster Cryptanalytic Time-Memory Trade-Off". What is the expected success probability for a table of width 16 and a height of 4096? Note that Oechslin states that the probability that any two plaintexts collide is $\frac{1}{m}$, where m is the number of possible plaintexts, which assumes, somewhat optimistically, that both the hash function and reducer select values perfectly uniformly at random. The number of successes you observe will probably be lower. Estimate your hash collision probability empirically by generating a table of width 1 and a height of m. How close is your implementation? Also note that Oechslin's notation is a little different than the notation we use in class.

5.8 *Development Tips*

This assignment may seem overwhelming; in actuality, like most software, it merely contains a large number of small steps. Work systematically, finishing off each step, and you will successfully complete the entire assignment.

- The password scheme we are attacking has 16^4 possible passwords, which is a big-ish number. But none of the techniques above actually depend on that number. Do yourself a favor and work on a smaller instance of the problem. For example, you might define a constant `PWLEN` that says how long a password is, and during development, `#define PWLEN 1`. This will make testing much faster, since you can manually check by hand whether your code is doing the right thing.

- Ciphertexts are a `uint8_t*`, which is a little bit of a pain, since you can't print them directly during debugging. Do yourself a favor and use the "pretty print" function for ciphertexts we came up with so that you can print them in debug output.
- You should be able to simulate dictionary, precomputed hash chain, and rainbow table lookups on paper. Be sure to work through each algorithm on paper *first*. If you are struggling with this part, I am happy to meet with you during office hours. Working with a friend on lookups is also an *excellent* use of a study group, particularly since I think that performing a lookup on paper is *a fair question to ask on a midterm exam*.[6]
- Finally, it's not a bad idea to implement `genTable` and `lookup` first only for the `EXHAUSTIVE` scheme, then the `RANDOM` scheme, then finally the `RAINBOW` scheme. Each scheme adds a little bit of complexity, so you can rule out problems by building your tool end-to-end for the simplest scheme (`EXHAUSTIVE`) first.

[6] HINT HINT HINT.

5.9 *Lab Deliverables*

By the start of lab, you should see a new private repository called `cs331lab02_pwcrack-USERNAME` in your GitHub account (where `USERNAME` is replaced by your username).
For this lab, please submit the following:

```
cs331lab02_pwcrack-{USERNAME}/
    BUILDING.md
    PROBLEMS.md
    README.md
    crackutil.c
    crackutil.h
    dictattack.c
    epassword.db
    hashchain.c
    Makefile
```

where the `.c`, `.h`, and `Makefile` files contain your *well-documented* source code. You may also add additional source files if you want.

It is always a good practice to create a small set of tests to facilitate development, and you are encouraged to do so here.

As in all labs, you will be graded on *design*, *documentation*, *style*, and *correctness*. Be sure to document your program with appropriate comments, including a general description at the top of the file, and a description of each function with pre- and post-conditions when appro-

priate. Also, use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.

Whenever you see yourself duplicating functionality, consider moving that code to a helper function. There are several opportunities in this lab to simplify your code by using helper functions.

5.10 *Submitting Your Lab*

As you complete portions of this lab, you should `commit` your changes and `push` them. **Commit early and often.** When the deadline arrives, we will retrieve the latest version of your code. If you are confident that you are done, please use the phrase `"Lab Submission"` as the commit message for your final commit. If you later decide that you have more edits to make, it is OK. We will look at the latest commit before the deadline.

Be sure to push your changes to GitHub. To verify your changes on GitHub, navigate in your web browser to your private repository on GitHub. It should be available at https://github.com/williams-cs/cs331lab02_pwcrack-{USERNAME}. You should see all changes reflected in the files that you `push`. If not, go back and make sure you have both committed and pushed.

Do not include identifying information in the code that you submit. We will know that the files are yours because they are in *your* `git` repository. We grade your lab programs anonymously to avoid bias. In your `README.md` file, please cite any sources of inspiration or collaboration (e.g., conversations with classmates). We take the honor code very seriously, and so should you. Please include the statement `"I am the sole author of the work in this repository."` in a comment at the top of your C files.

5.11 *Bonus: Feedback*

I am always looking to improve our labs. For one bonus percentage point, please submit answers to the following questions using the anonymous feedback form for this class:

1. How difficult was this assignment on a scale from 1 to 5 (1 = super

easy, ..., 5 = super hard)? Why?

2. Did this assignment help you to understand password attacks?
3. Is there is one skill/technique that you struggled to develop during this lab?
4. Your name, for the bonus point (if you want it).

5.12 *Bonus: Mistakes*

Did you find any mistakes in this writeup? If so, add a file called `MISTAKES.md` to your GitHub repository and provide a bulleted list of mistakes. Be sure to explain them in enough detail that I can verify them. For example, you might write

```
* Where it says "bypass the auxiliary sensor" you should have
    written "bypass the primary sensor".
* You spelled "college" wrong ("collej").
* A quadrilateral has four edges, not "too many to count" as you
    state.
```

For each mistake I am able to validate, I will award limited bonus credit, not to exceed 100% of your grade.