C++ Code Analysis

The main parts of this code are **toggle**() and **main_prog**(), which is the child process of **main**(). There are three global variables: mem_size, toggles, and *g_mem. The function **pick_addr**() is used to randomly select a memory address within the allocated memory range mem_size. The Timer class is used for measuring time durations of specific parts of code (similar to ‘struct’ in C and ‘class’ in Java with a bit of difference).

Now I will focus on the **toggle**() and **main_prog**() functions.

1. main_prog()

First, we use the mmap system call to allocate a continuous memory block, which is similar to C's dynamic memory allocation using malloc:

```
g_mem = (char *)malloc(mem_size)
```

‘g_mem’ is the pointer to the starting address of the allocated memory.

0xFF in hexadecimal represents a byte with all bits set to 1, which is 1111 1111 (8 bits) in binary.

We choose ‘char’ here because the char type is typically composed of 8 bits, which is equivalent to one byte.

After the dynamic memory allocation and initialization, we get the memory layout diagram of a 8 columns * 1 << 30 matrix(Image F_1).

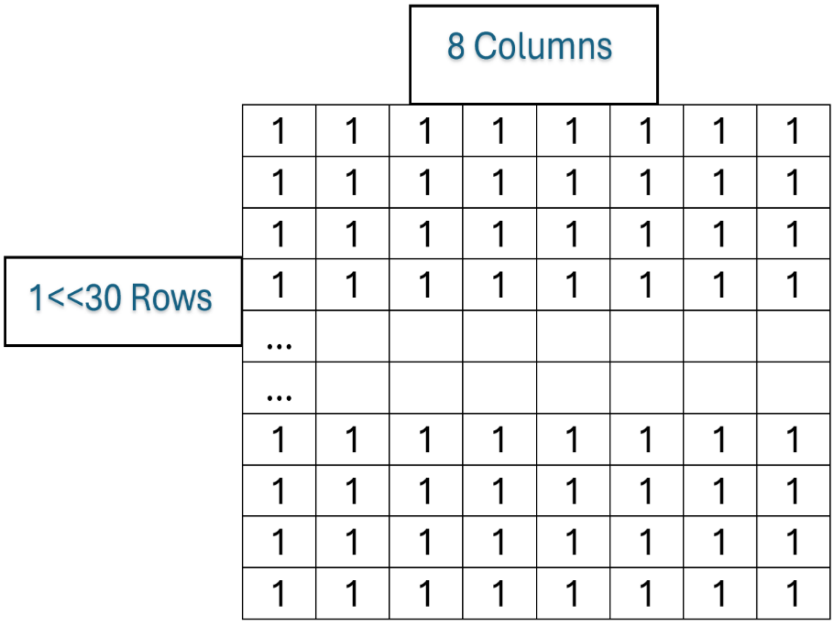
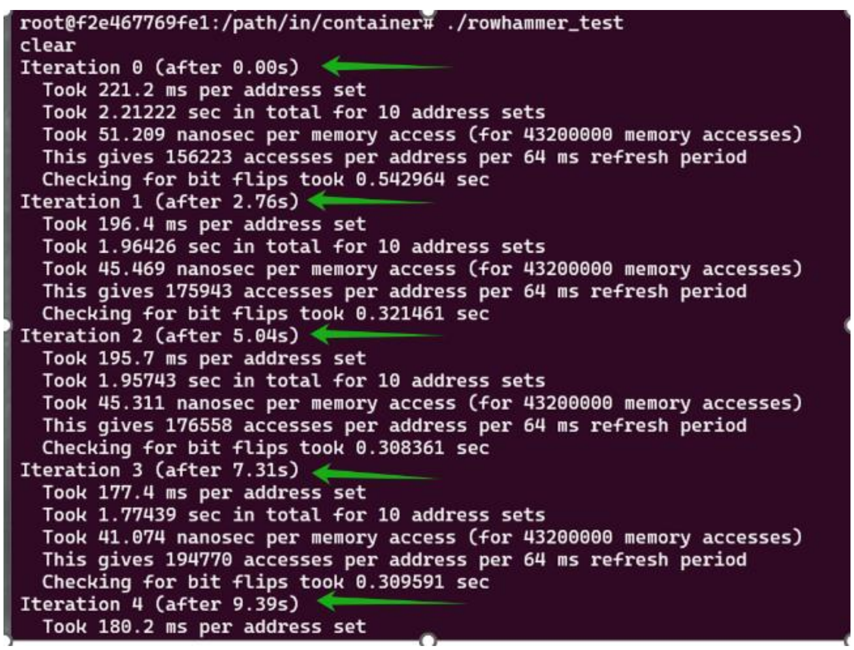


Image F_1

Then the code creates a ‘Timer’ object named ‘t’ and an integer variable ‘iter’ as the loop marker initialized to 0. Within each iteration of the infinite loop, it prints the current iteration number and the time elapsed using **Printf(),**as we can see in image F_2.



```
root@f2e467769fe1:/path/in/container# ./rowhammer_test
clear
Iteration 0 (after 0.00s)
  Took 221.2 ms per address set
  Took 2.21222 sec in total for 10 address sets
  Took 51.209 nanosec per memory access (for 43200000 memory accesses)
  This gives 156223 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.542964 sec
Iteration 1 (after 2.76s)
  Took 196.4 ms per address set
  Took 1.96426 sec in total for 10 address sets
  Took 45.469 nanosec per memory access (for 43200000 memory accesses)
  This gives 175943 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.321461 sec
Iteration 2 (after 5.04s)
  Took 195.7 ms per address set
  Took 1.95743 sec in total for 10 address sets
  Took 45.311 nanosec per memory access (for 43200000 memory accesses)
  This gives 176558 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.308361 sec
Iteration 3 (after 7.31s)
  Took 177.4 ms per address set
  Took 1.77439 sec in total for 10 address sets
  Took 41.074 nanosec per memory access (for 43200000 memory accesses)
  This gives 194770 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.309591 sec
Iteration 4 (after 9.39s)
  Took 180.2 ms per address set
```

Image F_2

Now, for **toggle**(10, 8), it will call another core function here. So let us temporarily leave **main_prog**()for **toggle**().

2. toggle(10, 8):

Firstly, we create another object of Timer named timer.

For each loop, the line **uint32_t *addrs[addr_count]** declares an array of pointers to uint32_t (unsigned 32-bit integer) values, which is 4 Bytes. Each element of the array is a pointer to a uint32_t value. To confirm this assumption, I added a print test code:

```
char *pick_addr() {
  size_t offset = (rand() << 12) % mem_size;
  printf("INSIDE pick_addr(): g_mem + offset=%p\n", (void*)(g_mem + offset));;
  return g_mem + offset;
}
```

Image F_3

The output below shows the assumption is correct.

```
INSIDE pick_addr(): g_mem + offset=0x7f50c215c000
INSIDE pick_addr(): g_mem + offset=0x7f50afc6e000
INSIDE pick_addr(): g_mem + offset=0x7f50cb7e3000
  Took 207.2 ms per address set
  Took 2.07172 sec in total for 10 address sets
  Took 47.956 nanosec per memory access (for 43200000 memory accesses)
  This gives 166817 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.107451 sec
Iteration 7 (after 15.66s)
INSIDE pick_addr(): g_mem + offset=0x7f50bbf05000
INSIDE pick_addr(): g_mem + offset=0x7f509e939000
INSIDE pick_addr(): g_mem + offset=0x7f509007d000
INSIDE pick_addr(): g_mem + offset=0x7f50b31b7000
INSIDE pick_addr(): g_mem + offset=0x7f50cadfb000
INSIDE pick_addr(): g_mem + offset=0x7f50ba1b4000
INSIDE pick_addr(): g_mem + offset=0x7f50a50e1000
INSIDE pick_addr(): g_mem + offset=0x7f50bd7c7000
INSIDE pick_addr(): g_mem + offset=0x7f50b29bc000
INSIDE pick_addr(): g_mem + offset=0x7f50a8bb0000
INSIDE pick_addr(): g_mem + offset=0x7f5093a28000
INSIDE pick_addr(): g_mem + offset=0x7f50a5986000
INSIDE pick_addr(): g_mem + offset=0x7f5099d74000
INSIDE pick_addr(): g_mem + offset=0x7f50ca5a9000
INSIDE pick_addr(): g_mem + offset=0x7f50b37e4000
INSIDE pick_addr(): g_mem + offset=0x7f50afde2000
INSIDE pick_addr(): g_mem + offset=0x7f50ccaac000
INSIDE pick_addr(): g_mem + offset=0x7f50aa60d000
INSIDE pick_addr(): g_mem + offset=0x7f509c0bc000
INSIDE pick_addr(): g_mem + offset=0x7f50bb9f8000
INSIDE pick_addr(): g_mem + offset=0x7f50bc6e4000
INSIDE pick_addr(): g_mem + offset=0x7f5090026000
```

There are 8 * 10 = 80 rows in each Iteration,which means 80 addresses.

Image F_4

Let us continue to execute our program:

```
for (int a = 0; a < addr_count; a++)
    addrs[a] = (uint32_t *) pick_addr();
```

Image F_5

After running the above snippet, we will obtain a memory layout diagram:

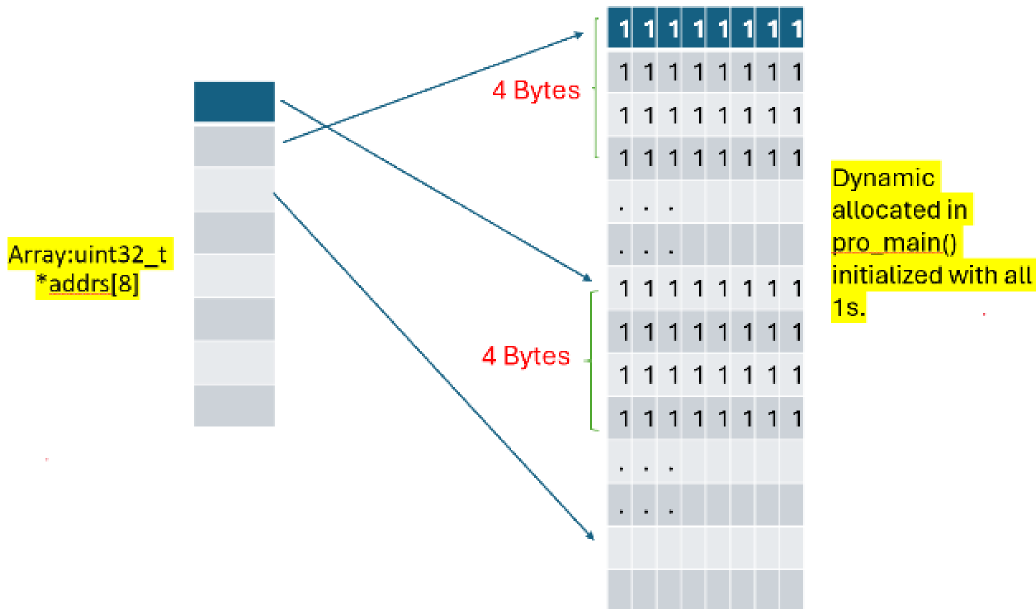


Image F_6

Let us move on:

```
uint32_t sum = 0;
for (int i = 0; i < toggles; i++) {
  for (int a = 0; a < addr_count; a++){
    sum += *addrs[a] + 1;
    printf("INSIDE LOOP: *addrs[a]=%x\n", *addrs[a]);  //This is for testing!
  }
  for (int a = 0; a < addr_count; a++)
    asm volatile("clflush (%0)" : : "r" (addrs[a]) : "memory");
}
```

```
printf("INSIDE LOOP: *addrs[a]=%x\n", *addrs[a]);  //This is for testing!
```

Image F_7

I added this **printf**() line to confirm whether they are the values that we are expecting, and I got the output below:

```
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
INSIDE LOOP: *addrs[a]=ffffffff
```

Image F_8

"ffffffff" in hexadecimal (or "1111 1111 1111 1111 1111 1111 1111 1111" in binary) represents 4 bytes filled with all 1s. Therefore, when adding 1 to "ffffffff", it will overflow, resulting in 0. Consequently, the sum should always be equal to **0**, unless something has gone wrong (such as a Rohammer event occurring).

The remaining part of this function comprises **printf** statements, and we can observe their outputs in Image F_2.

Now, let's return to **main_prog**().

3.main_prog()

Let us continue to execute the rest statements in **main_prog**().

```
toggle(10, 8);

Timer check_timer;
uint64_t *end = (uint64_t *) (g_mem + mem_size);
uint64_t *ptr;
int errors = 0;
for (ptr = (uint64_t *) g_mem; ptr < end; ptr++) {
  uint64_t got = *ptr;
  if (got != ~(uint64_t) 0) {
    printf("error at %p: got 0x%" PRIx64 "\n", ptr, got);
    errors++;
  }
}
printf("  Checking for bit flips took %f sec\n", check_timer.get_diff());
if (errors)
```

This block is used to check for bit flips. It iterates over memory addresses within the allocated memory range (refer to Image F_1). If a memory address contains a value different from "0xffffffffffffffff" (where all bits are set to 1), an error message is printed, and the error count is incremented. Rohammer could lead to this situation when repeated access to memory can induce bit flips in neighboring memory locations.

In this case, **uint32_t *addrs[addr_count]**, where it points to 8 blocks (each 4 bytes or 32 bits in size) in the previously allocated memory, are being repeatedly accessed. This repeated access can result in bit flips in neighboring memory

4.Conclusion

The code involves initializing memory, performing memory operations(repeated access to some addresses), and checking for errors to detect Rohammer.