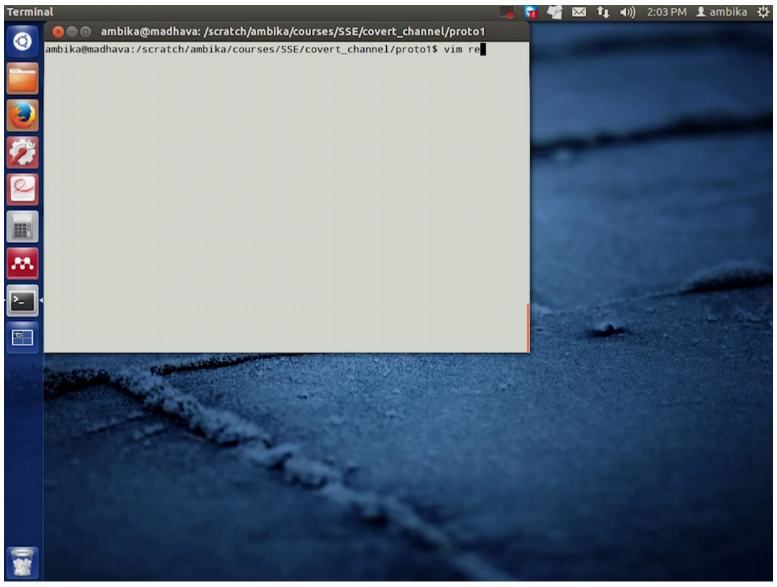
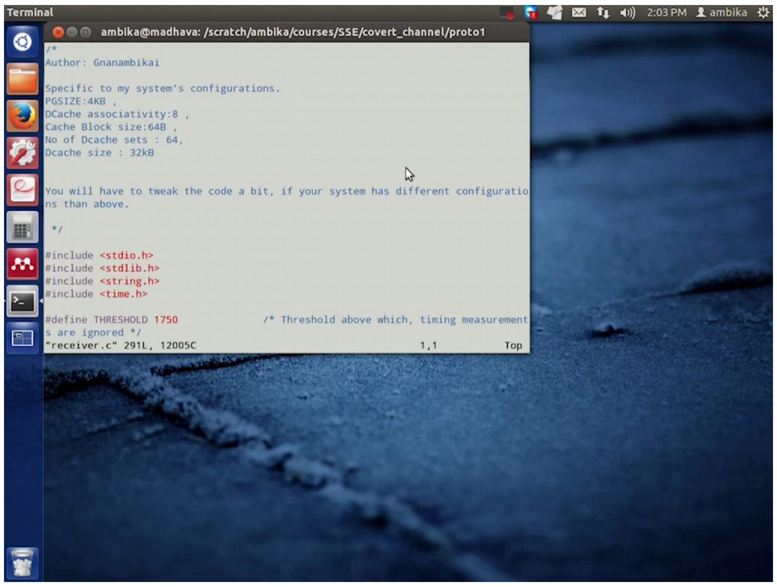
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
'Demo- Cache-timing based Covert Channel - Part 2'

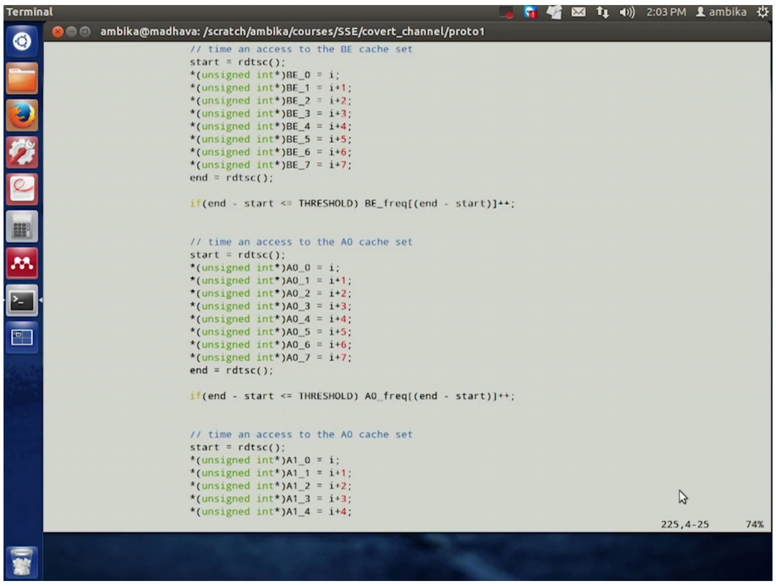
(Refer Slide Time: 0:16)


Terminal
2:03 PM
ambika
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ vim re
```

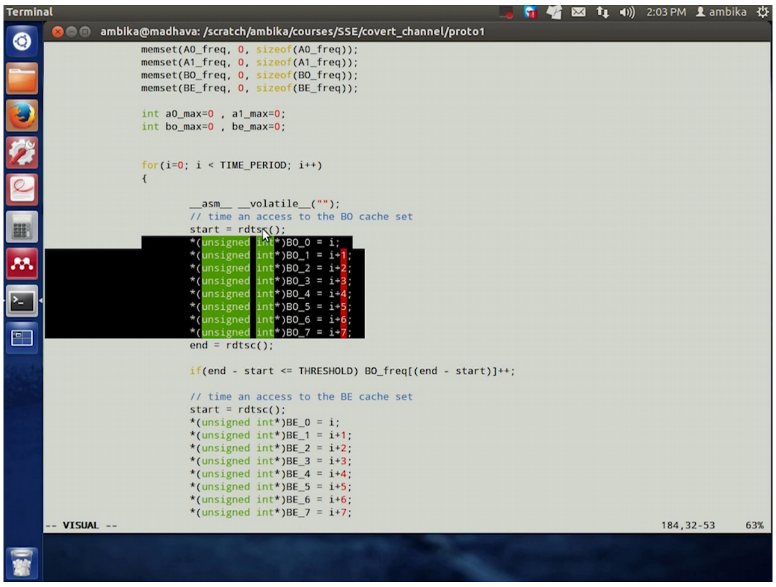

Terminal
2:03 PM
ambika
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```
/*
Author: Gnanambikai

Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB


You will have to tweak the code a bit, if your system has different configuratio
ns than above.

 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define THRESHOLD 1750              /* Threshold above which, timing measurement
s are ignored */
"receiver.c" 291L, 12005C                                     1,1           Top
```



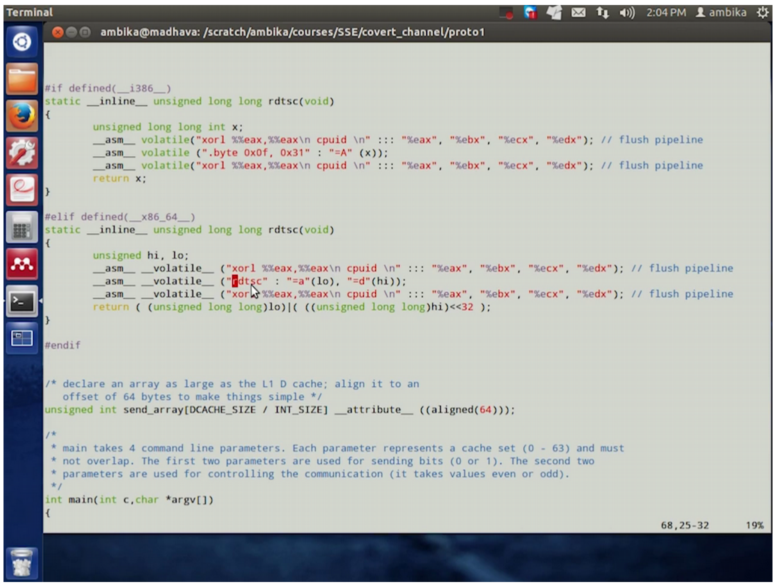
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```
                        // time an access to the BE cache set
                        start = rdtsc();
                        *(unsigned int*)BE_0 = i;
                        *(unsigned int*)BE_1 = i+1;
                        *(unsigned int*)BE_2 = i+2;
                        *(unsigned int*)BE_3 = i+3;
                        *(unsigned int*)BE_4 = i+4;
                        *(unsigned int*)BE_5 = i+5;
                        *(unsigned int*)BE_6 = i+6;
                        *(unsigned int*)BE_7 = i+7;
                        end = rdtsc();

                        if(end - start <= THRESHOLD) BE_freq[(end - start)]++;


                        // time an access to the A0 cache set
                        start = rdtsc();
                        *(unsigned int*)A0_0 = i;
                        *(unsigned int*)A0_1 = i+1;
                        *(unsigned int*)A0_2 = i+2;
                        *(unsigned int*)A0_3 = i+3;
                        *(unsigned int*)A0_4 = i+4;
                        *(unsigned int*)A0_5 = i+5;
                        *(unsigned int*)A0_6 = i+6;
                        *(unsigned int*)A0_7 = i+7;
                        end = rdtsc();

                        if(end - start <= THRESHOLD) A0_freq[(end - start)]++;


                        // time an access to the A0 cache set
                        start = rdtsc();
                        *(unsigned int*)A1_0 = i;
                        *(unsigned int*)A1_1 = i+1;
                        *(unsigned int*)A1_2 = i+2;
                        *(unsigned int*)A1_3 = i+3;
                        *(unsigned int*)A1_4 = i+4;
                                                                                          225,4-25      74%
```



ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```
                memset(A0_freq, 0, sizeof(A0_freq));
                memset(A1_freq, 0, sizeof(A1_freq));
                memset(BO_freq, 0, sizeof(BO_freq));
                memset(BE_freq, 0, sizeof(BE_freq));

                int a0_max=0 , a1_max=0;
                int bo_max=0 , be_max=0;


                for(i=0; i < TIME_PERIOD; i++)
                {

                        __asm__ __volatile__("");
                        // time an access to the BO cache set
                        start = rdtsc();
                        *(unsigned int*)BO_0 = i;
                        *(unsigned int*)BO_1 = i+1;
                        *(unsigned int*)BO_2 = i+2;
                        *(unsigned int*)BO_3 = i+3;
                        *(unsigned int*)BO_4 = i+4;
                        *(unsigned int*)BO_5 = i+5;
                        *(unsigned int*)BO_6 = i+6;
                        *(unsigned int*)BO_7 = i+7;
                        end = rdtsc();

                        if(end - start <= THRESHOLD) BO_freq[(end - start)]++;

                        // time an access to the BE cache set
                        start = rdtsc();
                        *(unsigned int*)BE_0 = i;
                        *(unsigned int*)BE_1 = i+1;
                        *(unsigned int*)BE_2 = i+2;
                        *(unsigned int*)BE_3 = i+3;
                        *(unsigned int*)BE_4 = i+4;
                        *(unsigned int*)BE_5 = i+5;
                        *(unsigned int*)BE_6 = i+6;
                        *(unsigned int*)BE_7 = i+7;
-- VISUAL --                                                                              184,32-53     63%
```



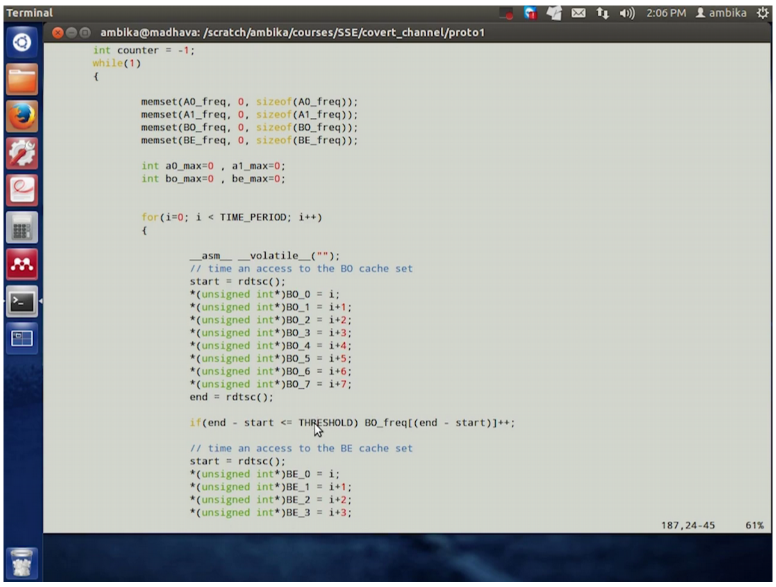
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```


#if defined(__i386__)
static __inline__ unsigned long long rdtsc(void)
{
        unsigned long long int x;
        __asm__ volatile("xorl %%eax,%%eax\n cpuid \n" ::: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
        __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
        __asm__ volatile("xorl %%eax,%%eax\n cpuid \n" ::: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
        return x;
}

#elif defined(__x86_64__)
static __inline__ unsigned long long rdtsc(void)
{
        unsigned hi, lo;
        __asm__ __volatile__ ("xorl %%eax,%%eax\n cpuid \n" ::: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
        __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
        __asm__ __volatile__ ("xorl %%eax,%%eax\n cpuid \n" ::: "%eax", "%ebx", "%ecx", "%edx"); // flush pipeline
        return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}

#endif


/* declare an array as large as the L1 D cache; align it to an
   offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));

/*
 * main takes 4 command line parameters. Each parameter represents a cache set (0 - 63) and must
 * not overlap. The first two parameters are used for sending bits (0 or 1). The second two
 * parameters are used for controlling the communication (it takes values even or odd).
 */
int main(int c,char *argv[])
{
                                                                                          68,25-32      19%
```

Okay, so let us now look at the receiver part of the code, so as we see the receiver part is very similar to that of the centre, we still have these memory accesses which are done but the major difference here is that the block of memory accesses is actually timed, so the timing is done by the function call rdtsc, so this rdtsc function returns what is known as the time stamp prior to actually executing these instructions we measure the timestamp and also at the end of these instruction executions.

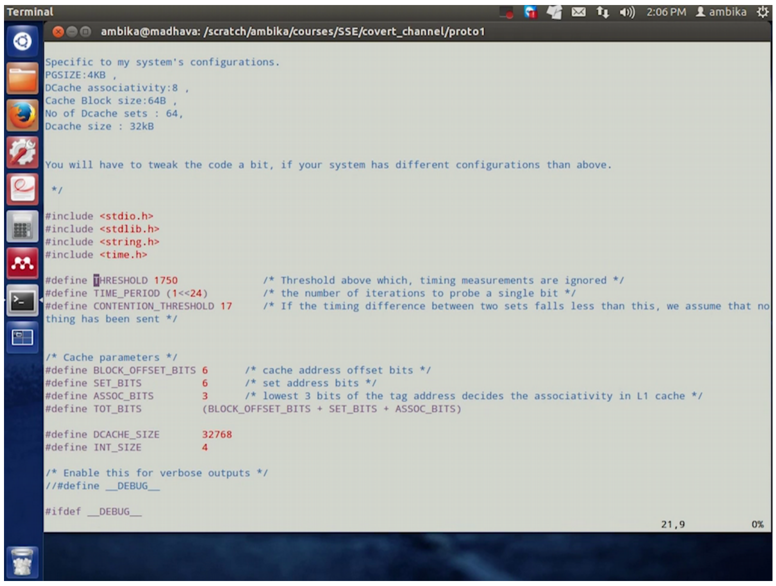
Therefore the end minus start would give you the time taken to execute these instructions, rdtsc function are received over here uses something known as the rdtsc instruction which is supported by all Intel platforms or Intel processors. So this instruction essentially reads a timestamp counter, so all Intel machines maintain a counter, it is a 128 bit counter which starts at 0 at the time of reset and implements at every clock pulse, so the rdtsc instruction then reads the timestamp counter at that particular incident. So we have 2 versions of this rdtsc function other one is for 32-bit and the other is for 64-bit and if you are using trying to actually replicated this particular cover channel on your own machine, you could suitably enable one of these 2 functions.

(Refer Slide Time: 2:01)



ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

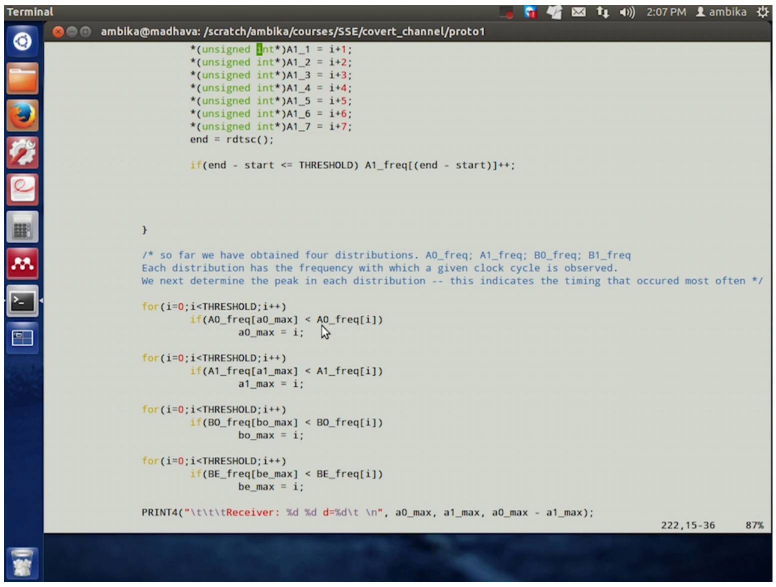
```
        int counter = -1;
        while(1)
        {

                memset(A0_freq, 0, sizeof(A0_freq));
                memset(A1_freq, 0, sizeof(A1_freq));
                memset(BO_freq, 0, sizeof(BO_freq));
                memset(BE_freq, 0, sizeof(BE_freq));

                int a0_max=0 , a1_max=0;
                int bo_max=0 , be_max=0;


                for(i=0; i < TIME_PERIOD; i++)
                {

                        __asm__ __volatile__("");
                        // time an access to the BO cache set
                        start = rdtsc();
                        *(unsigned int*)BO_0 = i;
                        *(unsigned int*)BO_1 = i+1;
                        *(unsigned int*)BO_2 = i+2;
                        *(unsigned int*)BO_3 = i+3;
                        *(unsigned int*)BO_4 = i+4;
                        *(unsigned int*)BO_5 = i+5;
                        *(unsigned int*)BO_6 = i+6;
                        *(unsigned int*)BO_7 = i+7;
                        end = rdtsc();

                        if(end - start <= THRESHOLD) BO_freq[(end - start)]++;

                        // time an access to the BE cache set
                        start = rdtsc();
                        *(unsigned int*)BE_0 = i;
                        *(unsigned int*)BE_1 = i+1;
                        *(unsigned int*)BE_2 = i+2;
                        *(unsigned int*)BE_3 = i+3;
                                                                187,24-45     61%
```



Terminal   2:06 PM ambika

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

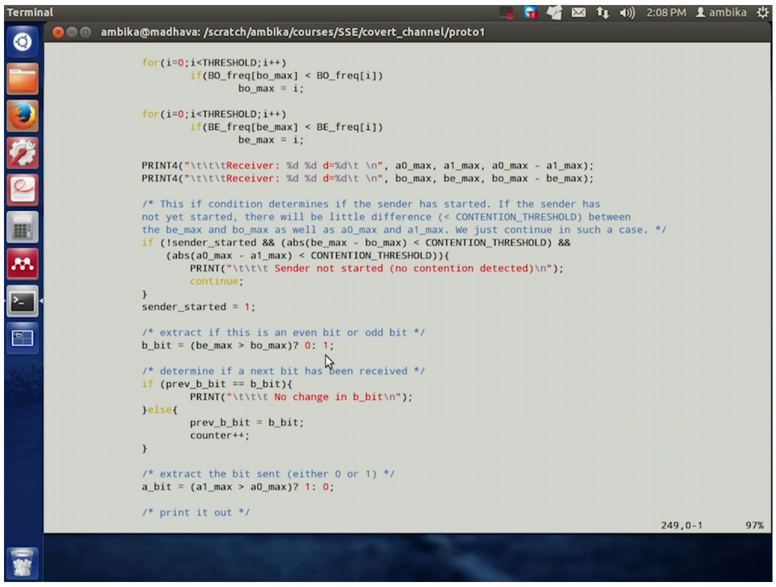
```
Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB


You will have to tweak the code a bit, if your system has different configurations than above.

 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define THRESHOLD 1750              /* Threshold above which, timing measurements are ignored */
#define TIME_PERIOD (1<<24)         /* the number of iterations to probe a single bit */
#define CONTENTION_THRESHOLD 17     /* If the timing difference between two sets falls less than this, we assume that no
thing has been sent */


/* Cache parameters */
#define BLOCK_OFFSET_BITS 6      /* cache address offset bits */
#define SET_BITS          6      /* set address bits */
#define ASSOC_BITS        3      /* lowest 3 bits of the tag address decides the associativity in L1 cache */
#define TOT_BITS          (BLOCK_OFFSET_BITS + SET_BITS + ASSOC_BITS)

#define DCACHE_SIZE       32768
#define INT_SIZE          4

/* Enable this for verbose outputs */
//#define __DEBUG__

#ifdef __DEBUG__
                                                                                          21,9          0%
```



Terminal   2:07 PM ambika

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

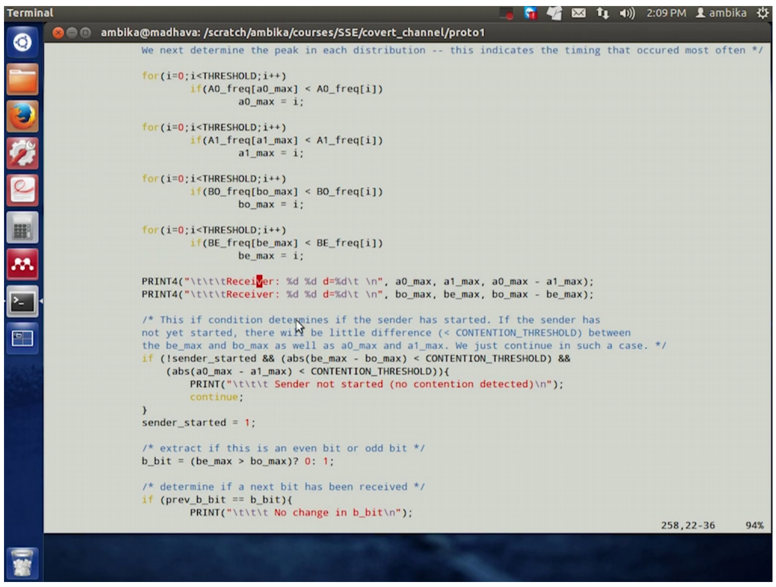
```
                        *(unsigned int*)A1_1 = i+1;
                        *(unsigned int*)A1_2 = i+2;
                        *(unsigned int*)A1_3 = i+3;
                        *(unsigned int*)A1_4 = i+4;
                        *(unsigned int*)A1_5 = i+5;
                        *(unsigned int*)A1_6 = i+6;
                        *(unsigned int*)A1_7 = i+7;
                        end = rdtsc();

                        if(end - start <= THRESHOLD) A1_freq[(end - start)]++;




                }

                /* so far we have obtained four distributions. A0_freq; A1_freq; B0_freq; B1_freq
                Each distribution has the frequency with which a given clock cycle is observed.
                We next determine the peak in each distribution -- this indicates the timing that occured most often */

                for(i=0;i<THRESHOLD;i++)
                        if(A0_freq[a0_max] < A0_freq[i])
                                a0_max = i;

                for(i=0;i<THRESHOLD;i++)
                        if(A1_freq[a1_max] < A1_freq[i])
                                a1_max = i;

                for(i=0;i<THRESHOLD;i++)
                        if(BO_freq[bo_max] < BO_freq[i])
                                bo_max = i;

                for(i=0;i<THRESHOLD;i++)
                        if(BE_freq[be_max] < BE_freq[i])
                                be_max = i;

                PRINT4("\t\t\tReceiver: %d %d d=%d\t \n", a0_max, a1_max, a0_max - a1_max);
                                                                                          222,15-36     87%
```



Terminal   2:08 PM ambika

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1

```
                for(i=0;i<THRESHOLD;i++)
                        if(BO_freq[bo_max] < BO_freq[i])
                                bo_max = i;

                for(i=0;i<THRESHOLD;i++)
                        if(BE_freq[be_max] < BE_freq[i])
                                be_max = i;

                PRINT4("\t\t\tReceiver: %d %d d=%d\t \n", a0_max, a1_max, a0_max - a1_max);
                PRINT4("\t\t\tReceiver: %d %d d=%d\t \n", bo_max, be_max, bo_max - be_max);

                /* This if condition determines if the sender has started. If the sender has
                not yet started, there will be little difference (< CONTENTION_THRESHOLD) between
                the be_max and bo_max as well as a0_max and a1_max. We just continue in such a case. */
                if (!sender_started && (abs(be_max - bo_max) < CONTENTION_THRESHOLD) &&
                    (abs(a0_max - a1_max) < CONTENTION_THRESHOLD)){
                        PRINT("\t\t\t Sender not started (no contention detected)\n");
                        continue;
                }
                sender_started = 1;

                /* extract if this is an even bit or odd bit */
                b_bit = (be_max > bo_max)? 0: 1;

                /* determine if a next bit has been received */
                if (prev_b_bit == b_bit){
                        PRINT("\t\t\t No change in b_bit\n");
                }else{
                        prev_b_bit = b_bit;
                        counter++;
                }

                /* extract the bit sent (either 0 or 1) */
                a_bit = (a1_max > a0_max)? 1: 0;

                /* print it out */
                                                                                          249,0-1       97%
```



```
We next determine the peak in each distribution -- this indicates the timing that occured most often */

for(i=0;i<THRESHOLD;i++)
        if(A0_freq[a0_max] < A0_freq[i])
                a0_max = i;

for(i=0;i<THRESHOLD;i++)
        if(A1_freq[a1_max] < A1_freq[i])
                a1_max = i;

for(i=0;i<THRESHOLD;i++)
        if(BO_freq[bo_max] < BO_freq[i])
                bo_max = i;

for(i=0;i<THRESHOLD;i++)
        if(BE_freq[be_max] < BE_freq[i])
                be_max = i;

PRINT4("\t\t\tReceiver: %d %d d=%d\t \n", a0_max, a1_max, a0_max - a1_max);
PRINT4("\t\t\tReceiver: %d %d d=%d\t \n", bo_max, be_max, bo_max - be_max);

/* This if condition determines if the sender has started. If the sender has
not yet started, there will be little difference (< CONTENTION_THRESHOLD) between
the be_max and bo_max as well as a0_max and a1_max. We just continue in such a case. */
if (!sender_started && (abs(be_max - bo_max) < CONTENTION_THRESHOLD) &&
    (abs(a0_max - a1_max) < CONTENTION_THRESHOLD)){
        PRINT("\t\t\t Sender not started (no contention detected)\n");
        continue;
}
sender_started = 1;

/* extract if this is an even bit or odd bit */
b_bit = (be_max > bo_max)? 0: 1;

/* determine if a next bit has been received */
if (prev_b_bit == b_bit){
        PRINT("\t\t\t No change in b_bit\n");
                                                                258,22-36     94%
```

So that being said the next thing we actually look at is there is something known as a threshold which is also defined, so the end minus start gives you the time required to execute these 8 instructions and we see that the time recorded for this set of instructions may be extremely noisy, the noise may come due to certain other aspects which are going on in the processor for example a page fault, interrupts and so on and these needs to be filtered out, so the threshold value should be selected on per processor basis or per system basis and it does not very accurate.

In this code we have hash design the threshold to a value of 1750, so this value should be good enough to filter out most of the noise due to interrupts and other aspects like context and so on but yet the big large enough to permit or to be able to distinguish between cache hits and cache misses, so for any of these timing which is less than the threshold we maintain something known as a frequency distribution table and identify how often a particular time is observed. So at the end of this particular iteration that is the time period and recollect that the time period is set to 2 power 24, we use these frequency distribution table to identify whether a cache hit or cache miss is observed.

Now a cache miss would imply that the sender has actually sent something through that particular port, so it would mean that the sender has actually loaded something in that corresponding set and therefore has evicted the receiver data from that set and therefore when the receiver is actually accessing through that set it would result in a cache miss and memory access would require to go to a lower level cache like the L2 LLC or the DRAM to complete the memory access.

So this would typically take longer which we have timed and this timing would actually show up in the frequency distribution, so in this way observing the various frequency distribution for the various sets 10, 20, 30 and 40 the receiver would be able to (())(4:32) whatever has been transmitted by the sender, so it would then be able to identify 0s and 1s it would be able to identify whether it was odd bit or even bit that the sender had actually transmitted.

So in this way what we have seen is 2 independent processes a sender and a receiver being able to communicate with each other through this cover channel this very indirect channel and being able to break all the security that is achieved by this underlined platform, such cover channels may not be restricted only to cache but other aspects or other processor and system level features such as in the file system page falls another things like key strokes and so on may be also used as a source of cover channel.