CSCI 476: Computer Security

Hashing (Part 2)

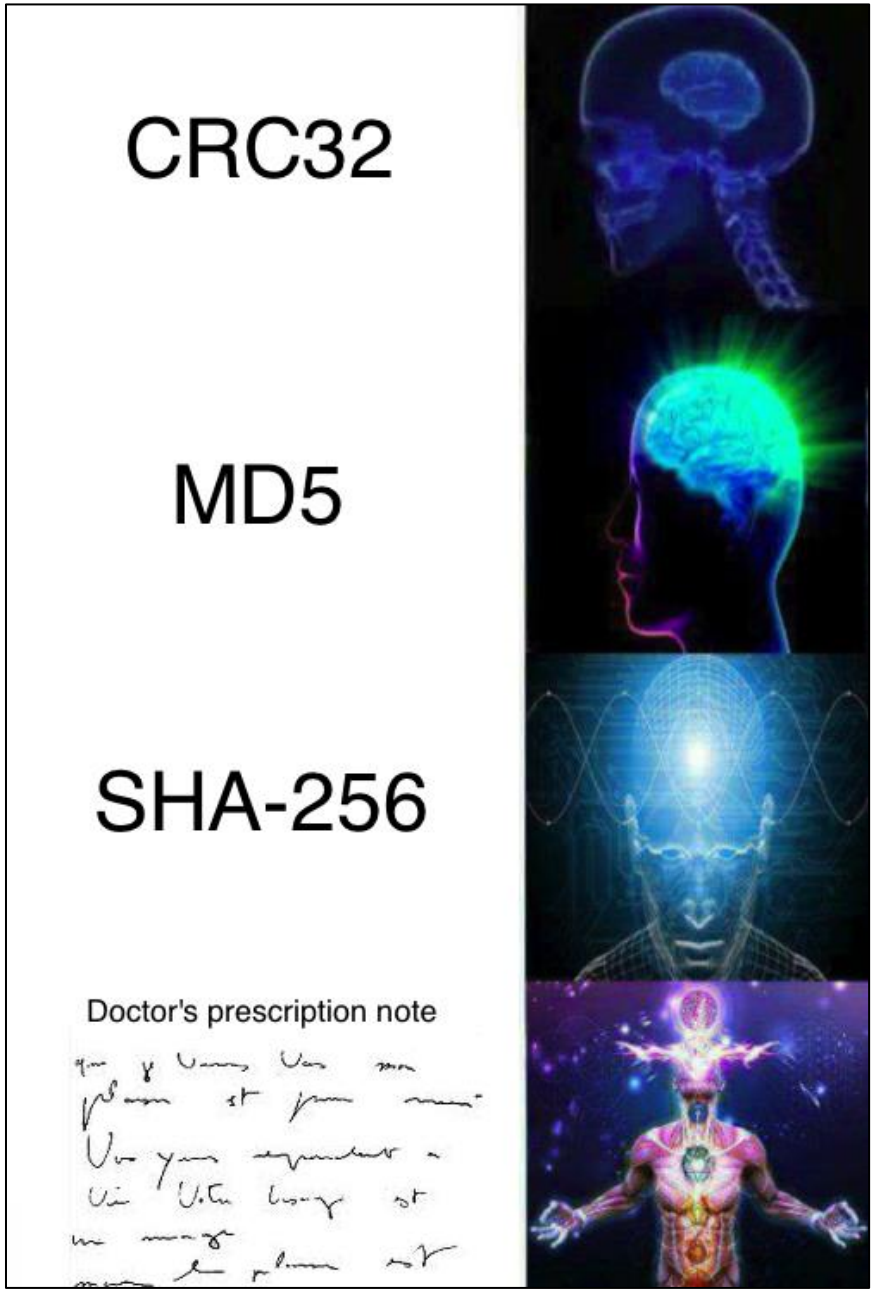
Reese Pearsall

Fall 2024

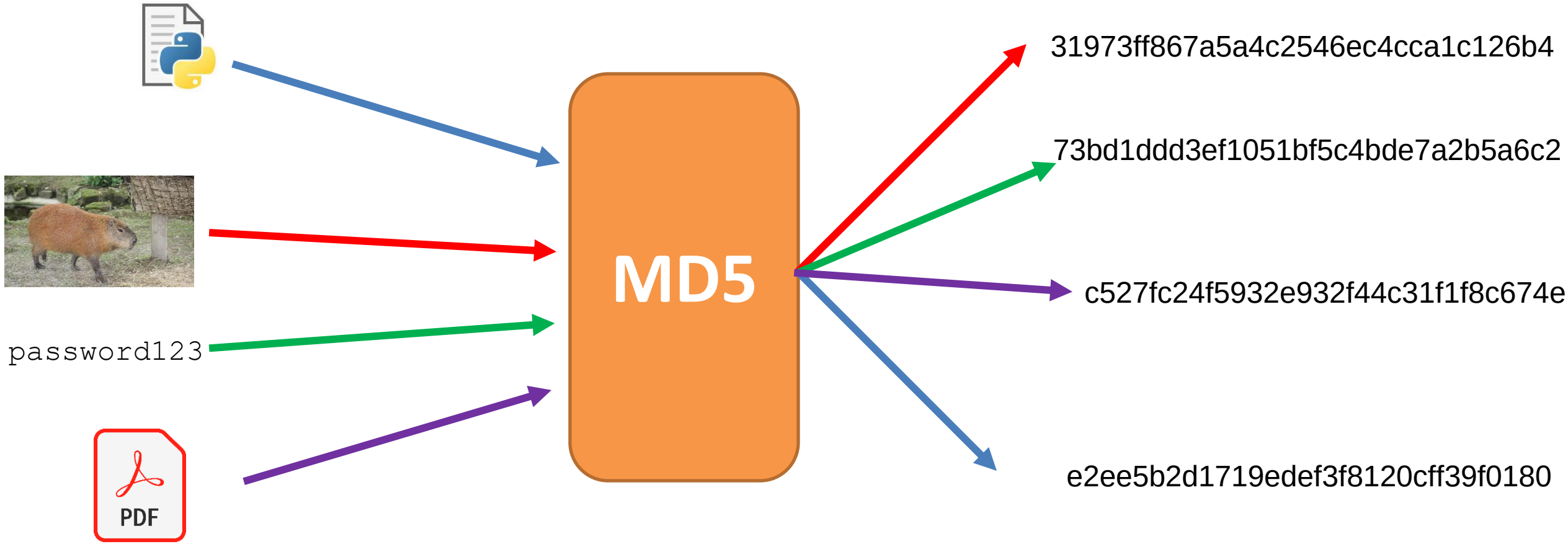
Announcements

Project due a week from today

Lab 8 due Sunday 11/24

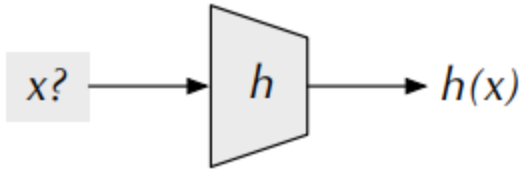


Applications of Hashing

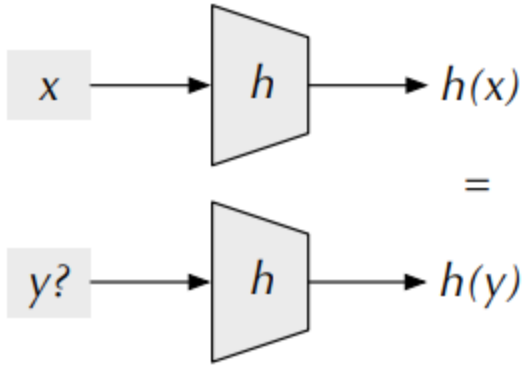


Hash Functions Properties

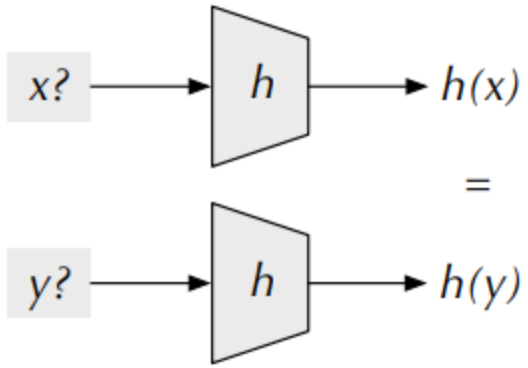
- **Preimage Resistance ("One-Way")**
  Given h(x) = z, hard to find x
  (or any input that hashes to z for that matter)

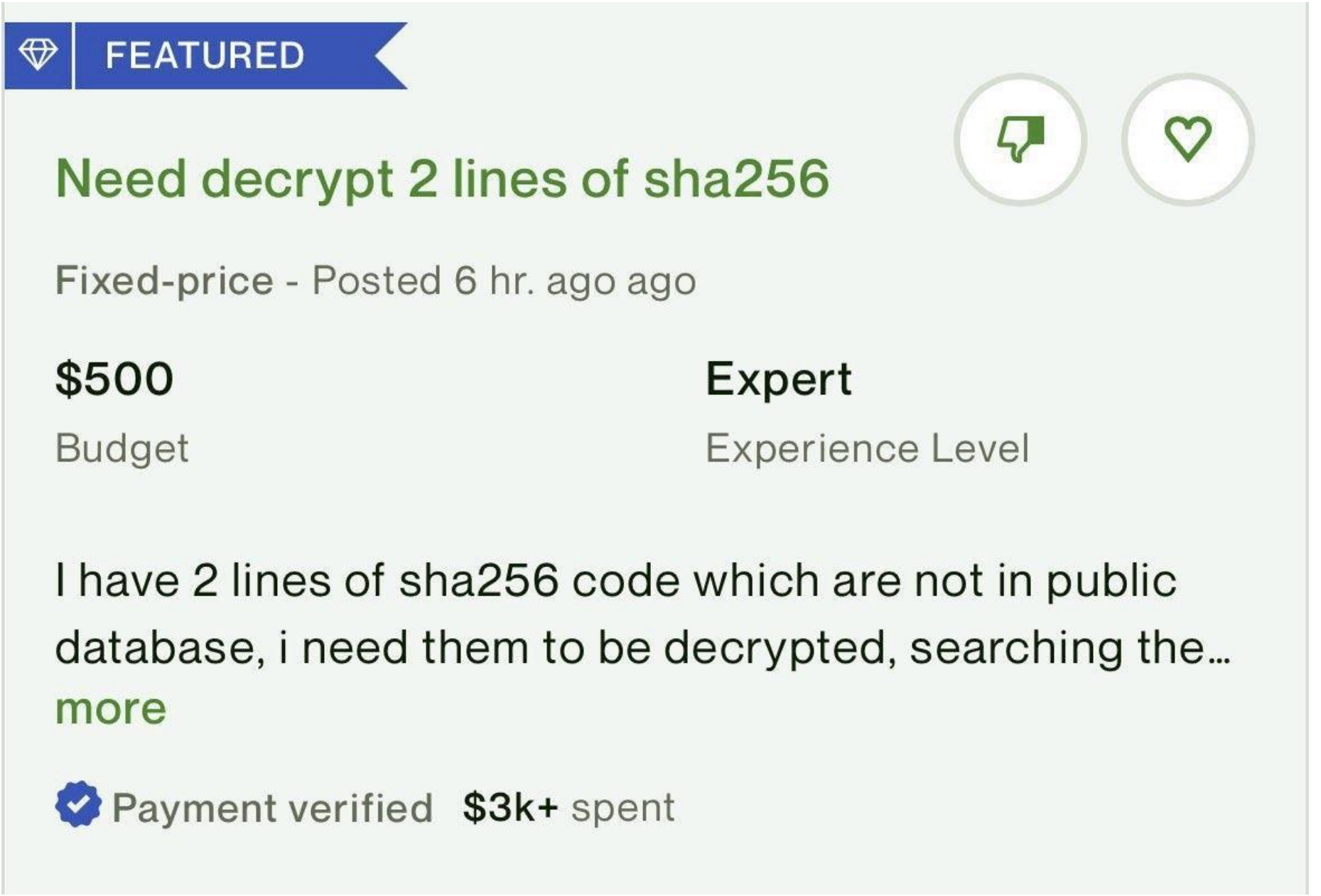


- **Second Preimage Resistance**
  Given x and h(x), hard to find y s.t. h(x) = h(y)



- Collision Resistance (or, ideally, "Collision Free")
  Difficult to find x and y s.t. hash(x) = hash(y)




FEATURED
Need decrypt 2 lines of sha256
Fixed-price - Posted 6 hr. ago ago
$500
Budget
Expert
Experience Level
I have 2 lines of sha256 code which are not in public database, i need them to be decrypted, searching the...
more
Payment verified $3k+ spent

Hash Functions Properties (tl;dr)

```
[11/15/22]seed@VM:~$ md5sum capy.bmp
bb52593852da21b95a8ab8ce64ca7261  capy.bmp
```

Gives an arbitrary size input a fixed-size unique* hash identifier

Hash values are very difficult to **reverse.** They were designed to be one-way

The go-to way to reverse a hash is through brute force

Brute Force Approaches

Long time, and for very unfeasible for cryptographically secure hash functions

Given a hashed password, can you brute force the original password?

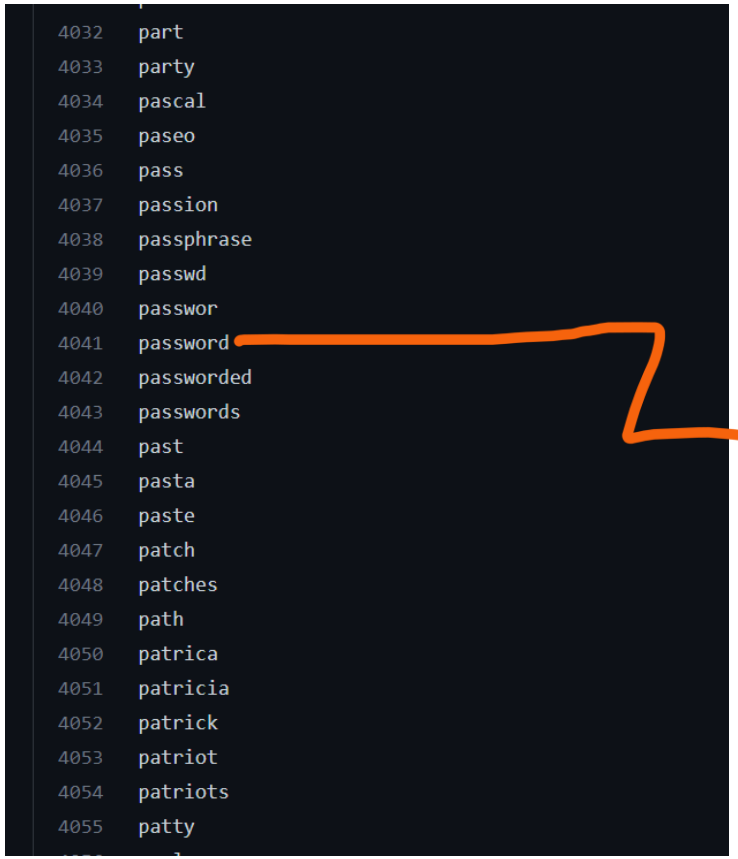
```
afc285bebb3dd733796cb06db01cd59a
```

Techniques

- Dictionary Attack
- Rainbow Tables

Dictionary Attack

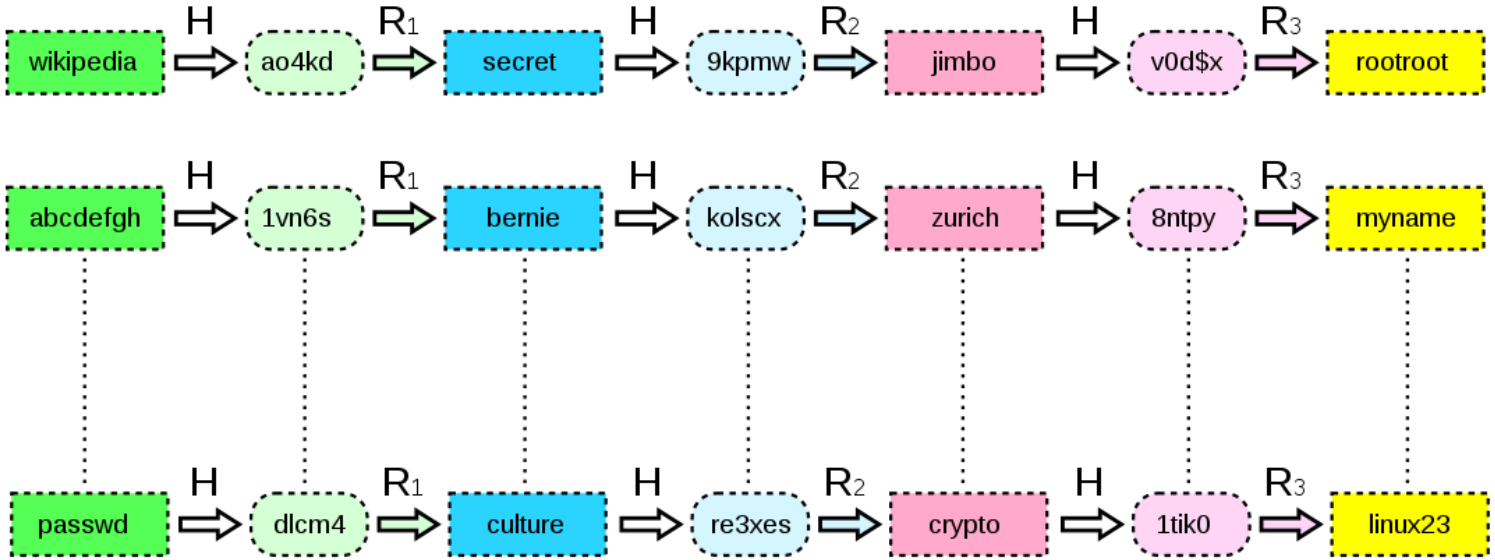
We will use an existing list of common passwords



1. Iterate through each line of file
2. Compute hash of word
3. Check for match

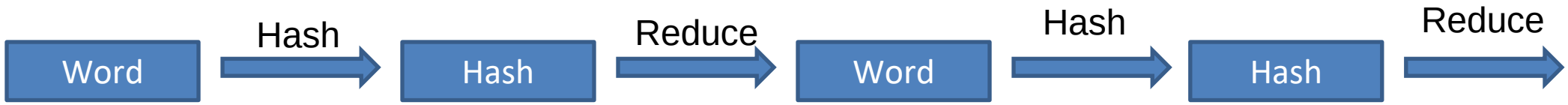
This works for cracking weak, unsalted passwords

Rainbow Tables



A **large** file of pre-computed hashes

Efficient way to store password hashes. Consists of plaintext-hash chains



Looking up a value in the rainbow table can happen quick, but these files are typically very large

Not efficient for complex, salted passwords

(Brute force can take years, with rainbow tables, it can take weeks/months)

Rainbow Tables

Tables for alphanumeric, special character passwords can take a long time to generate, so instead of doing it yourself, you can buy rainbow tables that other people have generated!

There are free, open-source tools that can generate rainbow tables for you

- Project-RainbowCrack

Rainbow Tables using RainbowCrack

```
Reese@DESKTOP-87PAGSR MINGW64 ~/Downloads/rainbowcrack-1.8-win64/rainbowcrack-1.8-win64
$ ./rtgen md5 loweralpha-numeric 1 4 0 3800 100000 0

rainbow table md5_loweralpha-numeric#1-4_0_3800x100000_0.rt parameters
hash algorithm:         md5
hash length:            16
charset name:           loweralpha-numeric
charset data:           abcdefghijklmnopqrstuvwxyz0123456789
charset data in hex:    61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76 77 78 79 7a 30 31 32 33 34 35 36 37 38 39
charset length:         36
plaintext length range: 1 - 4
reduce offset:          0x00000000
plaintext total:        1727604

sequential starting point begin from 0 (0x0000000000000000)
generating...
100000 of 100000 rainbow chains generated (0 m 5.4 s)
```

1

2

```
Reese@DESKTOP-87PAGSR MINGW64 ~/Downloads/rainbowcrack-1.8-win64/rainbowcrack-1.8-win64
$ ./rtsort .
```

```
Reese@DESKTOP-87PAGSR MINGW64 ~/Downloads/rainbowcrack-1.8-win64/rainbowcrack-1.8-win64
$ ./rcrack . -h c3b830f9a769b49d3250795223caad4d
2 rainbow tables found
memory available: 3818671308 bytes
memory for rainbow chain traverse: 60800 bytes per hash, 60800 bytes for 1 hashes
memory for rainbow table buffer: 2 x 4000016 bytes
disk: .\md5_loweralpha-numeric#1-4_0_3800x100000_0.rt: 1600000 bytes read
disk: .\md5_loweralpha-numeric#1-6_0_3800x250000_0.rt: 4000000 bytes read
disk: finished reading all files
plaintext of c3b830f9a769b49d3250795223caad4d is aja

statistics
--------------------------------------------------------------------
plaintext found:                            1 of 1
total time:                                 0.14 s
time of chain traverse:                     0.13 s
time of alarm check:                        0.00 s
time of disk read:                          0.00 s
hash & reduce calculation of chain traverse: 7216200
hash & reduce calculation of alarm check:   586
number of alarm:                            390
performance of chain traverse:              57.27 million/s
performance of alarm check:                 0.59 million/s

result
--------------------------------------------------------------------
c3b830f9a769b49d3250795223caad4d  aja  hex:616a61
```

3

Hash Collisions

Goal: Create two **different files** with the **same md5 hash**

Our **ultimate goal** would be to create two executables (one benign, one malicious) with the same hash

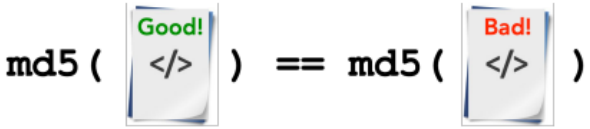
Motivation

Forging public-key certificates

- Assume two certificate requests for www.example.com and www.attacker.com have same hash due to a collision
- CA signing of either request would be equivalent
- Attacker can get certificate signed for www.example.com without owning it!

Integrity of Programs

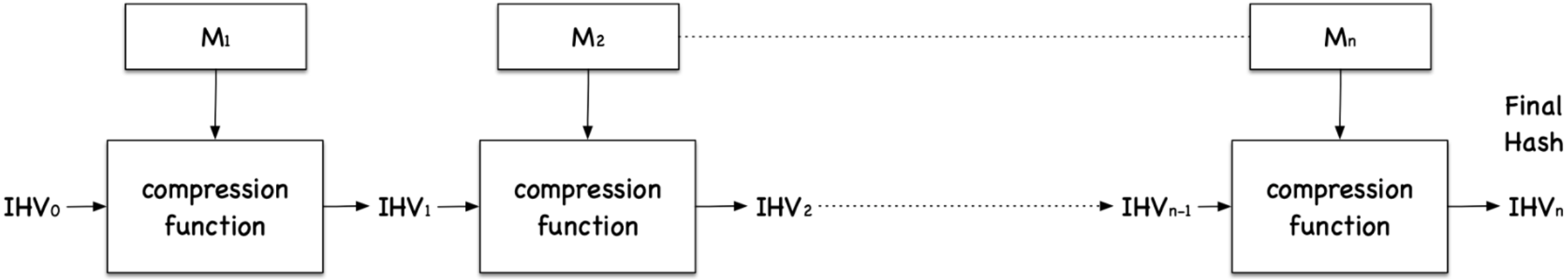
- Ask CA to sign a legitimate program's hash
- Attacker creates a malicious program with same hash
- The certificate for legitimate program is also valid for malicious version



Hash Collisions (MD5collgen)

On our VM, we have a tool called **md5collgen** that will generate two files with the **same prefix** ⟸ We get to choose this prefix!

MD5 Under the hood



Fact: Message is divided into blocks, and each block is run through a compression function

Important Fact: Each block will be 64 bytes

Hash Collisions (MD5collgen)

On our VM, we have a tool called **md5collgen** that will generate two files with the **same prefix** ⟵ We get to choose this prefix!

```
[11/17/22]seed@VM:~/.../example$ echo "I am a prefix!" > prefix.txt
[11/17/22]seed@VM:~/.../example$ ls -ld prefix.txt
-rw-rw-r-- 1 seed seed 15 Nov 17 15:16 prefix.txt
```

Hash Collisions (MD5collgen)

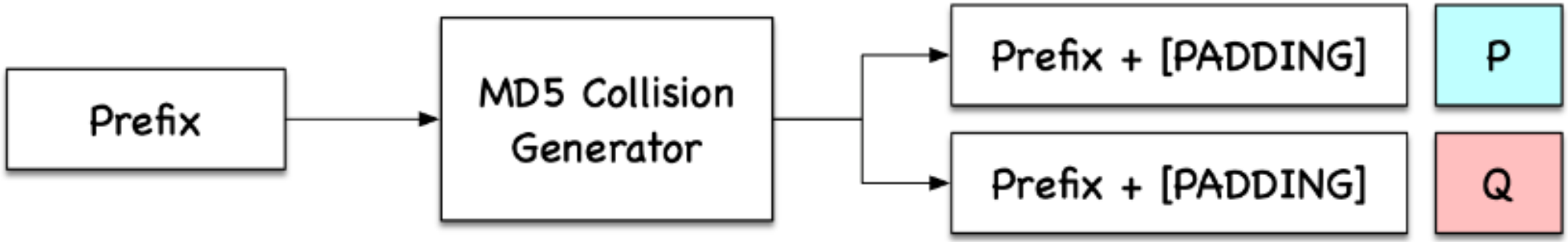
On our VM, we have a tool called **md5collgen** that will generate two files with the **same prefix** ⟵ We get to choose this prefix!

```
[11/17/22]seed@VM:~/.../example$ echo "I am a prefix!" > prefix.txt
[11/17/22]seed@VM:~/.../example$ ls -ld prefix.txt
-rw-rw-r-- 1 seed seed 15 Nov 17 15:16 prefix.txt
```

```
[11/17/22]seed@VM:~/.../example$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: 1eb37d6bfcb868196d9e93aacce724e2

Generating first block: ........................................
Generating second block: S00.......
Running time: 37.3691 s
```

Hash Collisions (MD5collgen)

What if out prefix is a multiple of 64?



Hash Collisions (MD5collgen)

On our VM, we have a tool called **md5collgen** that will generate two files with the **same prefix**

We get to choose this prefix!

```
[11/17/22]seed@VM:~/.../example$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: 1eb37d6bfcb868196d9e93aacce724e2

Generating first block: ........................................
Generating second block: S00.....
Running time: 37.3691 s


[11/17/22]seed@VM:~/.../example$ ls -al
total 20
drwxrwxr-x 2 seed seed 4096 Nov 17 15:17 .
drwxrwxr-x 4 seed seed 4096 Nov 17 15:15 ..
-rw-rw-r-- 1 seed seed  192 Nov 17 15:17 out1.bin
-rw-rw-r-- 1 seed seed  192 Nov 17 15:17 out2.bin
-rw-rw-r-- 1 seed seed   15 Nov 17 15:16 prefix.txt
[11/17/22]seed@VM:~/.../example$ md5sum out1.bin
35993d8b2dde3df7fee8186426cb4f2b  out1.bin
[11/17/22]seed@VM:~/.../example$ md5sum out2.bin
35993d8b2dde3df7fee8186426cb4f2b _out2.bin
```

Same Hash!

Hash Collisions (MD5collgen)

On our VM, we have a tool called **md5collgen** that will generate two files with the **same prefix**

We get to choose this prefix!

```
[11/17/22]seed@VM:~/.../example$ md5collgen -p prefix.txt -o out1.bin out2.bin
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1.bin' and 'out2.bin'
Using prefixfile: 'prefix.txt'
Using initial value: 1eb37d6bfcb868196d9e93aacce724e2

Generating first block: ........................................
Generating second block: S00.....
Running time: 37.3691 s


[11/17/22]seed@VM:~/.../example$ ls -al
total 20
drwxrwxr-x 2 seed seed 4096 Nov 17 15:17 .
drwxrwxr-x 4 seed seed 4096 Nov 17 15:15 ..
-rw-rw-r-- 1 seed seed  192 Nov 17 15:17 out1.bin
-rw-rw-r-- 1 seed seed  192 Nov 17 15:17 out2.bin
-rw-rw-r-- 1 seed seed   15 Nov 17 15:16 prefix.txt
[11/17/22]seed@VM:~/.../example$ md5sum out1.bin
35993d8b2dde3df7fee8186426cb4f2b  out1.bin
[11/17/22]seed@VM:~/.../example$ md5sum out2.bin
35993d8b2dde3df7fee8186426cb4f2b _out2.bin
```

Same Hash!

Compare with xxd

Hash Collisions (MD5collgen)

What if out prefix is a multiple of 64?

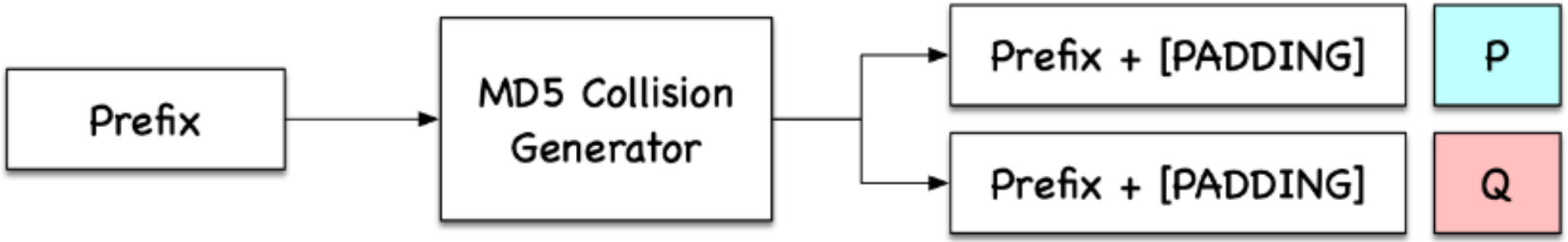
```
[11/17/22]seed@VM:~/.../07_hash$ echo "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!
" > prefix64.txt
[11/17/22]seed@VM:~/.../07_hash$ ls -al
total 232
drwxrwxr-x  4 seed seed   4096 Nov 17 15:34 .
drwxrwxr-x 14 seed seed   4096 Oct 27 12:00 ..
-rw-rw-r--  1 seed seed   1266 Oct 27 12:00 benign_evil.c
-rw-rw-r--  1 seed seed    693 Oct 27 12:00 calculate_sha256.c
drwxrwxr-x  2 seed seed   4096 Oct 27 12:00 demo_md5collgen
drwxrwxr-x  2 seed seed   4096 Nov 17 15:17 example
-rw-rw-r--  1 seed seed    719 Oct 27 12:00 find_nonce.c
-rw-rw-r--  1 seed seed 184974 Oct 27 12:00 pic_original.bmp
-rw-rw-r--  1 seed seed     64 Nov 17 15:34 prefix64.txt
-rw-rw-r--  1 seed seed   1386 Oct 27 12:00 print_array.c
-rw-rw-r--  1 seed seed     51 Oct 27 12:00 README.md
-rw-rw-r--  1 seed seed    749 Oct 27 12:00 sha256_length_extension.c
-rw-rw-r--  1 seed seed    537 Oct 27 12:00 sha256_padding.c
[11/17/22]seed@VM:~/.../07_hash$ md5collgen -p prefix64.txt -o out1.bin out2.bin
```

Our prefix is exactly 64 bytes
→ No padding is added!

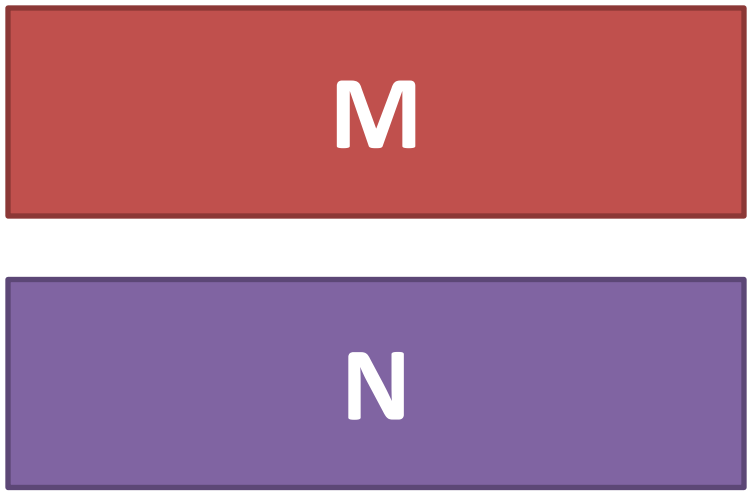
```
[11/17/22]seed@VM:~/.../07_hash$ xxd out1.bin
00000000: 6162 6364 6566 6768 696a 6b6c 6d6e 6f70  abcdefghijklmnop
00000010: 7172 7374 7576 7778 797a 4142 4344 4546  qrstuvwxyzABCDEF
                  4a 4b4c 4d4e 4f50 5152 5354 5556  GHIJKLMNOPQRSTUV
                  5a 3031 3233 3435 3637 3839 210a  WXYZ0123456789!.
                  23 59e5 b79c ce98 92a0 b122 918c  ^.~#Y........"..
                  c3 14b1 4b0a 591e 8139 6a2a c26d  ?.:...K.Y..9j*.m
                  c7 7cc5 0d68 0b02 d253 b15d d615  ....|..h...S.]..
                  c5 c79b 666c 9f66 e355 8678 44c0  .!<...fl.f.U.xD.
                  8e cbf5 d8f6 b16e 0f61 354e 5c42  .`.......n.a5N\B
                  83 03e6 2533 cb5a fecb ec06 fe6f  .}....%3.Z.....o
                  59 04d1 df0d 682a 4dd7 a134 d2ee  .#&Y....h*M..4..
                  1c d348 e152 11ae 7d5a 3557 47d1  .....H.R..}Z5WG.
                  :~/.../07_hash$ xxd out2.bin
                  64 6566 6768 696a 6b6c 6d6e 6f70  abcdefghijklmnop
                  74 7576 7778 797a 4142 4344 4546  qrstuvwxyzABCDEF
                  4a 4b4c 4d4e 4f50 5152 5354 5556  GHIJKLMNOPQRSTUV
00000030: 5758 595a 3031 3233 3435 3637 3839 210a  WXYZ0123456789!.
00000040: 5ea5 7e23 59e5 b79c ce98 92a0 b122 918c  ^.~#Y........"..
00000050: 3fc6 3a43 14b1 4b0a 591e 8139 6a2a c26d  ?.:C..K.Y..9j*.m
00000060: dfc8 b3c7 7cc5 0d68 0b02 d253 b1dd d615  ....|..h...S....
00000070: ff21 3cc5 c79b 666c 9f66 e3d5 8678 44c0  .!<...fl.f...xD.
00000080: 0c60 df8e cbf5 d8f6 b16e 0f61 354e 5c42  .`.......n.a5N\B
00000090: a87d 0d03 03e6 2533 cb5a fecb ec06 fe6f  .}....%3.Z.....o
000000a0: ae23 2659 04d1 df0d 682a 4dd7 a1b4 d1ee  .#&Y....h*M.....
000000b0: ff15 ba1c d348 e152 11ae 7dda 3557 47d1  .....H.R..}.5WG.
```

Hash Collisions (MD5collgen)

What if out prefix is a multiple of 64?

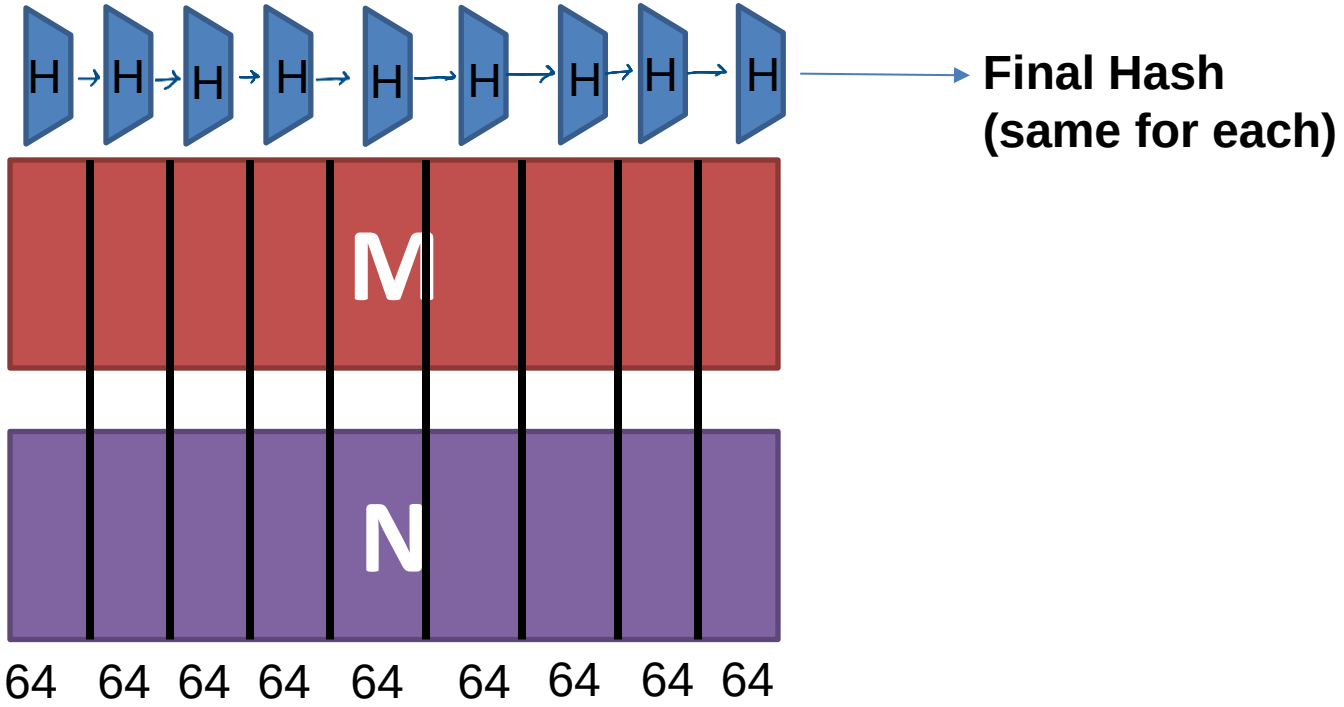


Hash Collisions (Suffix Extension)



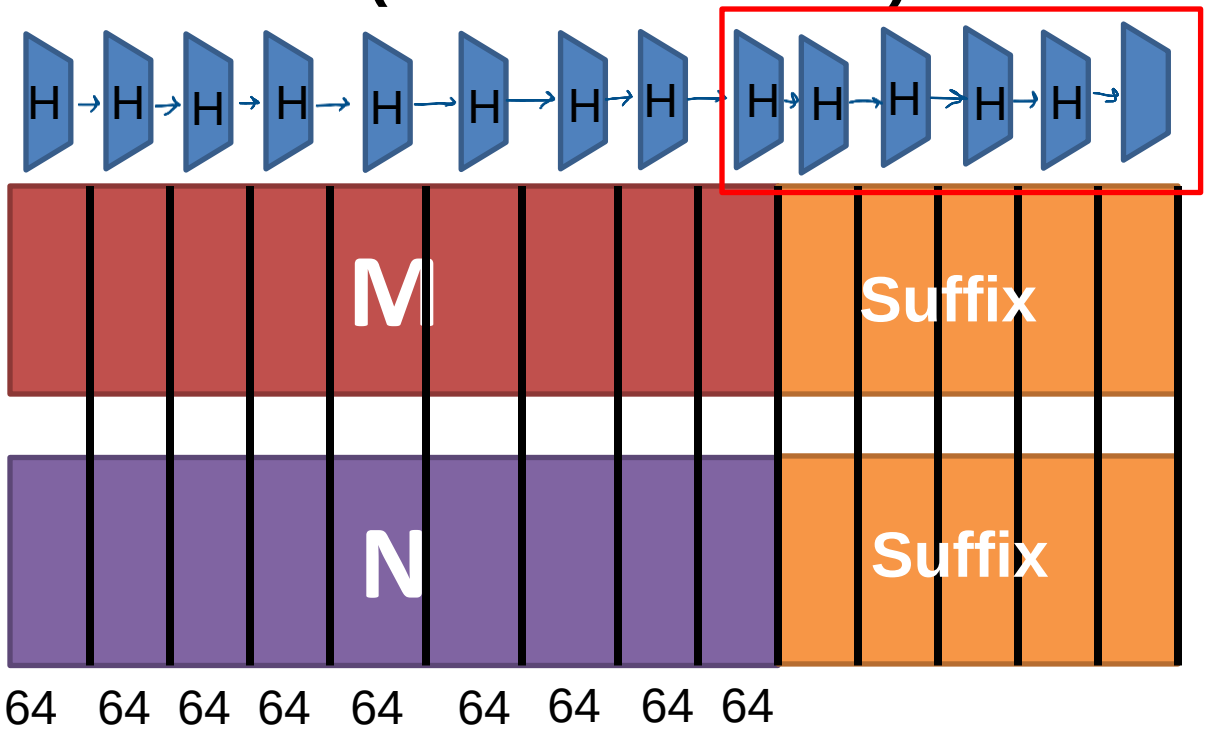
H(m) == H(n)

Hash Collisions (Suffix Extension)



H(m) == H(n)

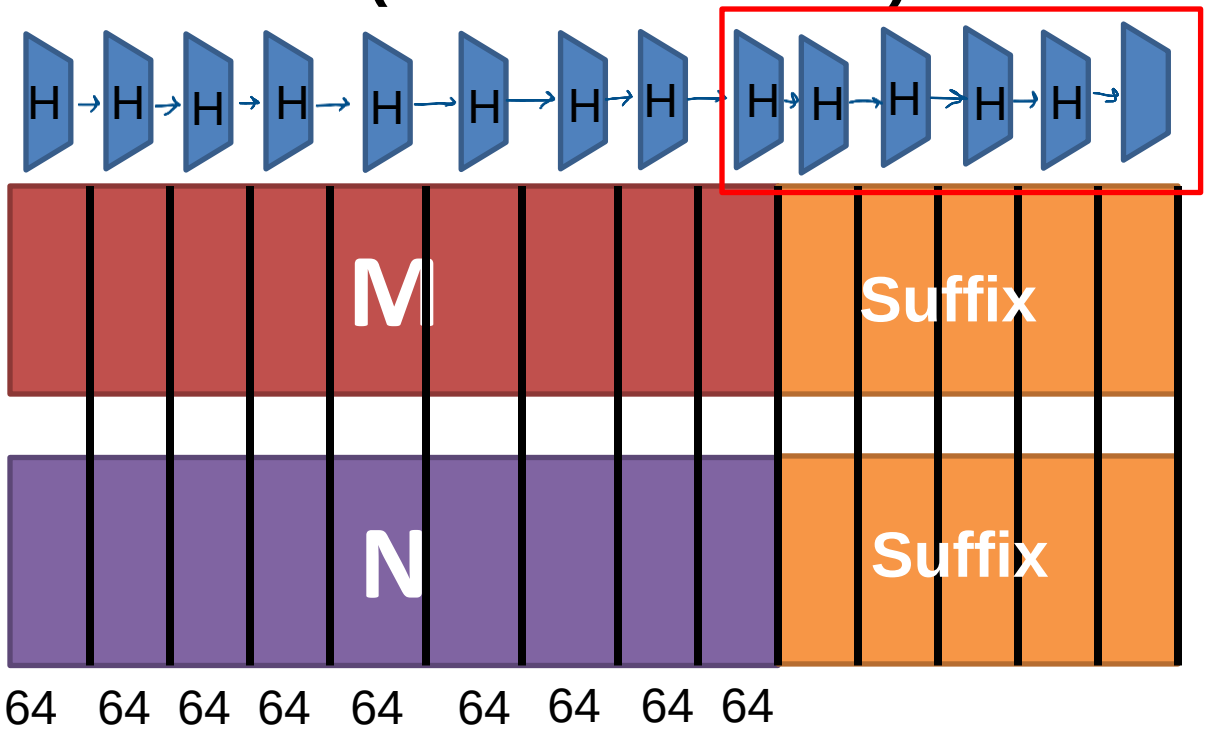
Hash Collisions (Suffix Extension)



If we append the same suffix, then this computation will also be the exact same for M and N

H(m) == H(n)

Hash Collisions (Suffix Extension)



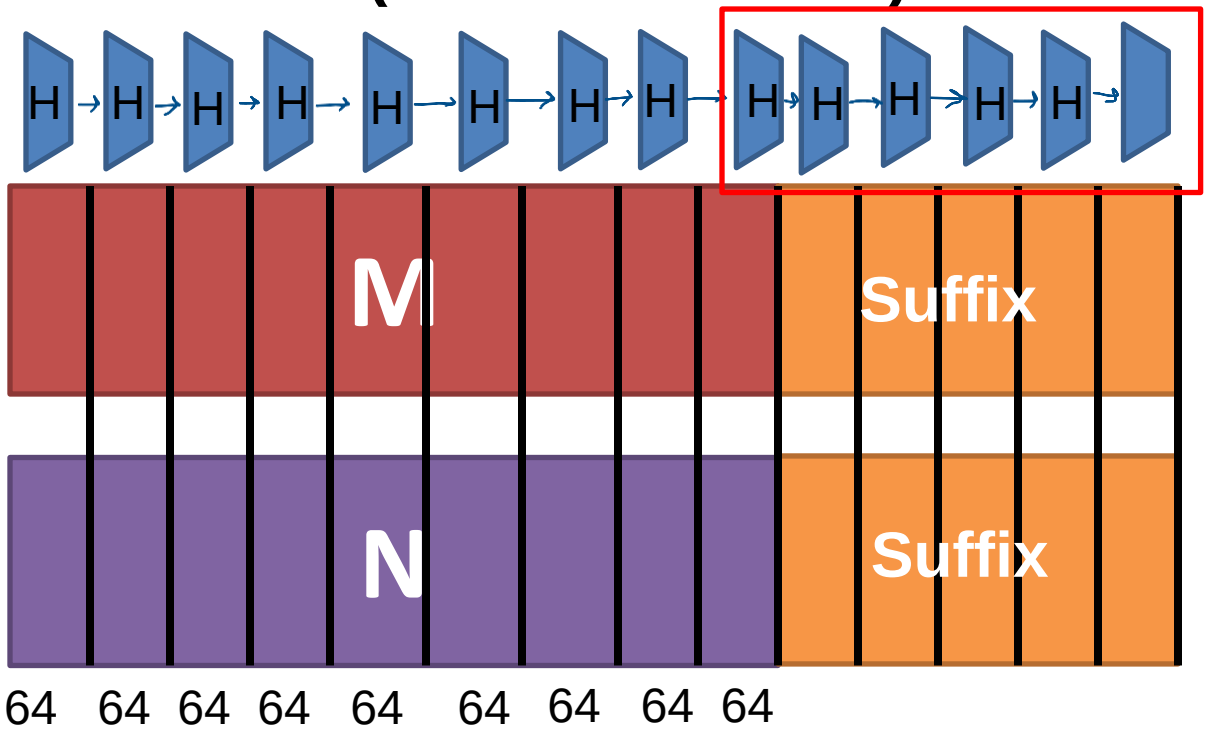
If we append the same suffix, then this computation will also be the exact same for M and N

$$H(m) == H(n)$$

$$H(m \,||\, s) == H(n \,||\, s)$$

s = shared suffix

Hash Collisions (Suffix Extension)



If we append the same suffix, then this computation will also be the exact same for M and N

```
[11/17/22]seed@VM:~/.../07_hash$ echo "suffix" > suffix.txt
[11/17/22]seed@VM:~/.../07_hash$ cat out1.bin suffix.txt > out1suffix.bin
][11/17/22]seed@VM:~/.../07_hash$ cat out2.bin suffix.txt > out2suffix.bin
```

$$H(m) == H(n)$$

$$H(m \,||\, s) == H(n \,||\, s)$$

s = shared suffix

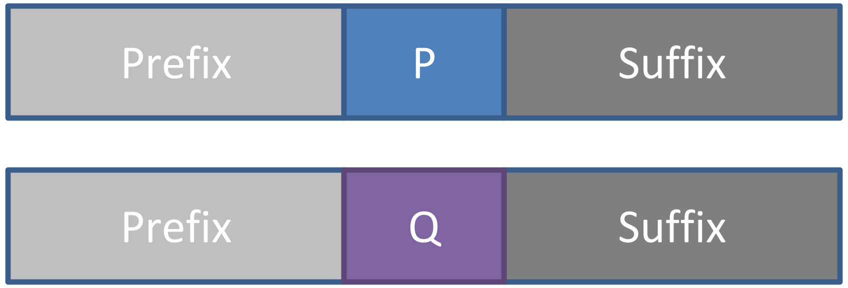
```
[11/17/22]seed@VM:~/.../07_hash$ md5sum out1suffix.bin
a63075af11518048cff11bf3d11a5462  out1suffix.bin
[11/17/22]seed@VM:~/.../07_hash$ md5sum out2suffix.bin
a63075af11518048cff11bf3d11a5462 _out2suffix.bin
```

Hash Collisions (Generating Two executable files with the same MD5 hash)

```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

This is a program that will print out the contents of an array

We will create two variants of this program, but the program will have the same hash



Hash Collisions (Generating Two executable files with the same MD5 hash)

```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

We will create two variants of this program, but the program will have the same hash

Prefix

Suffix

Hash Collisions (Generating Two executable files with the same MD5 hash)

```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

We will create two variants of this program, but the program will have the same hash

md5collgen(Prefix)

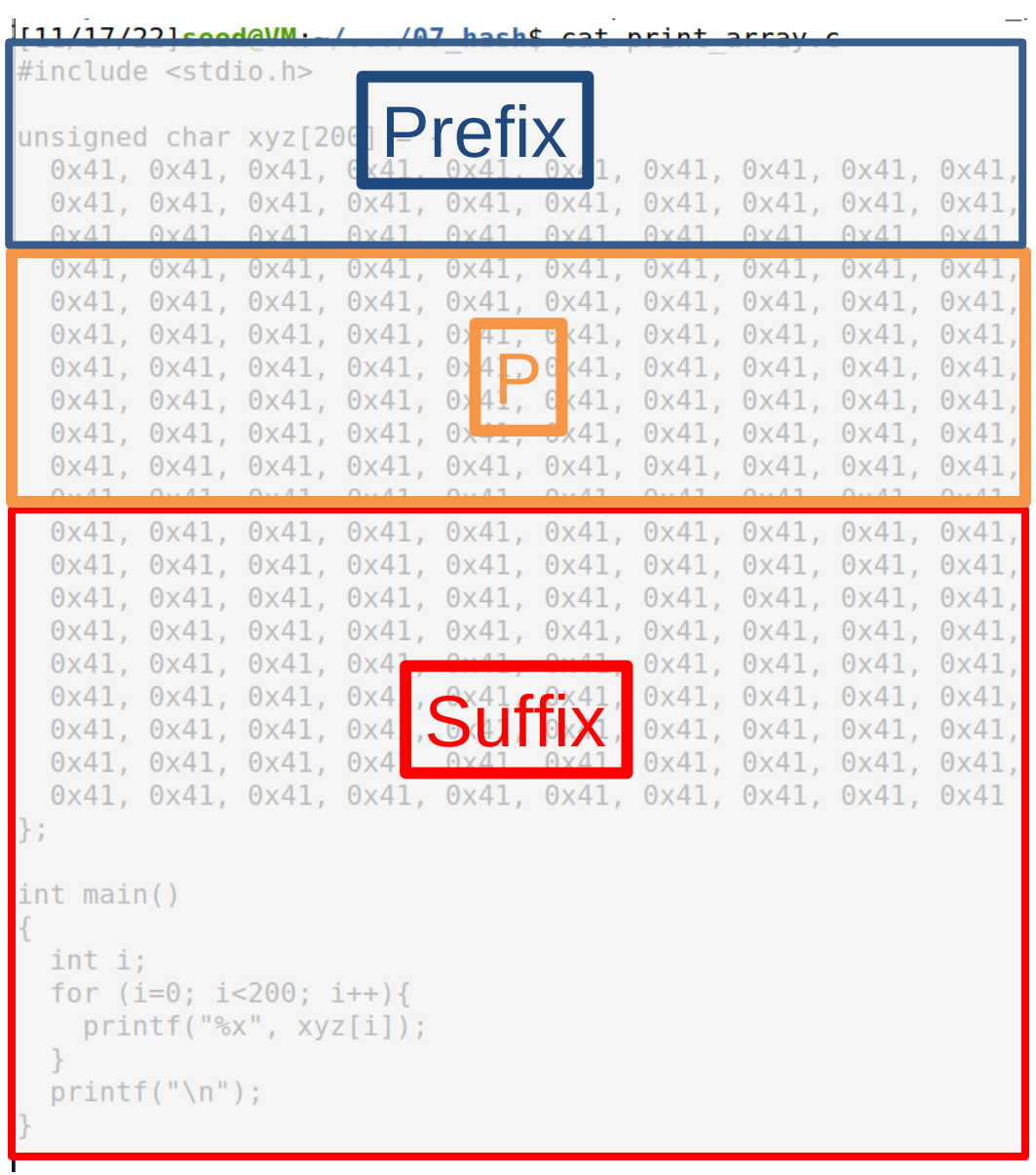
Prefix | P Prefix | Q

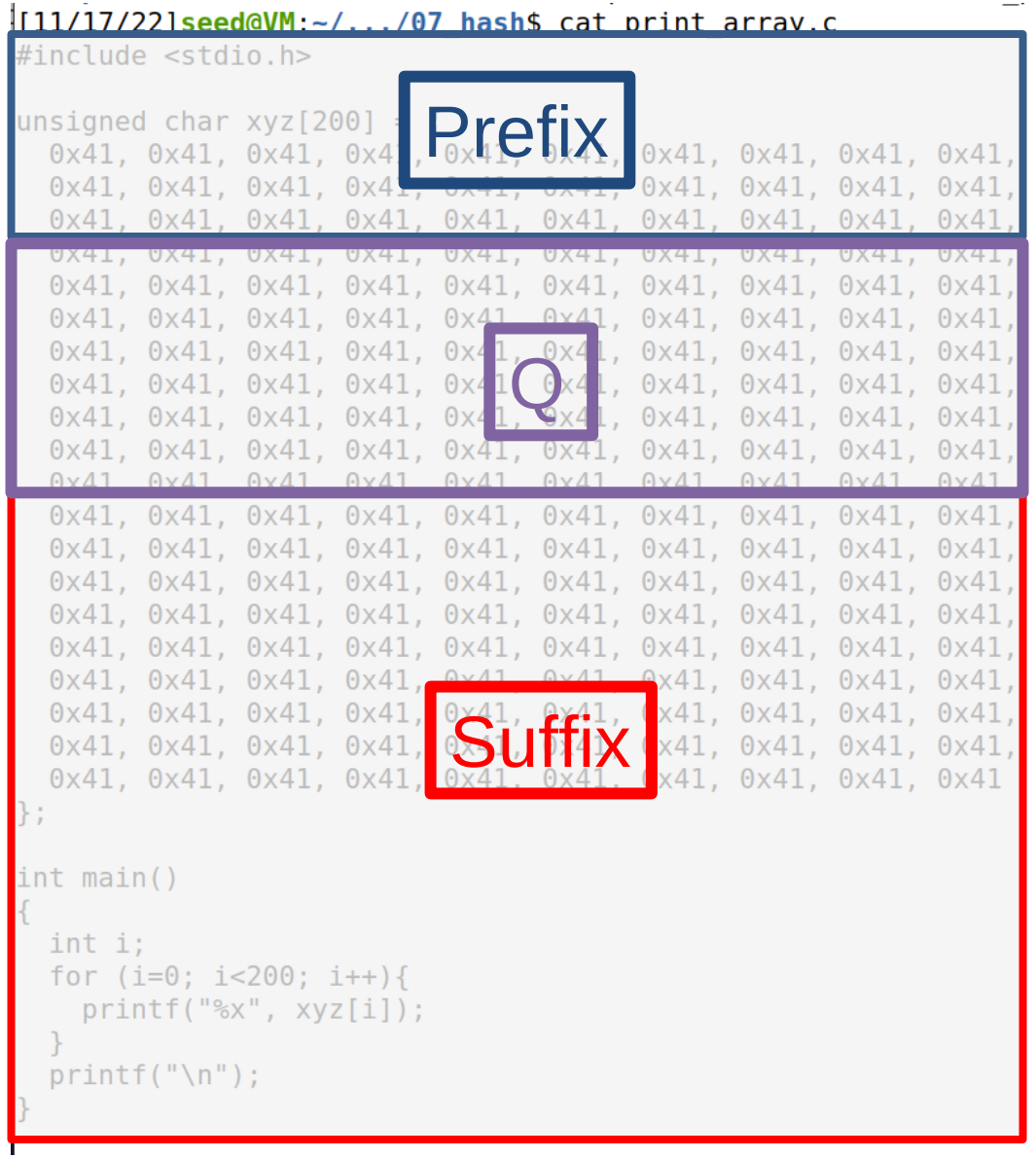
These will have the same hash!
P and Q will be 128 bytes (multiple of 64)

Prefix | P | Suffix Prefix | Q | Suffix

Because we know the suffix extension property holds true, we know the hash of these two programs will also be the same

Hash Collisions (Generating Two executable files with the same MD5 hash)





Hash Collisions (Generating Two executable files with the same MD5 hash but behave very differently)

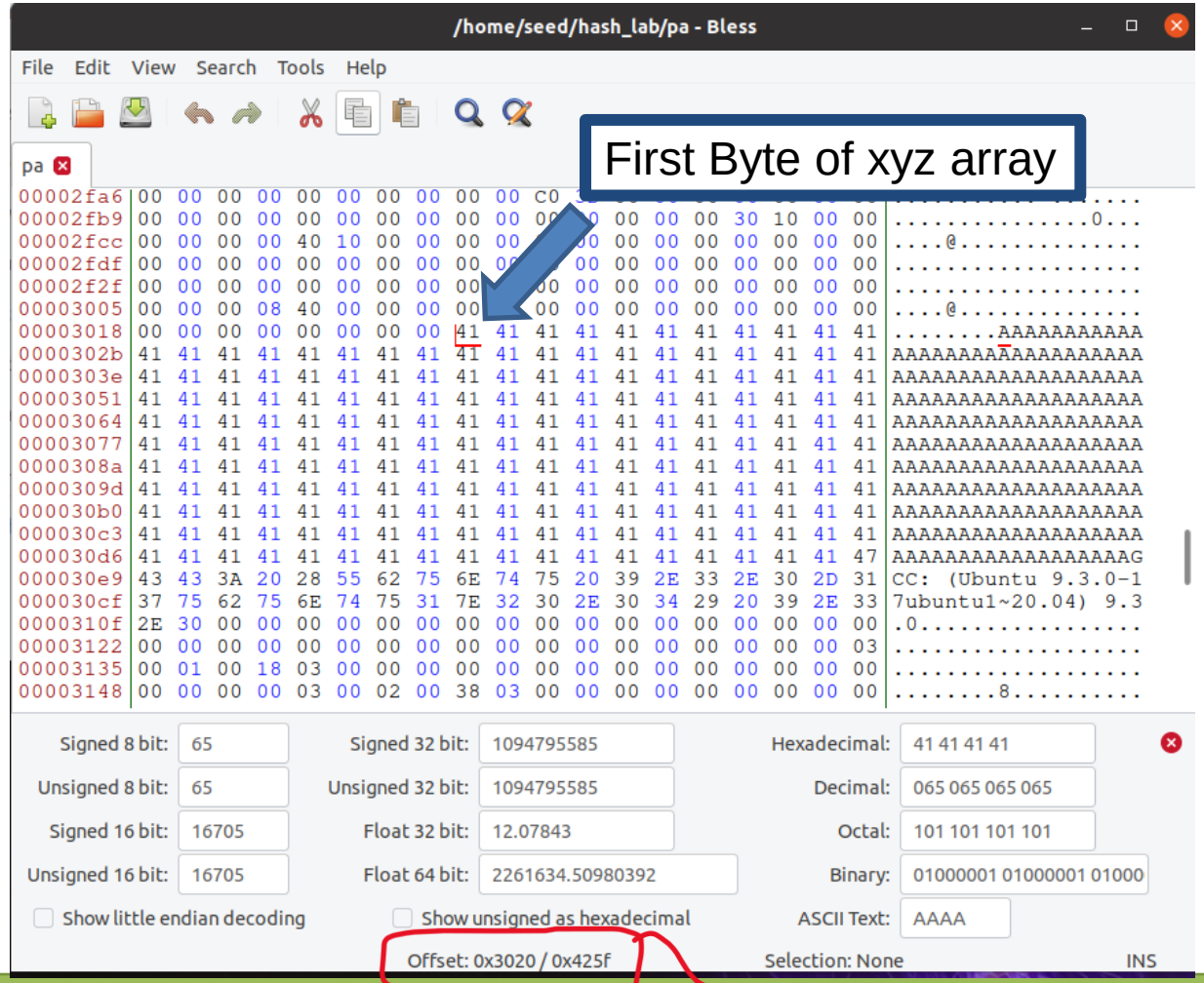
```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

We can change the contents of this section of the program because it is just array data (it won’t break anything)

First, we need to find the starting location (the offset) of the xyz array → this will be the beginning of P and Q

Hash Collisions (Generating Two executable files with the same MD5 hash but behave very differently)

```
[11/17/22]seed@VM:~/hash_lab$ gcc print_array.c -o pa
[11/17/22]seed@VM:~/hash_lab$ bless pa
```



We can find where xyz begins in our program easily, because we filled it with A's

Start of XYZ = 0x3020 (Hexadecimal)
12320 (decimal)

Task 4 on the lab

```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

Prefix

0

12320

Our prefix will be bytes 0-12320 of the program!

We want our **P** and **Q** to be 128 bytes

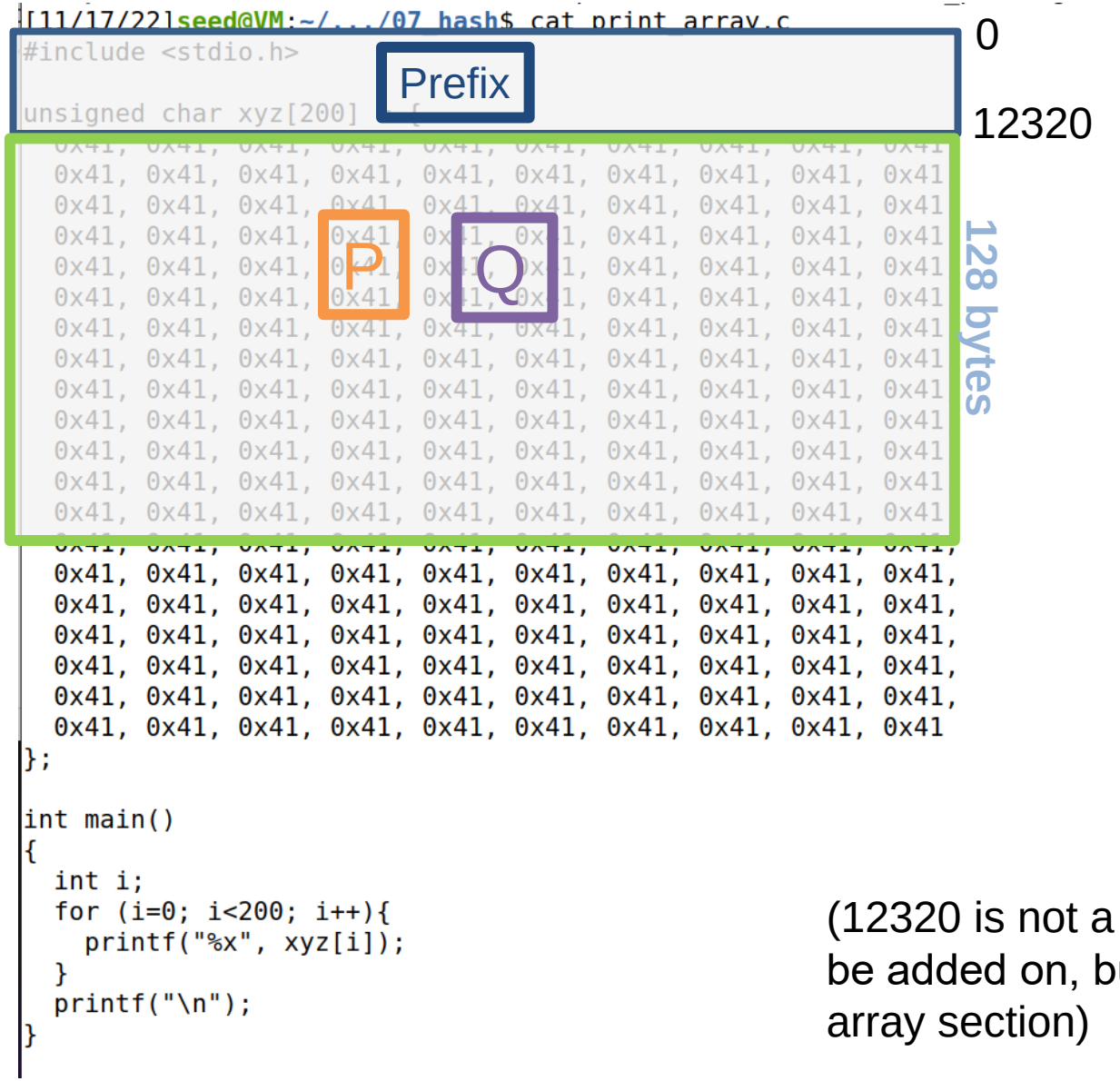
Why 128?

→ Multiple of 64

→ Wont overflow an array of size 200

(12320 is not a multiple of 64, which means that some padding will be added on, but in this case it's fine because it will just go in our array section)

Task 4 on the lab



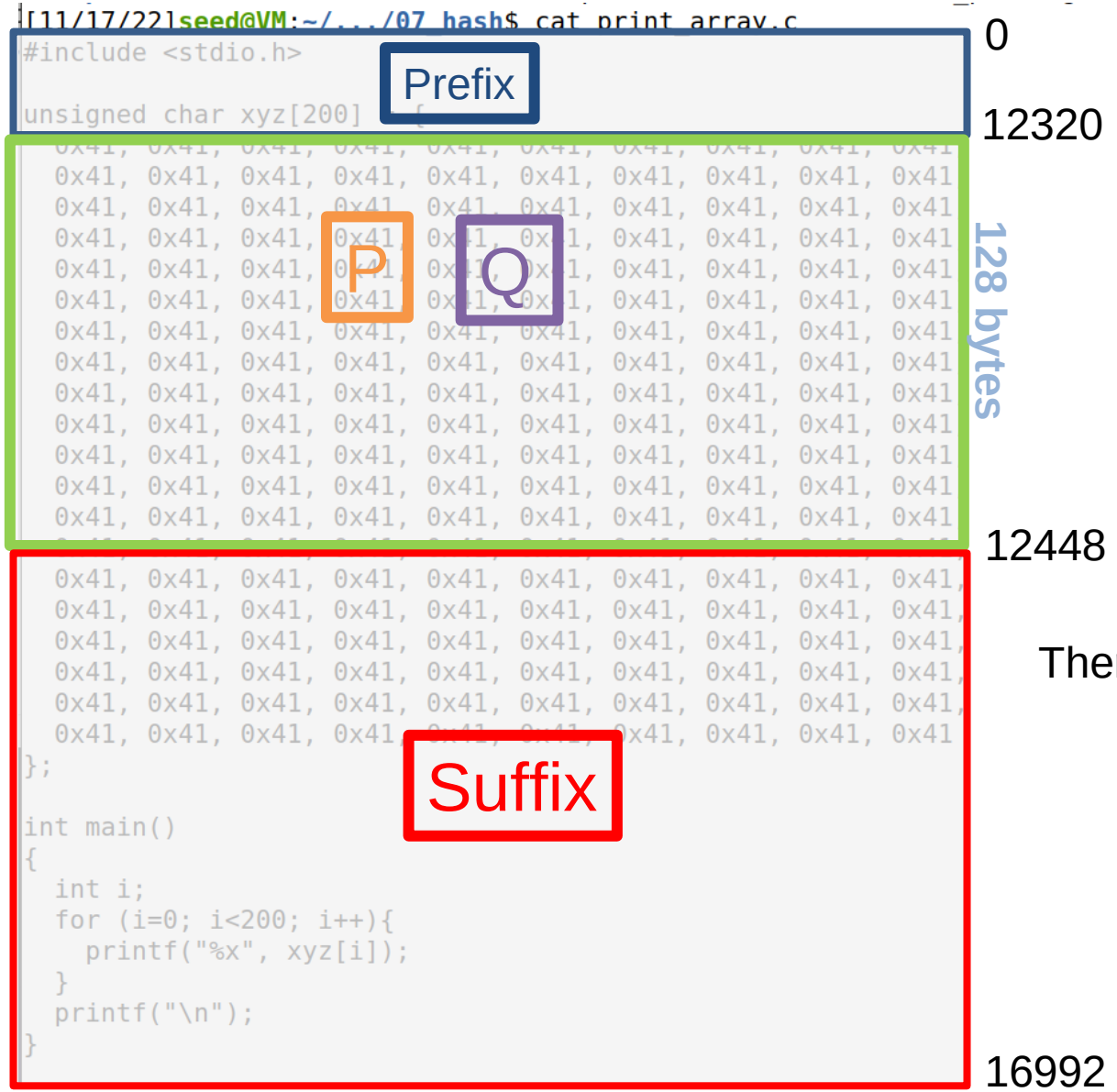
Our prefix will be bytes 0-12320 of the program!

We want our **P** and **Q** to be 128 bytes

Why 128?
→ Multiple of 64
→ Wont overflow an array of size 200

(12320 is not a multiple of 64, which means that some padding will be added on, but in this case it's fine because it will just go in our array section)

Task 4 on the lab



Our prefix will be bytes 0-12320 of the program!

We want our **P** and **Q** to be 128 bytes

Why 128?
→ Multiple of 64
→ Wont overflow an array of size 200

Therefore, our suffix will begin at byte # 12320 + 128 = **12448**

Task 4 on the lab

```
[11/17/22]seed@VM:~/.../07_hash$ cat print_array.c
#include <stdio.h>

unsigned char xyz[200] = {
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
  0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41
};

int main()
{
  int i;
  for (i=0; i<200; i++){
    printf("%x", xyz[i]);
  }
  printf("\n");
}
```

Prefix

P

Q

Suffix

0

12320

128 bytes

12448

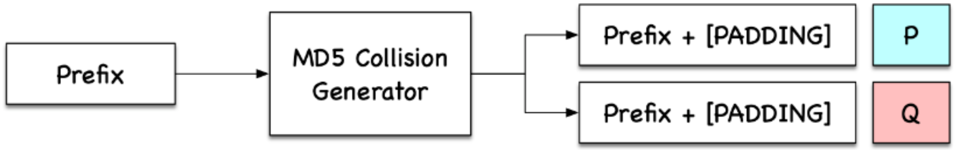
16992 (size of executable)

Get contents of prefix and suffix

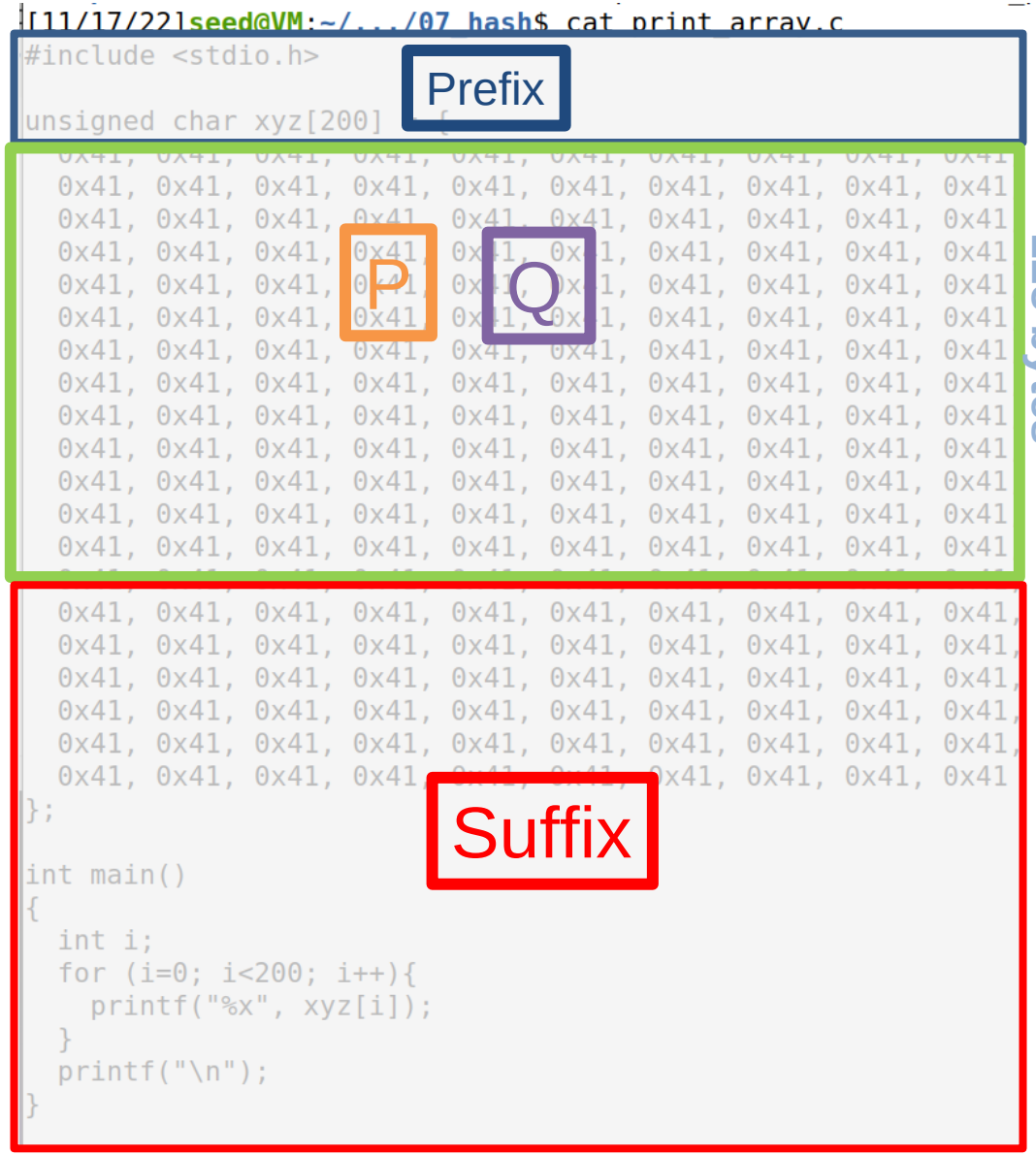
```
[11/17/22]seed@VM:~/hash_lab$ head -c 12320 pa > prefix
[11/17/22]seed@VM:~/hash_lab$ tail -c +12448 pa > suffix
```

Use collision tool to get (prefix + P) and (prefix + Q)

```
[11/17/22]seed@VM:~/hash_lab$ md5collgen -p prefix -o prefix_and_P prefix_and_Q
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'prefix_and_P' and 'prefix_and_Q'
Using prefixfile: 'prefix'
Using initial value: fa3f7a62525b9c90471862a4a04139a5

Generating first block: ..
Generating second block: S01..
Running time: 1.78726 s
```



Task 4 on the lab



1

Get contents of prefix and suffix

```
[11/17/22]seed@VM:~/hash_lab$ head -c 12320 pa > prefix
[11/17/22]seed@VM:~/hash_lab$ tail -c +12448 pa > suffix
```

a

Use collision tool to get (prefix + P) and (prefix + Q)

```
[11/17/22]seed@VM:~/hash_lab$ md5collgen -p prefix -o prefix_and_P prefix_and_Q
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'prefix_and_P' and 'prefix_and_Q'
Using prefixfile: 'prefix'
Using initial value: fa3f7a62525b9c90471862a4a04139a5

Generating first block: ..
Generating second block: S01..
Running time: 1.78726 s
```

3

Add suffix to programs

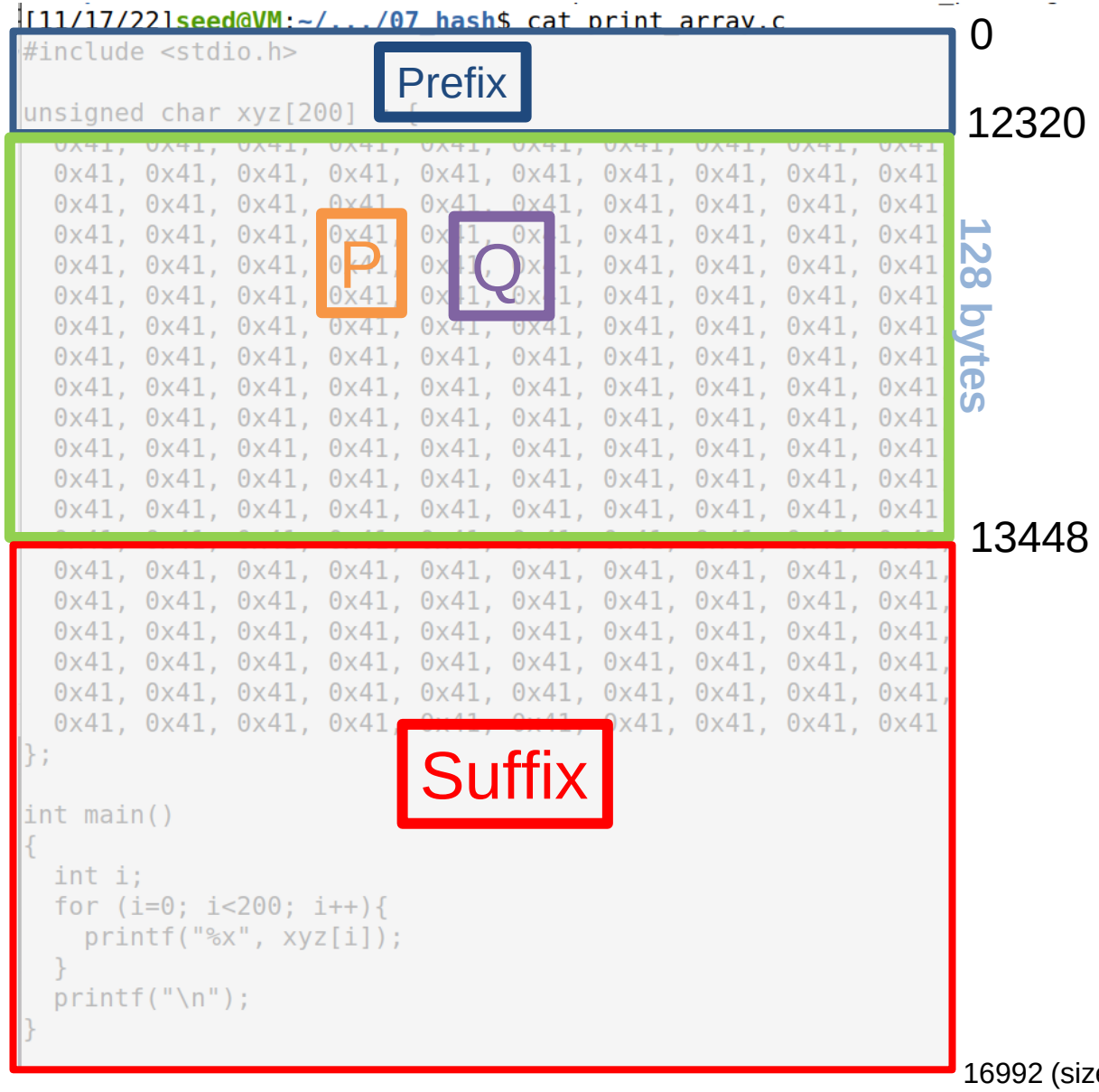
```
[11/17/22]seed@VM:~/hash_lab$ cat prefix_and_P suffix > program1.out
[11/17/22]seed@VM:~/hash_lab$ cat prefix_and_Q suffix > program2.out
```

4

Verify that executables are different, but have the same hash

```
[11/17/22]seed@VM:~/hash_lab$ diff program1.out program2.out
Binary files program1.out and program2.out differ
[11/17/22]seed@VM:~/hash_lab$ md5sum program1.out
f489a326ed9c692f31eabccab06062ce  program1.out
[11/17/22]seed@VM:~/hash_lab$ md5sum program2.out
f489a326ed9c692f31eabccab06062ce  program2.out
```

Task 4 on the lab



0

12320

128 bytes

13448

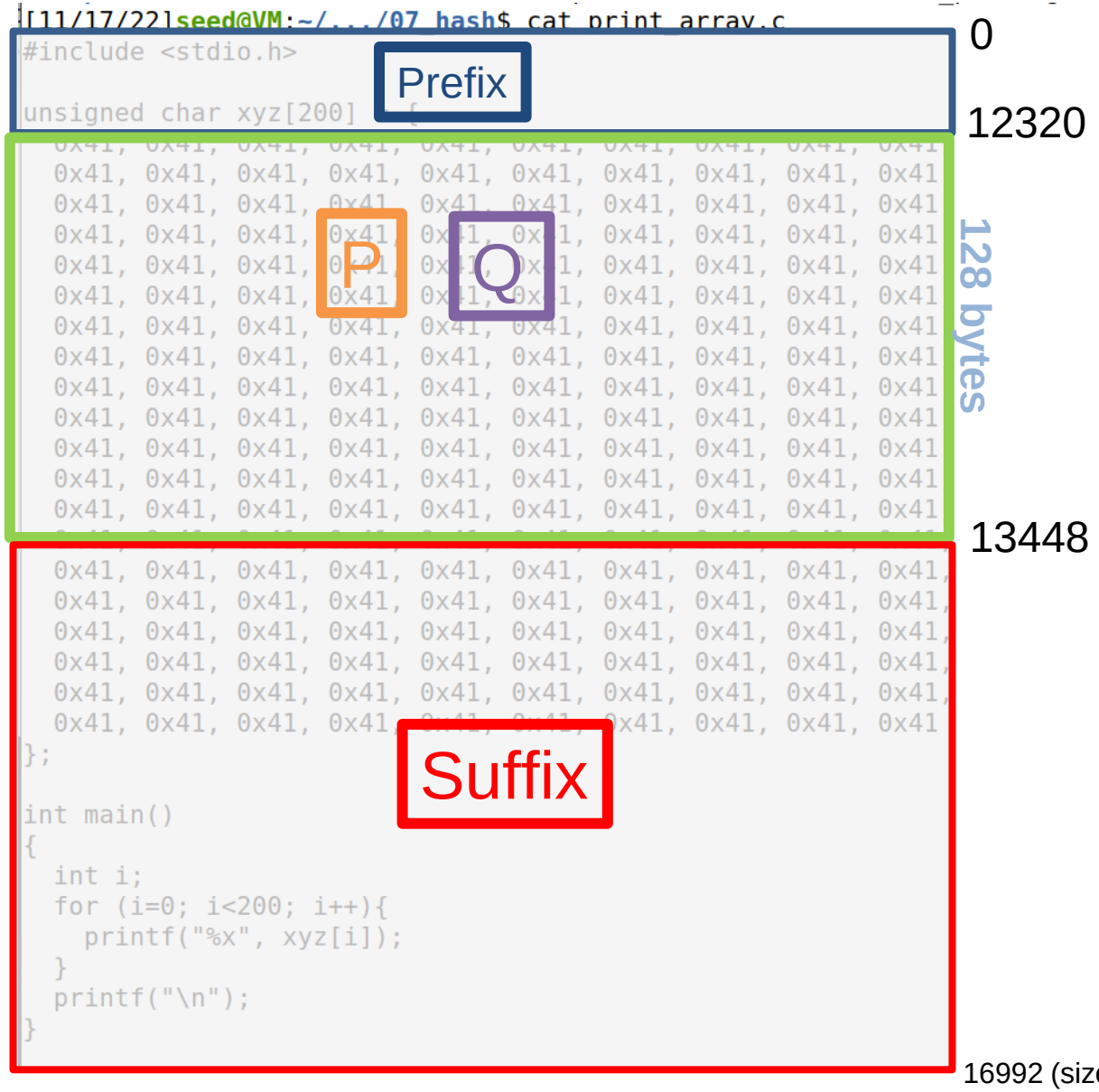
16992 (size of executable)

Make sure you still have a valid program ☺

```
[11/17/22]seed@VM:~/hash_lab$ ./program1.out
0000000000000000000000000000002515c9822b0c8e1c3265b6d48b082b19d4fae48dbe5b2412
773177388eb1e812afe86cc3f982b6f3d9ecda828b4bdcb17c9d974a22315b9db0e36b2165c6d3d4
598f447c72b594f618043ca1ede8b2546366ed15aee4bec86de6b70259e58f498e442da6e482fc64
c67acfbec66eab7dc38908fbc2d77dfe9841414141414141414141414141414141414141414141
4141414141414141414141414141414141
[11/17/22]seed@VM:~/hash_lab$ ./program2.out
0000000000000000000000000000002515c9822b0c8e1c3265b6d48b082b19d4f2e48dbe5b2412
773177388eb1e812afe86cc3f982b6f3d9ec8da828b4bdcb17c9d974a22315b1db0e36b2165c6d3d
4598f447c72b594f618043ca1edeb2546366ed15aee4bec86de6b70259e58f498e442da6e482fc6c
c66acfbec66eab7dc38908fbcad77dfe9841414141414141414141414141414141414141414141
4141414141414141414141414141414141
[11/17/22]seed@VM:~/hash_lab$
```

Somewhere in this output, you should find a small difference

Task 4 on the lab



0

12320

128 bytes

13448

16992 (size of executable)

Make sure you still have a valid program ☺

```
[11/17/22]seed@VM:~/hash_lab$ ./program1.out
000000000000000000000000000000002515c9822b0c8e1c3265b6d48b082b19d4fae48dbe5b2412
773177388eb1e812afe86cc3f982b6f3d9ecda828b4bdcb17c9d974a22315b9db0e36b2165c6d3d4
598f447c72b594f618043ca1ede8b2546366ed15aee4bec86de6b70259e58f498e442da6e482fc64
c67acfbec66eab7dc38908fbc2d77dfe984141414141414141414141414141414141414141414141
414141414141414141414141414141414141
[11/17/22]seed@VM:~/hash_lab$ ./program2.out
000000000000000000000000000000002515c9822b0c8e1c3265b6d48b082b19d4f2e48dbe5b2412
773177388eb1e812afe86cc3f982b6f3d9ec8da828b4bdcb17c9d974a22315b1db0e36b2165c6d3d
4598f447c72b594f618043ca1edeb2546366ed15aee4bec86de6b70259e58f498e442da6e482fc6c
c66acfbec66eab7dc38908fbcad77dfe984141414141414141414141414141414141414141414141
414141414141414141414141414141414141
[11/17/22]seed@VM:~/hash_lab$
```

These programs print out different things, which is very benign

Our next goal is to write two programs with the same MD5 hash, but one does something malicious, and the other does something benign

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

This program has two arrays X and Y

The program compares the contents of these two arrays

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

This program has two arrays X and Y

The program compares the contents of these two arrays

If the two arrays are the same, then it will execute the benign code

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

This program has two arrays X and Y

The program compares the contents of these two arrays

If the two arrays are the same, then it will execute the benign code

If the two arrays are different, then it will execute the benign code

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```


```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```


Goal: Generate two variants of this program. One variant where the two arrays are the same (benign version), and one variant where the two arrays are different (malicious version)

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

X array: **P**; Y array: **Q**

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

X array: **P**; Y array: **P**

Goal: Generate two variants of this program. One variant where the two arrays are the same (benign version), and one variant where the two arrays are different (malicious version)

Because we can overwrite Array data without crashing the program, we will once again place **P** and **Q** (from md5collgen) into our arrays

Prefix

P

Q

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

Prefix

P

P

```c
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

unsigned char Y[LENGTH]= {
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

Prefix

```
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
```

P

```
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

suffix1

```
};

unsigned char Y[LENGTH]= {
```

Q

```
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

```
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

Prefix

```
#include <stdio.h>
#define LENGTH 400

unsigned char X[LENGTH]= {
```

P

```
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

suffix1

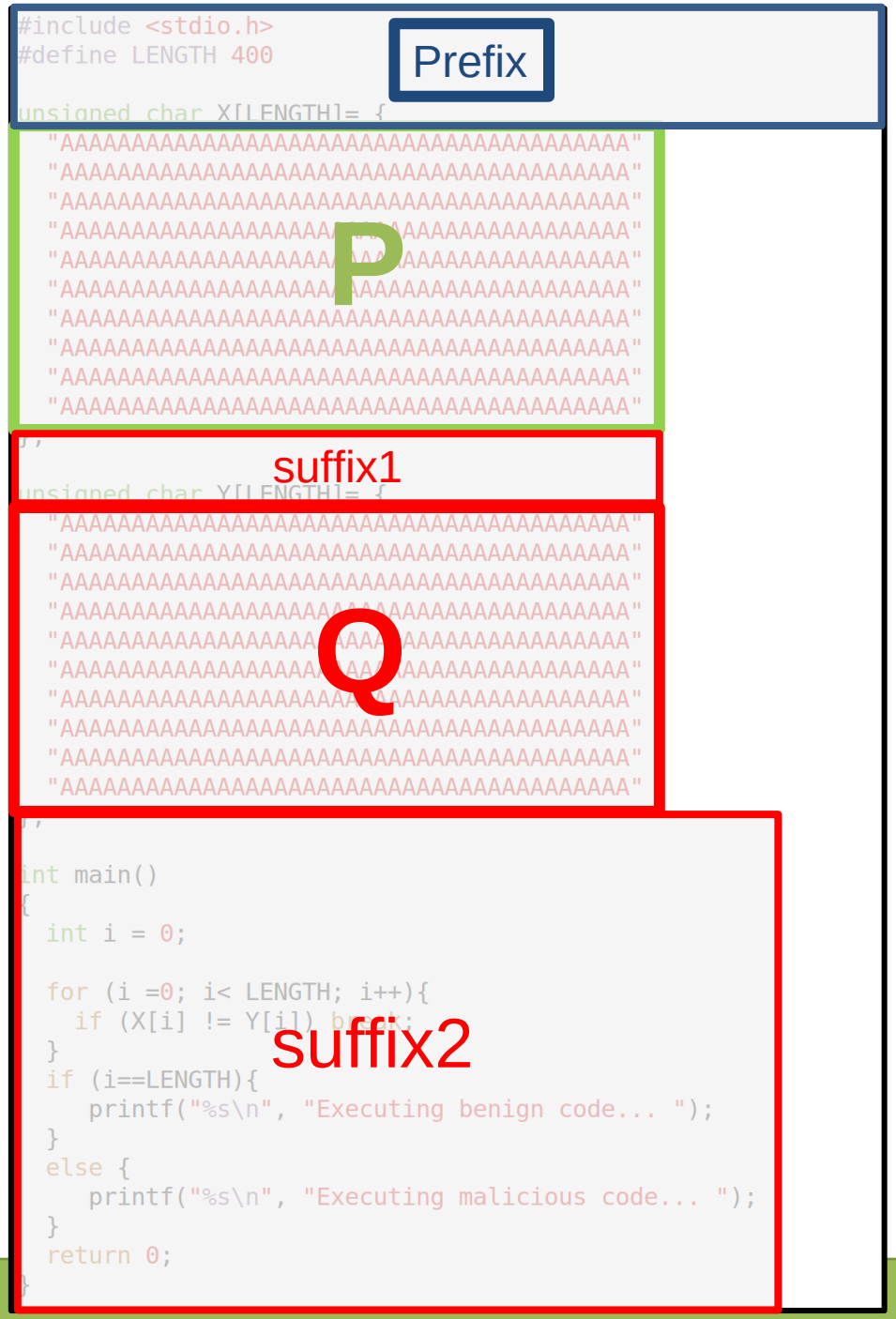
```
};

unsigned char Y[LENGTH]= {
```

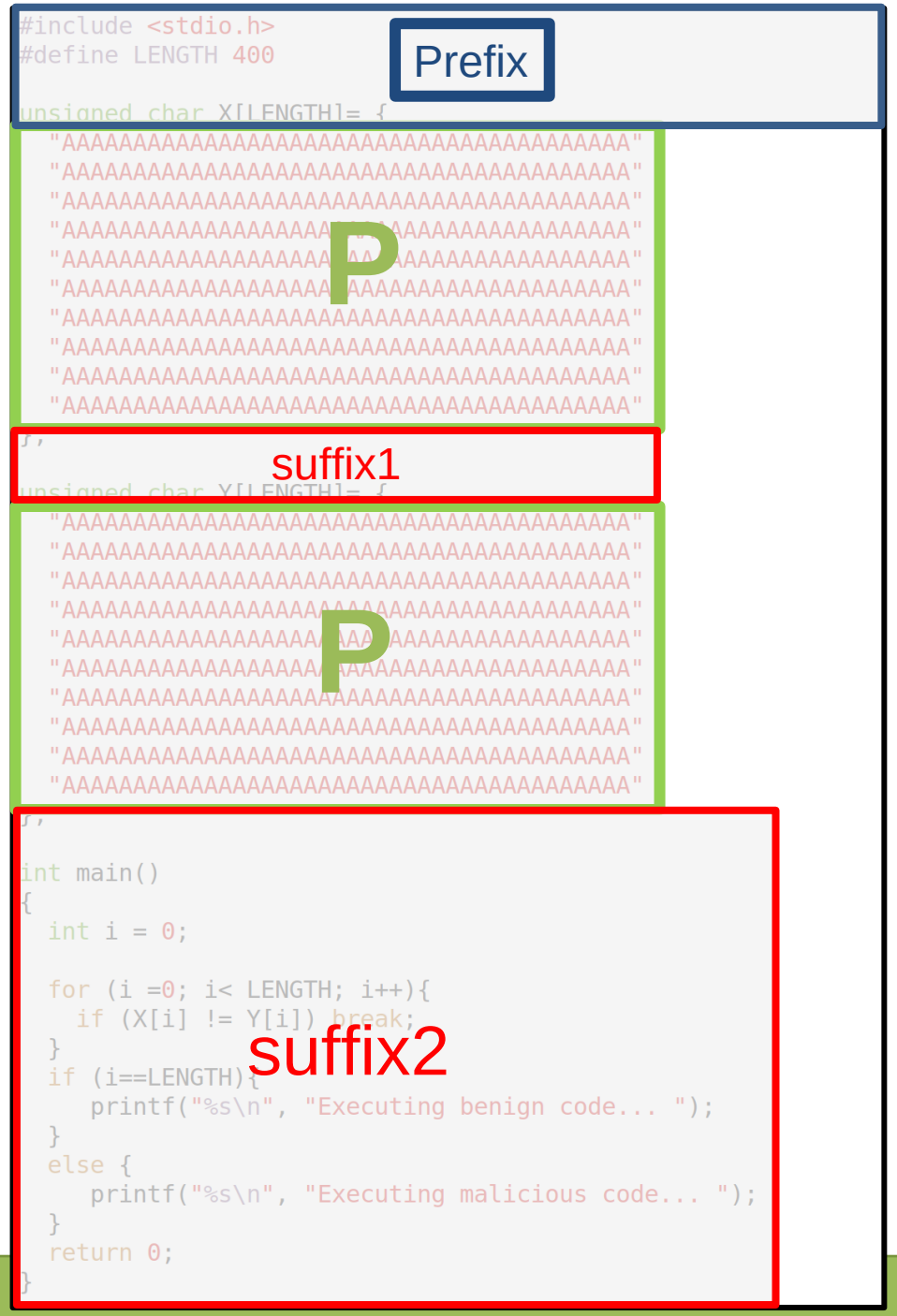
P

```
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
```

```
};

int main()
{
  int i = 0;

  for (i =0; i< LENGTH; i++){
    if (X[i] != Y[i]) break;
  }
  if (i==LENGTH){
     printf("%s\n", "Executing benign code... ");
  }
  else {
     printf("%s\n", "Executing malicious code... ");
  }
  return 0;
}
```

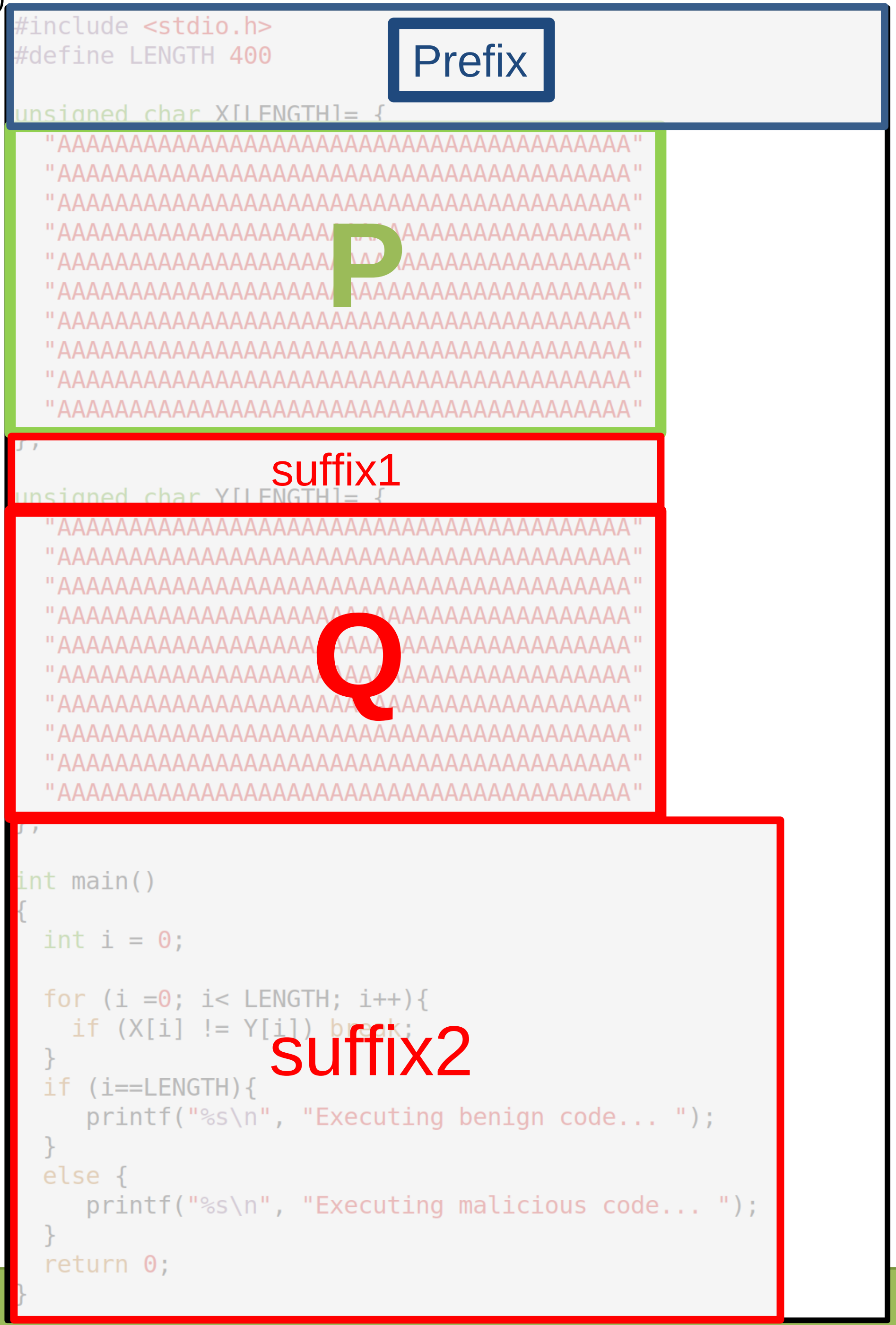
Because we are inserting P/Q at two different points into our program, we will have two suffixes



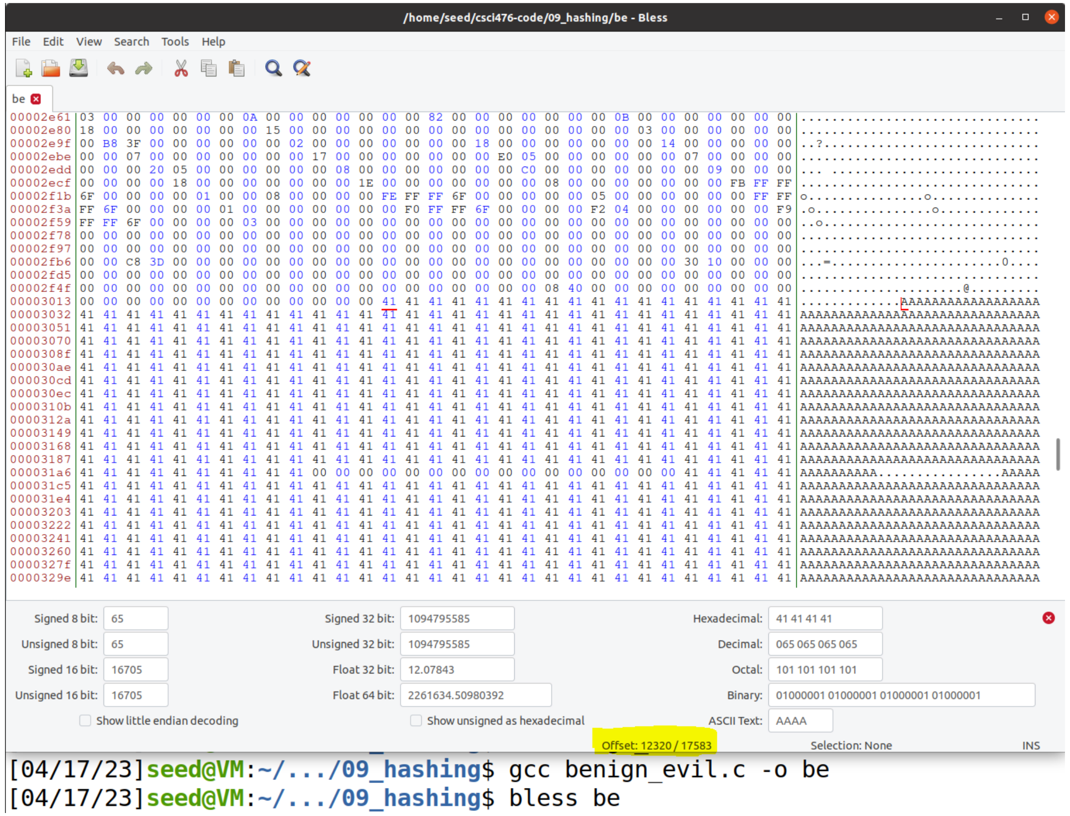


Because we are inserting P/Q at two different points into our program, we will have two suffixes

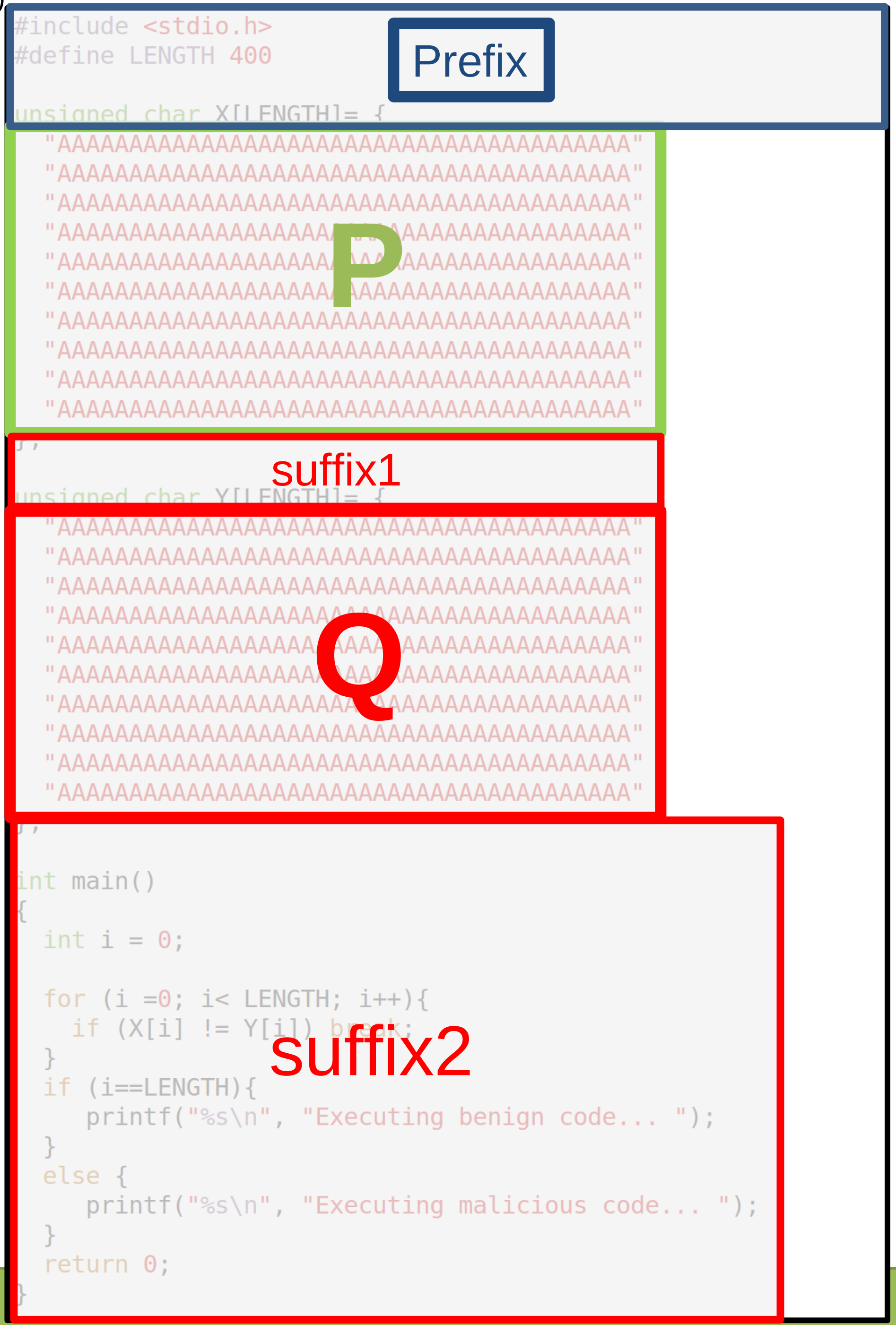
Memory Locations (Bytes)



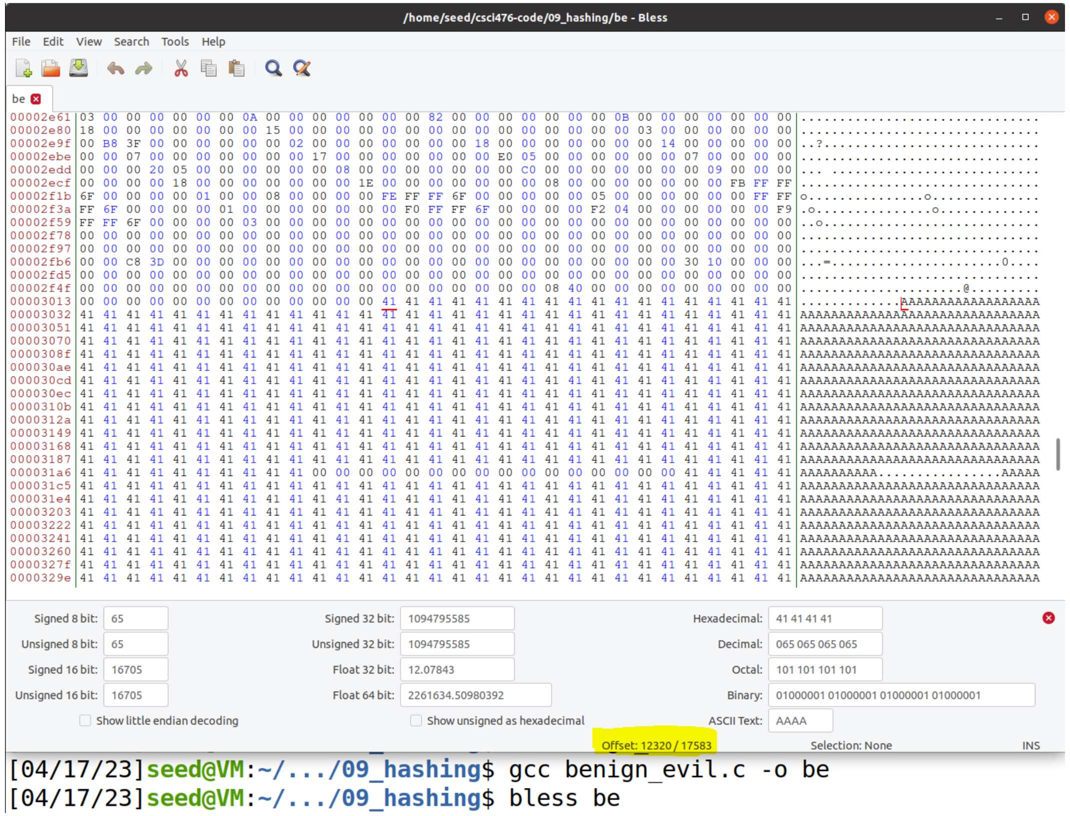
First, we must get the starting location of our array X



Memory Locations (Bytes)



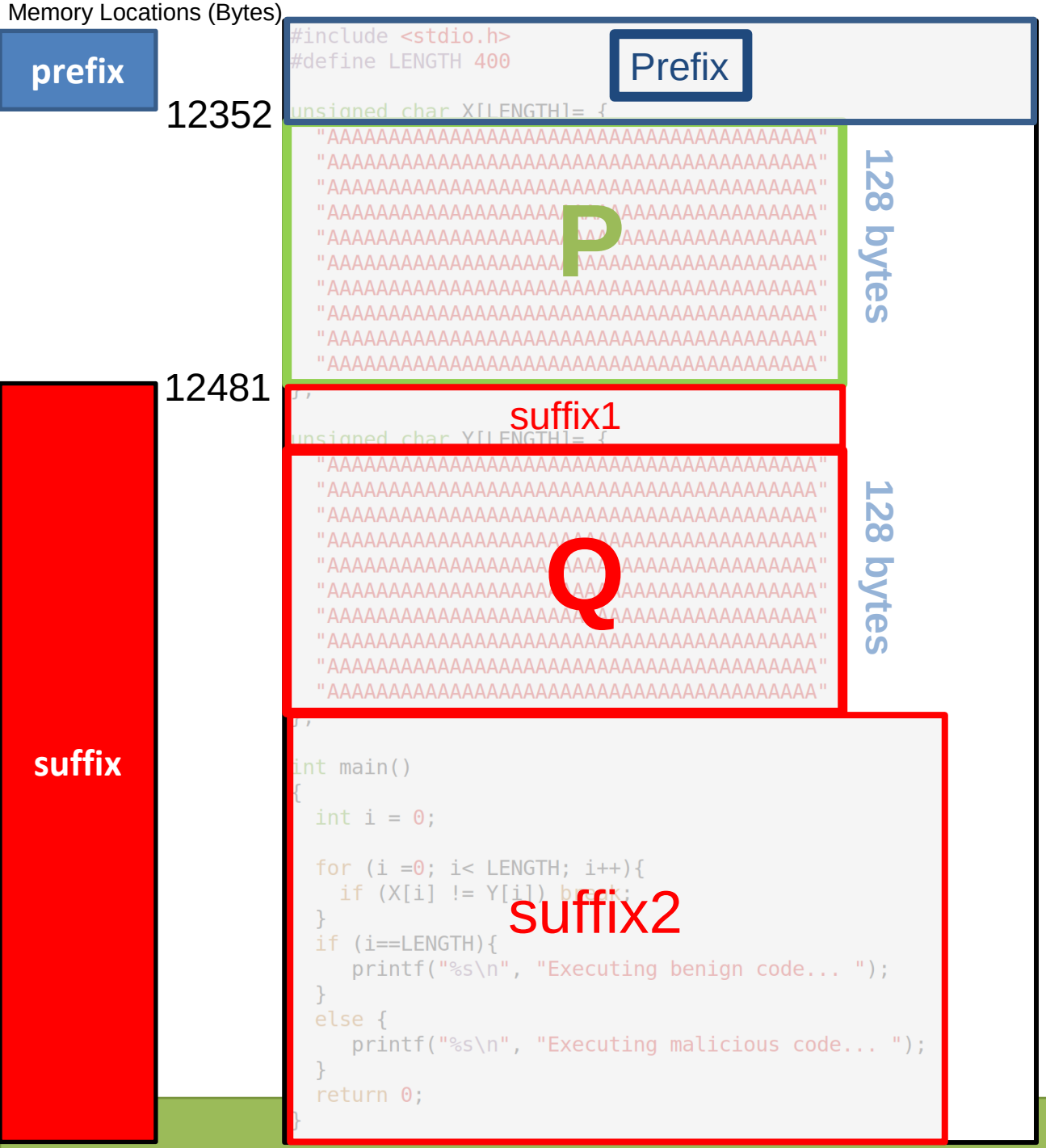
First, we must get the starting location of our array X



Offset = 12320

Padding will mess up out attack, so we must make sure that padding doesn't get added

Offset = 12352 (multiple of 64)



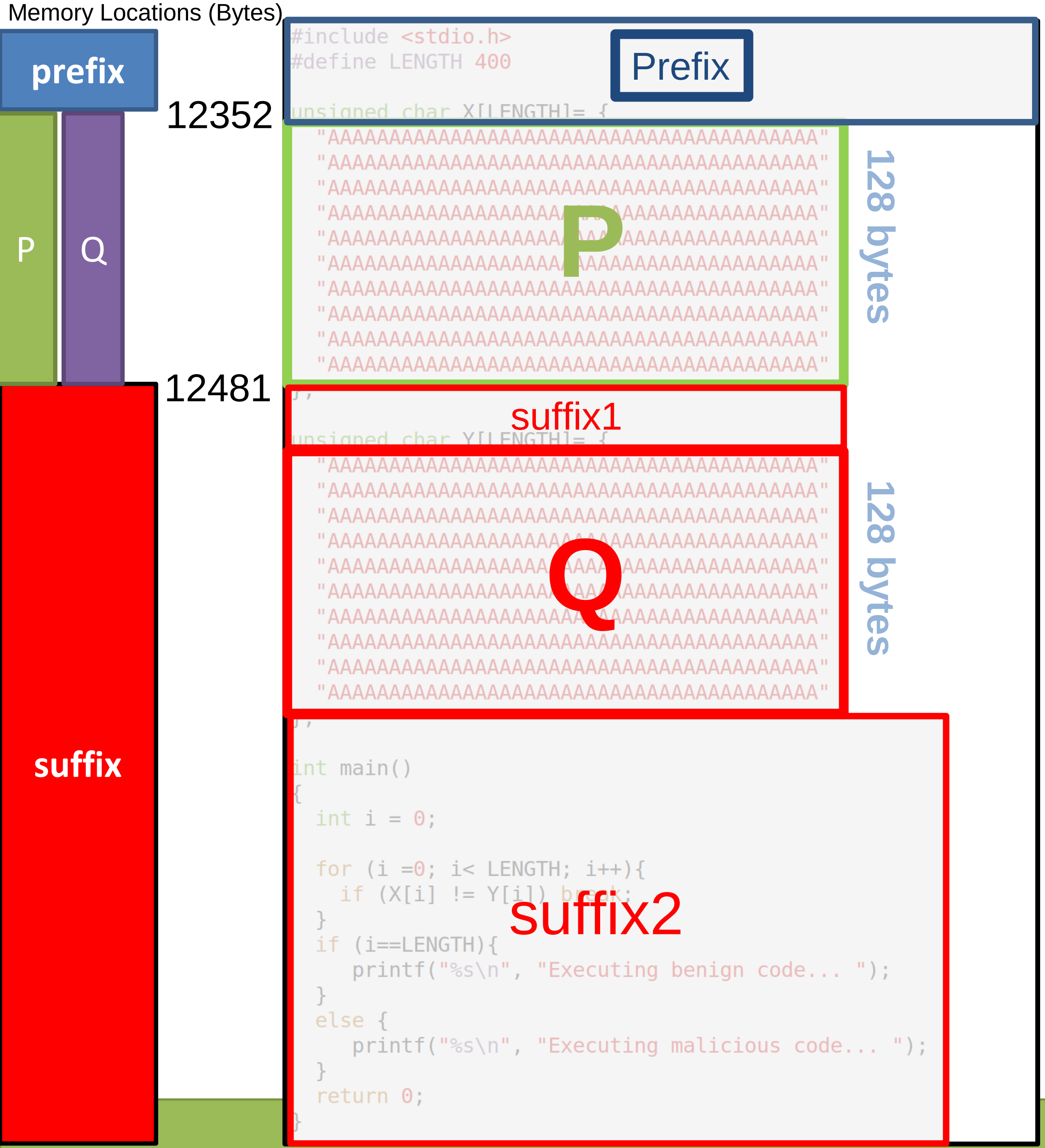
Offset = 12352

P and Q will once again be 128 bytes long

Therefore, P ends at (12352 + 128) = 12480

(we have to add +1 when getting the suffix to prevent getting an extra byte)

```
[04/17/23]seed@VM:~/.../09_hashing$ head -c 12352 be > prefix
[04/17/23]seed@VM:~/.../09_hashing$ tail -c +12481 be > suffix
```

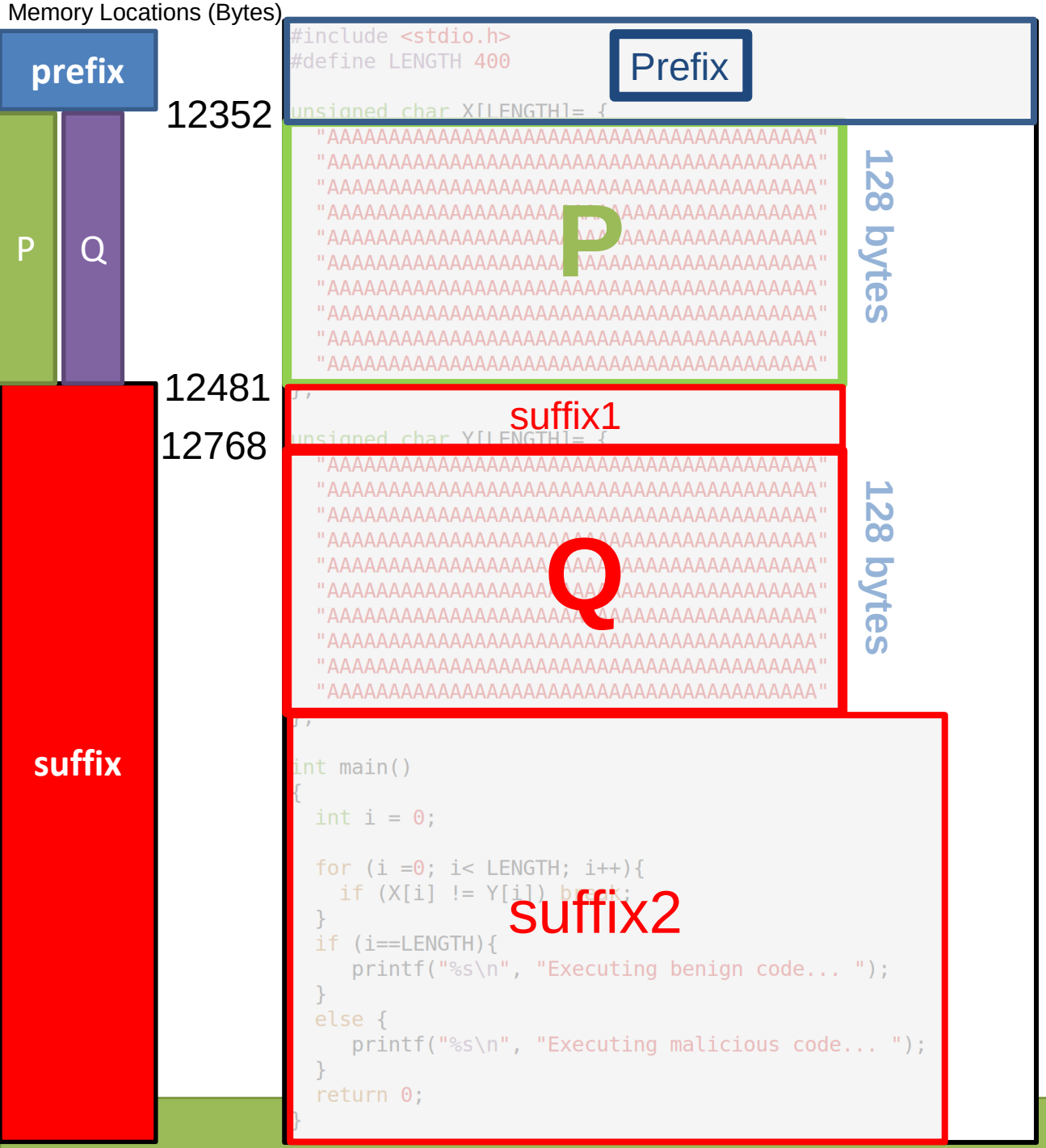


Now that we have the prefix, we can generate P and Q

```
[04/17/23]seed@VM:~/.../09_hashing$ md5collgen -p prefix -o out1 out2
MD5 collision generator v1.5
by Marc Stevens (http://www.win.tue.nl/hashclash/)

Using output filenames: 'out1' and 'out2'
Using prefixfile: 'prefix'
Using initial value: fe5a2a34aa864ea9110d0e2fa43e0327

Generating first block: ........
Generating second block: S10.........................................
Running time: 4.60817 s
```

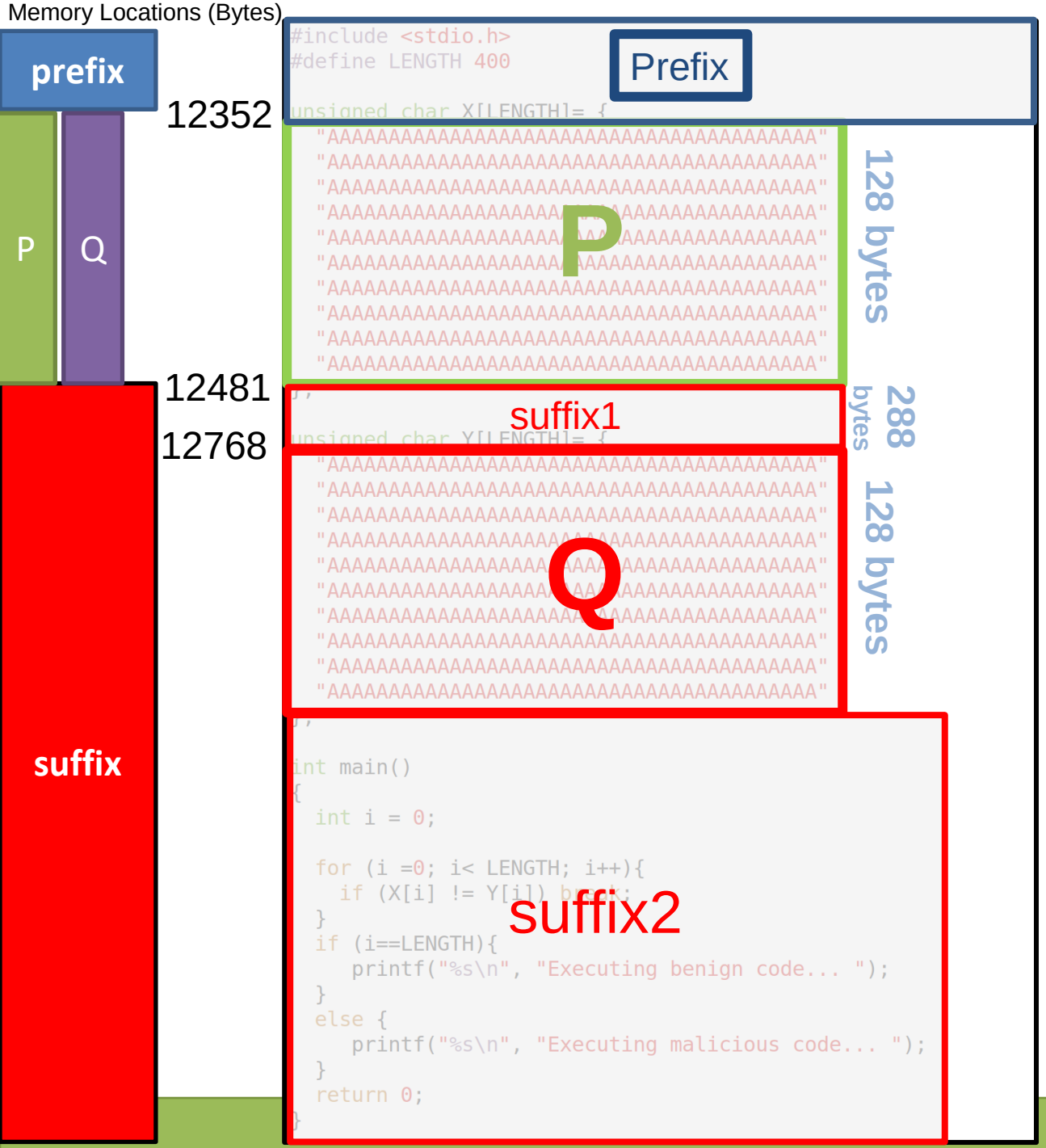
Because we just want P and Q (not the prefix), we will take the final 128 bytes of the output of `out1` and `out2`

```
[04/17/23]seed@VM:~/.../09_hashing$ tail -c 128 out1 > P
[04/17/23]seed@VM:~/.../09_hashing$ tail -c 128 out2 > Q
```



To avoid padding, we had to move the beginning of P up 32 bytes into the array X, so when we insert P/Q into array Y, we need to make sure we also do 32 bytes

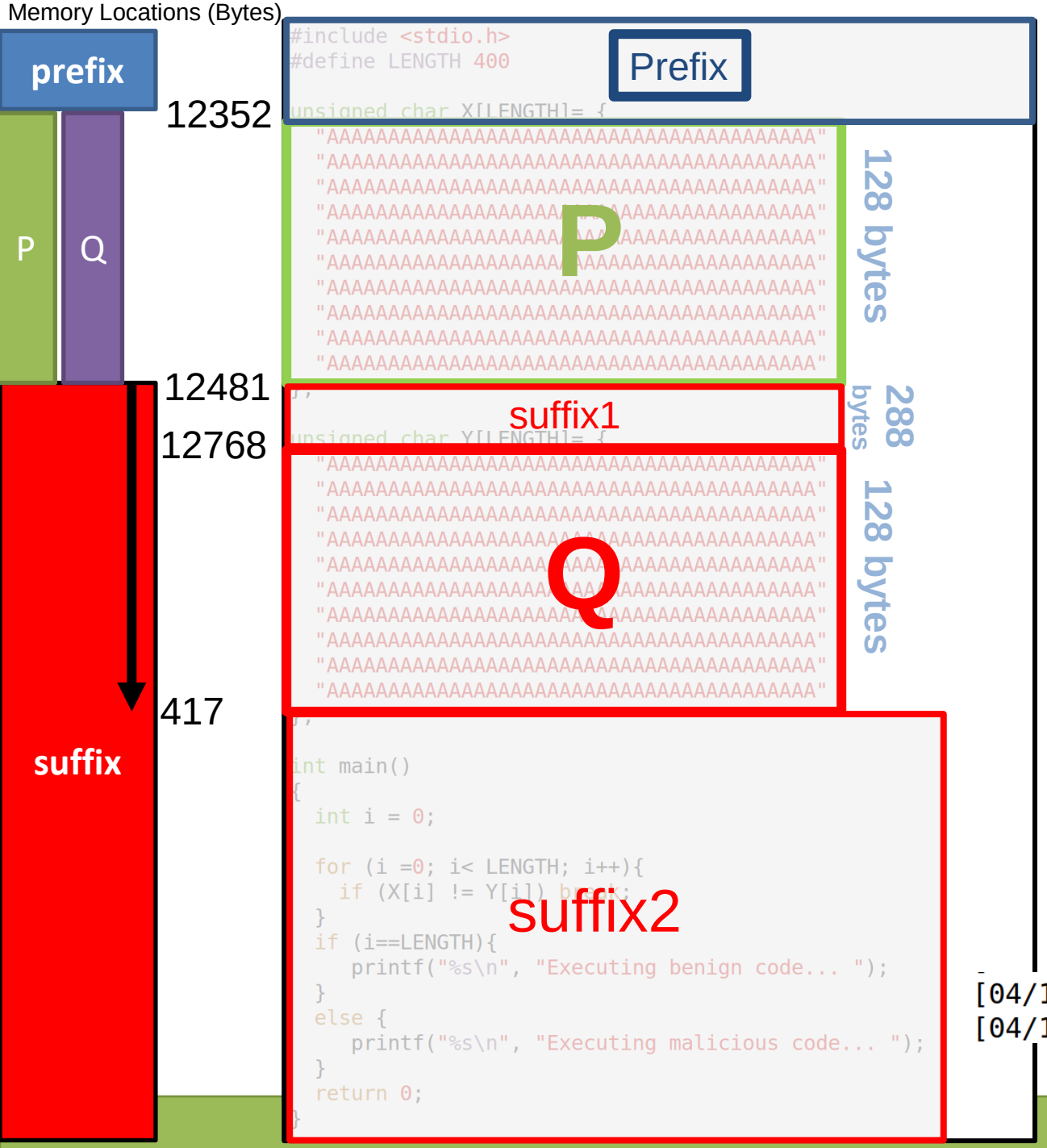
Array Y starts at 12736, but we need to inject at byte (12736 + 32) = **12768**



To avoid padding, we had to move the beginning of P up 32 bytes into the array X, so when we insert P/Q into array Y, we need to make sure we also do 32 bytes

Array Y starts at 12736, but we need to inject at byte (12736 + 32) = **12768**

Therefore, the size of suffix1 will be 12768 – 12481 = **288**



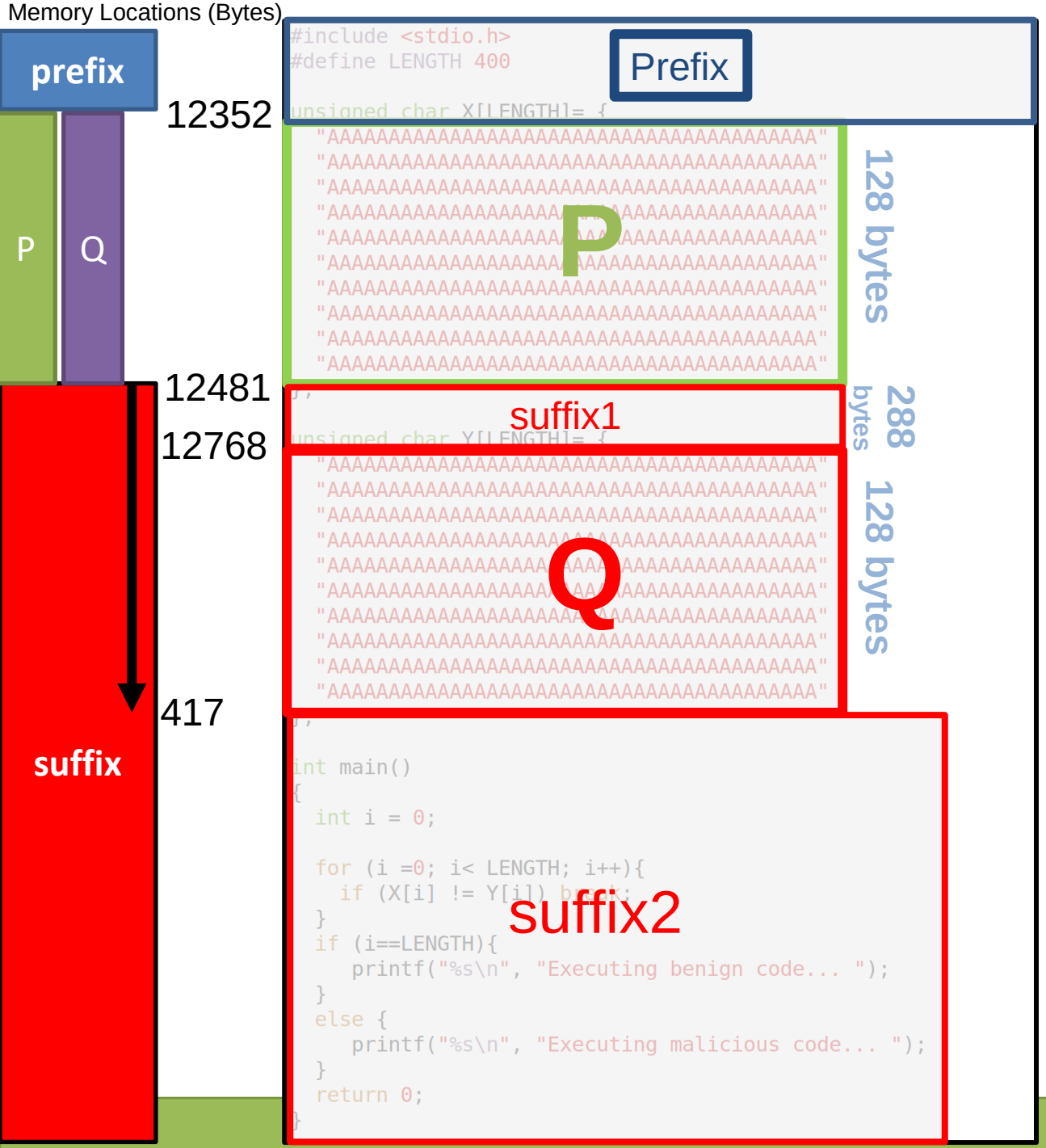
To avoid padding, we had to move the beginning of P up 32 bytes into the array X, so when we insert P/Q into array Y, we need to make sure we also do 32 bytes

Array Y starts at 12736, but we need to inject at byte (12736 + 28) = **12768**

Therefore, the size of suffix1 will be 12768 – 12481 = **288**

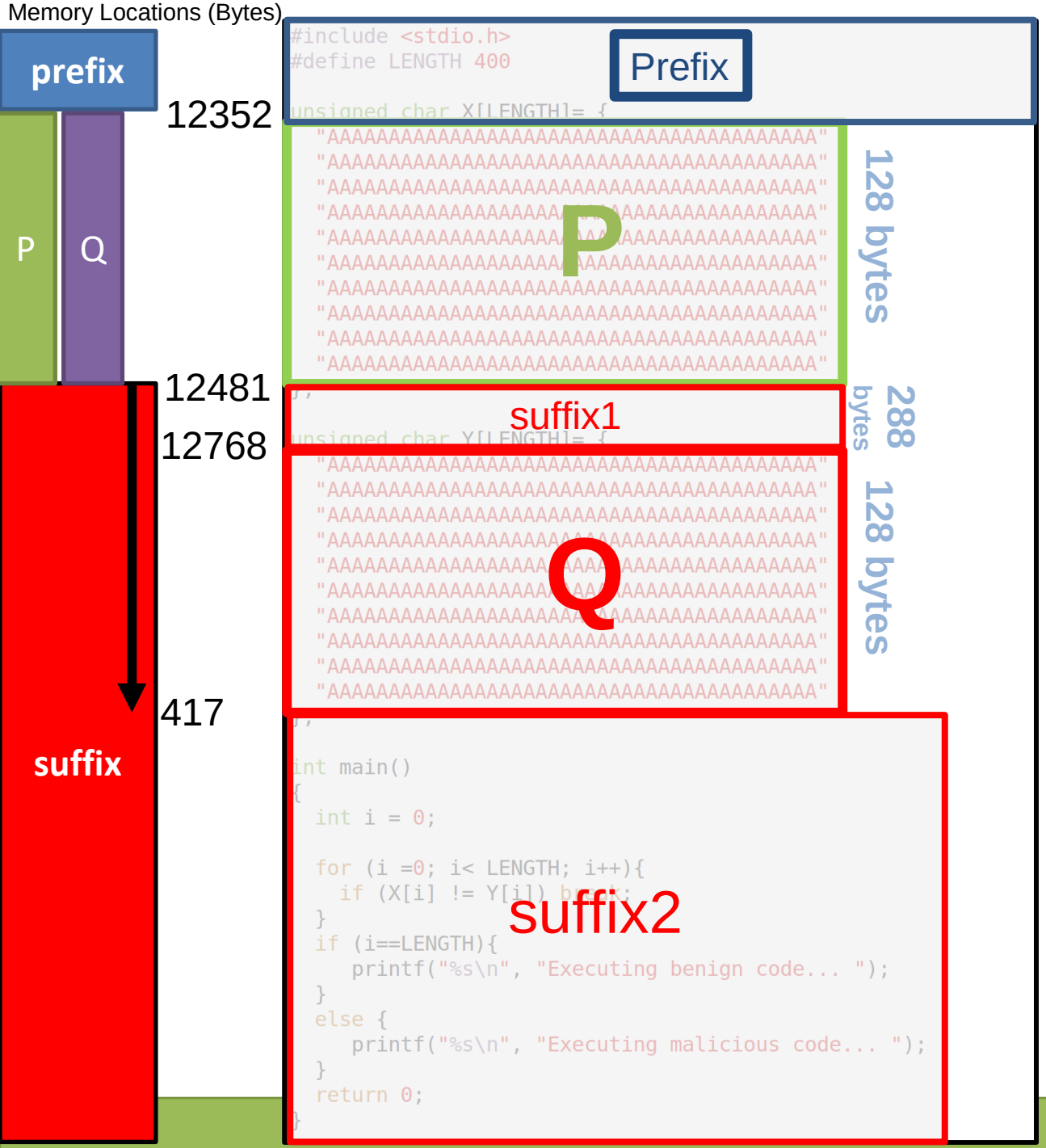
Suffix 2 will begin at byte 288 + 128 = 416 of **suffix** (but we add +1 to prevent getting an extra byte)

```
[04/17/23]seed@VM:~/.../09_hashing$ head -c 288 suffix > suffix1
[04/18/23]seed@VM:~/.../09_hashing$ head -c +417 suffx > suffix2
```



Now, we put everything together

```
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix P suffix1 P suffix2 > final1
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix Q suffix1 P suffix2 > final2
```

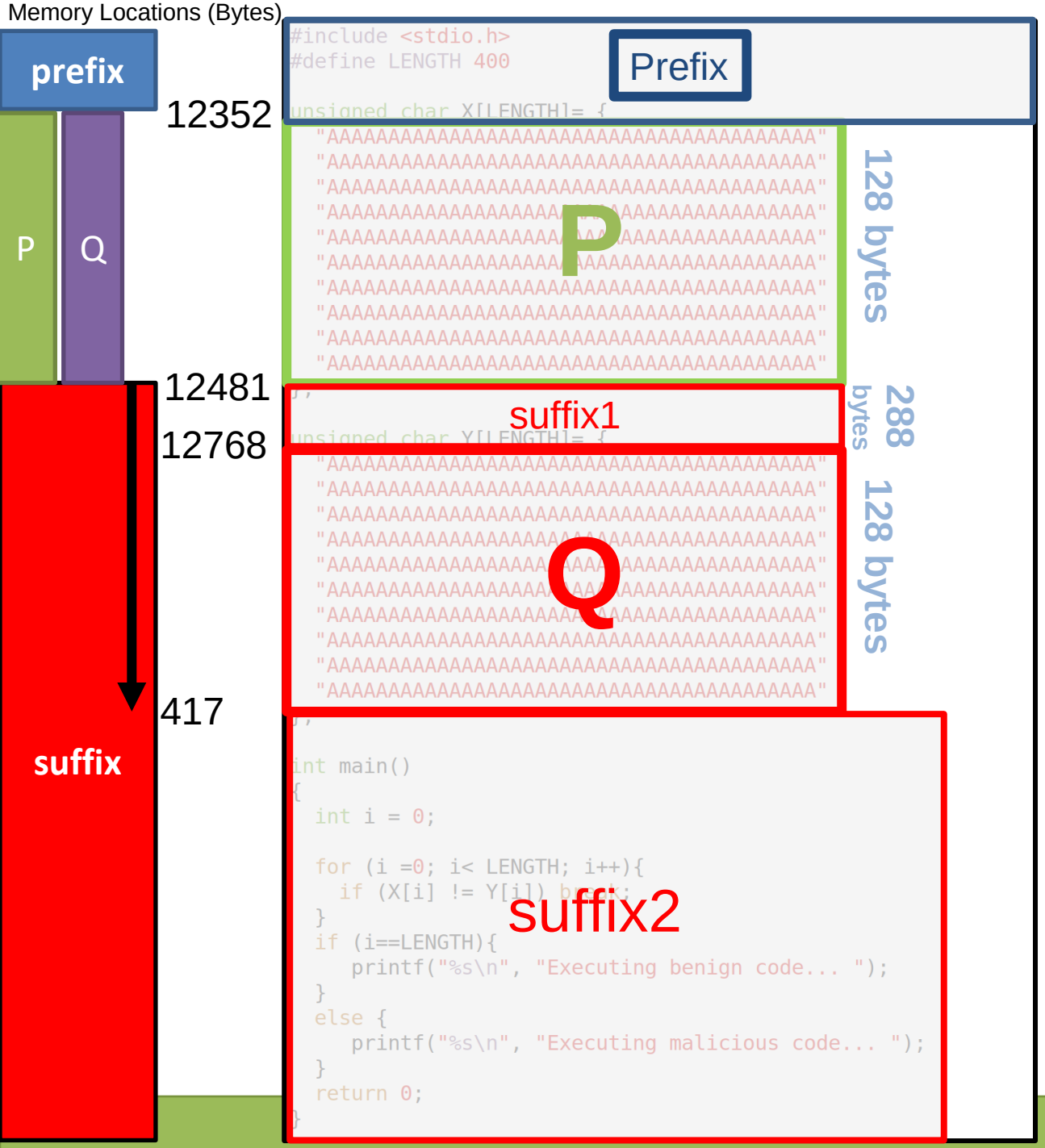


Now, we put everything together

```
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix P suffix1 P suffix2 > final1
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix Q suffix1 P suffix2 > final2
```

Verify that hashes match:

```
[04/18/23]seed@VM:~/.../09_hashing$ md5sum final1 final2
7eb3ea7eaefaa2efbd0ddfa0c7022e76  final1
7eb3ea7eaefaa2efbd0ddfa0c7022e76  final2
```



Now, we put everything together

```
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix P suffix1 P suffix2 > final1
[04/18/23]seed@VM:~/.../09_hashing$ cat prefix Q suffix1 P suffix2 > final2
```

Verify that hashes match:

```
[04/18/23]seed@VM:~/.../09_hashing$ md5sum final1 final2
7eb3ea7eaefaa2efbd0ddfa0c7022e76  final1
7eb3ea7eaefaa2efbd0ddfa0c7022e76  final2
```

Make programs executable and run them!

```
[04/18/23]seed@VM:~/.../09_hashing$ chmod u+x final1 final2
[04/18/23]seed@VM:~/.../09_hashing$ ./final1
Executing benign code...
[04/18/23]seed@VM:~/.../09_hashing$ ./final2
Executing malicious code...
```