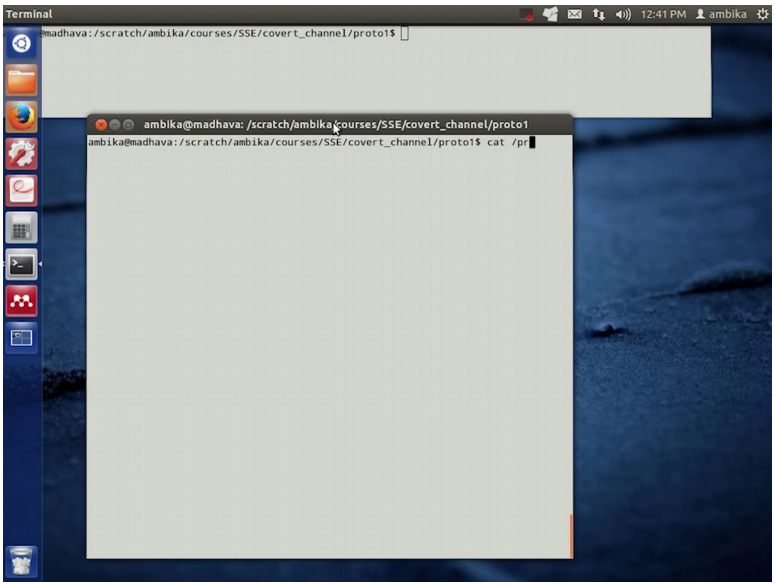
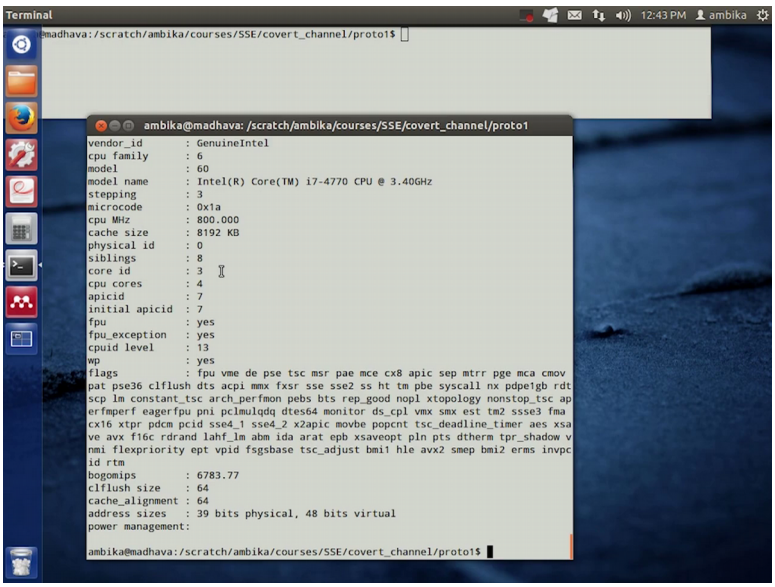
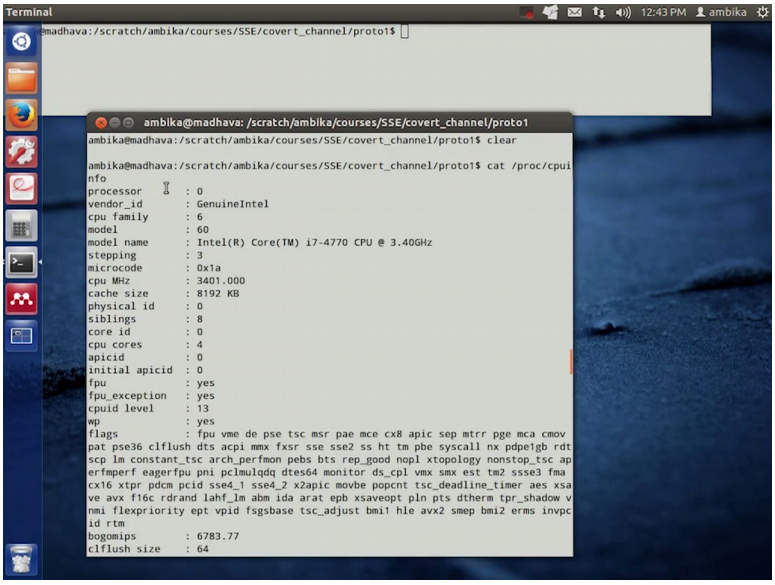
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
'Demo- Cache-timing based Covert Channel - Part 1'

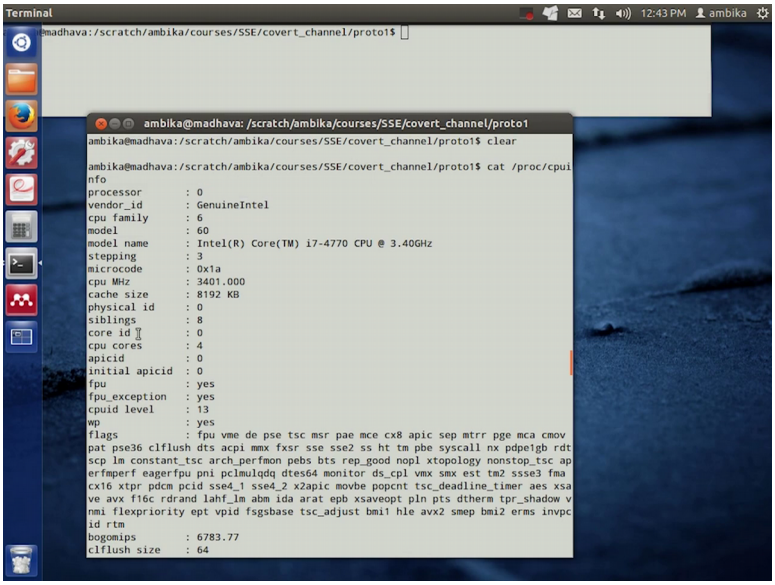
Hello and welcome to this demonstration in the course for secure systems engineering. In this particular demonstration we will look at cover channels, we have already seen the theory about cover channels and how they can be established through the cache and in this demonstration we will actually look at creating such a cover channel and how we could actually transfer information from one process to another.

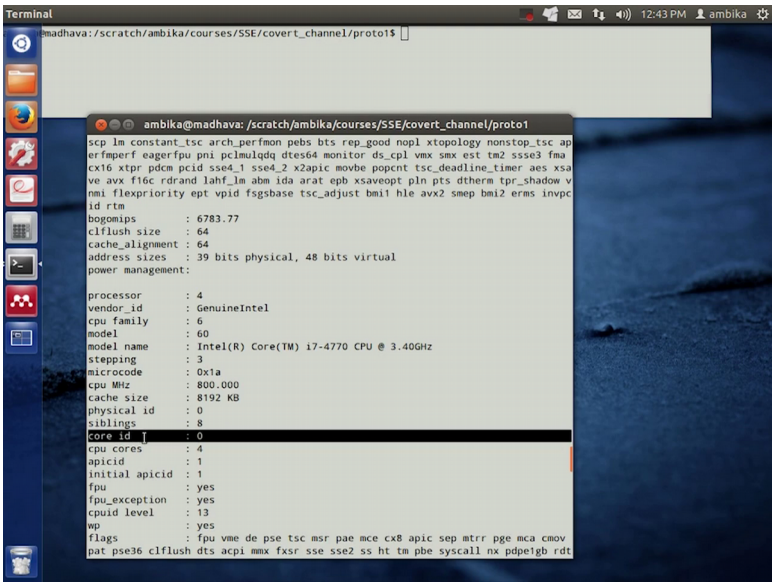
(Refer Slide Time: 0:45)

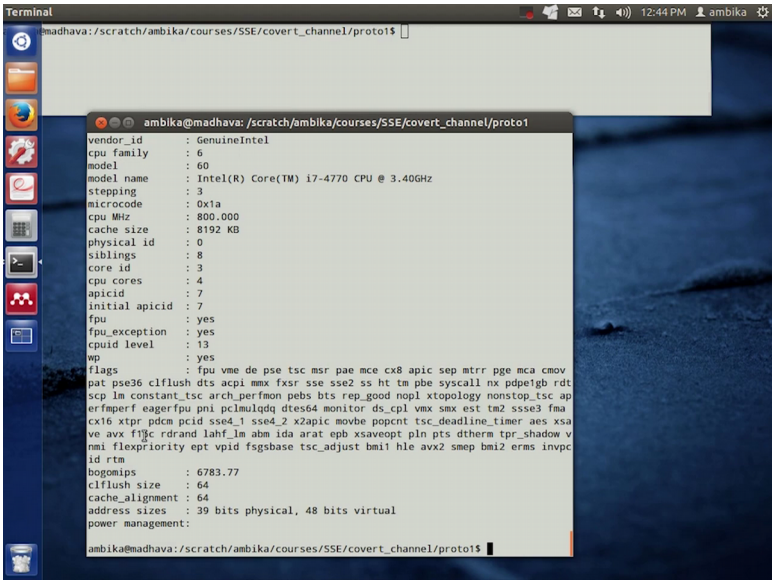


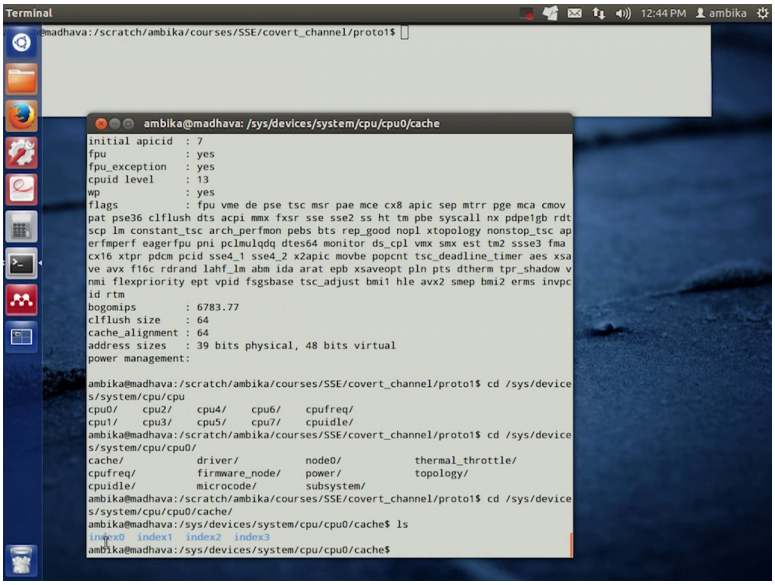


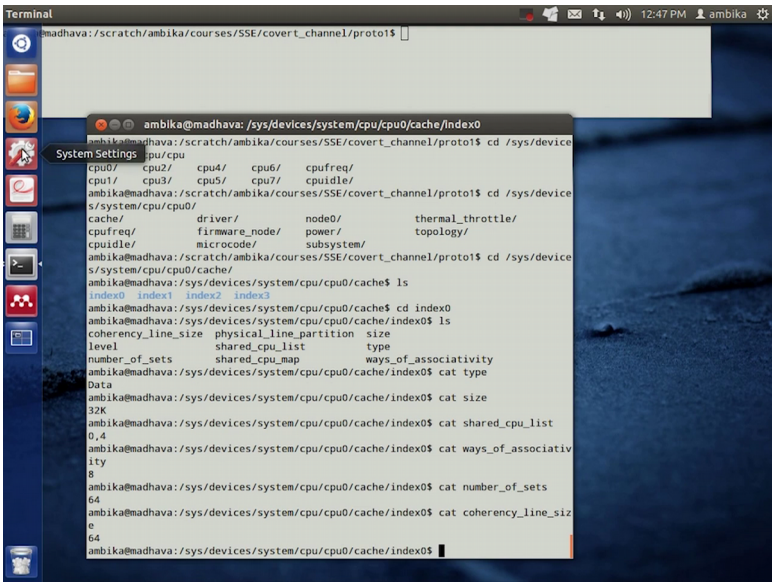

Terminal
12:43 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ clear
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpui
nfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 3401.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64


Terminal
12:43 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ clear
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cat /proc/cpui
nfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 3401.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64


Terminal
12:43 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
processor : 4
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 800.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 1
initial apicid : 1
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt


Terminal
12:44 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
vendor_id : GenuineIntel
cpu family : 6
model : 60
model name : Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
stepping : 3
microcode : 0x1a
cpu MHz : 800.000
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 3
cpu cores : 4
apicid : 7
initial apicid : 7
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$


Terminal
12:44 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /sys/devices/system/cpu/cpu0/cache
initial apicid : 7
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdt
scp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc ap
erfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma
cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm ida arat epb xsaveopt pln pts dtherm tpr_shadow v
nmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpc
id rtm
bogomips : 6783.77
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu
cpu0/ cpu2/ cpu4/ cpu6/ cpufreq/
cpu1/ cpu3/ cpu5/ cpu7/ cpuidle/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu0/
cache/ driver/ node0/ thermal_throttle/
cpufreq/ firmware_node/ power/ topology/
cpuidle/ microcode/ subsystem/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu0/cache/
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ ls
index0 index1 index2 index3
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$


Terminal
12:47 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
System Settings
ambika@madhava: /sys/devices/system/cpu/cpu0/cache/index0
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu
cpu0/ cpu2/ cpu4/ cpu6/ cpufreq/
cpu1/ cpu3/ cpu5/ cpu7/ cpuidle/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu0/
cache/ driver/ node0/ thermal_throttle/
cpufreq/ firmware_node/ power/ topology/
cpuidle/ microcode/ subsystem/
ambika@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$ cd /sys/device
s/system/cpu/cpu0/cache/
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ ls
index0 index1 index2 index3
ambika@madhava:/sys/devices/system/cpu/cpu0/cache$ cd index0
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ ls
coherency_line_size physical_line_partition size
level shared_cpu_list type
number_of_sets shared_cpu_map ways_of_associativity
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat type
Data
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat size
32K
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat shared_cpu_list
0,4
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat ways_of_associativ
ity
8
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat number_of_sets
64
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$ cat coherency_line_siz
e
64
ambika@madhava:/sys/devices/system/cpu/cpu0/cache/index0$

So this particular sector may not be runnable from the virtual machine, although we will actually provide the source code that can be downloaded and tried on your own machines. So while setting up a cover channel using the cache memory the 1st thing to identify when we are starting to design a cache-based cover channel or is to identify information about the processor and also about the cache memory present in that processor, so the machine that were using over here is a regular i7 machine from Intel and it is running an open to operating system.

The 1st thing to note is details about this processor, so this processor is 4 cores processor and these 4 cores there are 8 hyper threats, essentially per core as 2 hyper threats. If you want to try this on your own system which is running open 2 you can run the commands cache slash cat slash proc cpuinfo and it would list details of all the processor present in your system for example over here since we have mentioned that this is 4 core machine, so each core is given a particular core ID starting from 0, 1, 2 and 3, so the core ID for example over here is 3 and as we scroll upwards we see other core IDs of 1 and 0 and so on.

The next thing we also mention that in this processor we had to hyper threads sharing the same CPU core particularly what we see if we scroll up is that the processor number 0 and the processor number 4 share the same CPU core, so we can verify this as follows. So this is processor 0 and which is present in this on the core with an ID of 0. Now if you look at the processor 4 so processor 4 is also on the same for core with the same core ID.

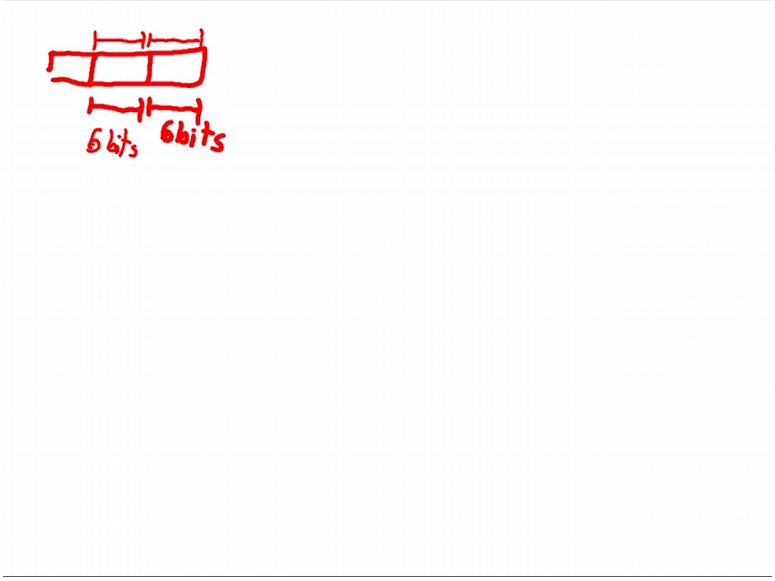
Similarly if you go through the other things and this particular machine you see that processor 1 and processor 5 are sharing the same core essentially the core ID 1 and so on. In order to create a cover channel between 2 processors what we would require is to identify 2 hyper threats which are running on the same core, so let us choose the core processors 0 and processor 4, the next thing we need to know is the cache structure for this particular processors, so this information can be obtained from the directory…sys devices CPU cache, so this particular directory list all the cache memories that are accessible from the CPU 0.

So will go into index 0 and we will look at the type of cache which is present over here and the type is D data which indicates that this particular cache represented at CPU 0 index 0 is a data cache. It has a size of 32 kilobytes as seen over here and we can also get the information about the other processes which are sharing this particular cache memory, so this can be obtained from the file shared CPU list and we see that there are 2 processors, processor 0 and processor 4 which are actually sharing this particular cache, so now if we want to build a

cache cover channel, we could have…and we want to build this cache cover channels between 2 processes and shared information between these 2 processes.

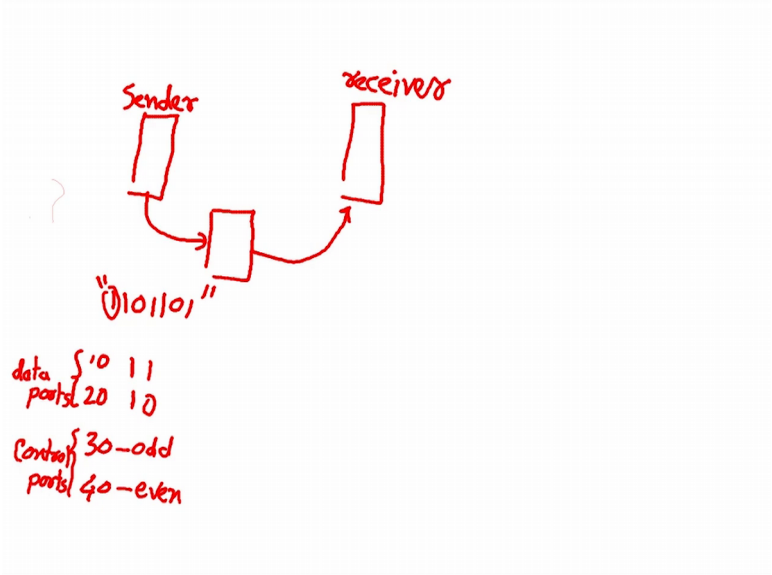
What we would be able to do is we could run one process in the CPU 0 and one process in CPU 4 and make use of the shared cache to exchange information. We can also obtain other details about this particular cache memory for example the ways of associate key is as follow so this is 8 way associative cache and the number of sets that are present is 64 okay and the size of each cache line that is present is as follows, is also 64. Based on all of this information we can next construct a typical address of such a cache.

(Refer Slide Time: 6:18)



So what we did mention was that this particular cache memory had a cache line size of 64 bits which would indicate that the lowest bits of the address, the lowest 6 bits of the address comprises addressing within the cache line size which is of 6 bits. It also had 64 cache sets and therefore in order to address 64 cache sets we need another 6 bits and therefore we have set addressing which has like another 6 bits, these 2 actually give the offset with in a line and these 6 bits provide the address for a particular set. So later on as we would see in the programs we would require to no such information in order to build our cache base cover channel. So now that we know about the cache structure and the processors within this particular machine let us next try to find out how we could actually build a cache base cover channel.

(Refer Slide Time: 7:39)



So what we want to achieve is that we have 2 processes running on the system these processors are called sender and receiver… and receiver and we assume that these 2 processors are completely isolated from each other, you could think of a sender to be a privileged process something like which is run by the system administrator and the receiver process to be something which is an privilege process, so you can also assume that these 2 sender and receiver processes are run by totally different users.

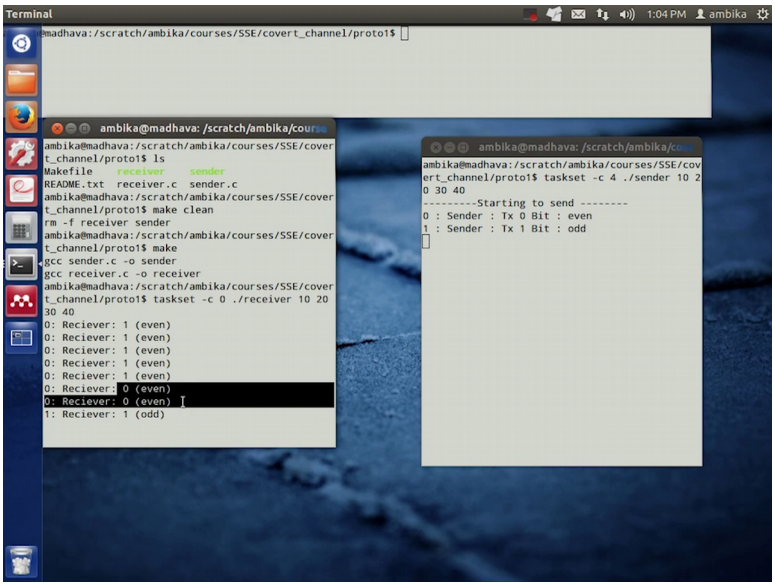
So what we do know about the operating system and the hardware is that there are various mechanisms which are incorporated in both the hardware and the system software that would ensure that this receiver would not directly be able to read or write data from the sender and vice versa, however what we have shown now is that the sender and the receiver can actually use a shared cache memory to transfer information from one process to another, so note that when cache memory is used there is no implicit way to actually…

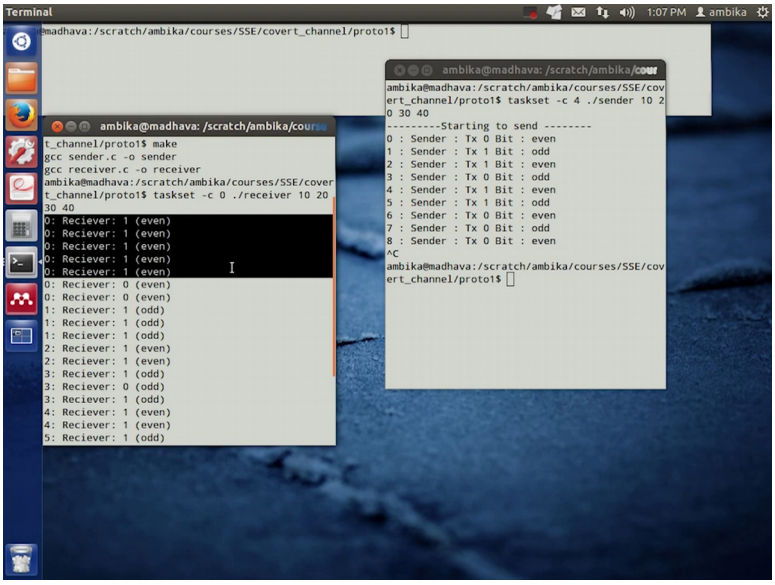
For one process to read from the cache memory but what we will see in as we have seen in the theory videos before, we could use the execution time for a memory access to pass on information from one process to the other, so let us say that we have a message and assume a binary message of say 1101101 to be sent from the sender to the receiver. What we assume further is that both of these processors that is the sender and the receiver are sharing a common cache memory and running on the same processor for example the processor 0 and processor 4, so what is done in order to create the cache cover channel is that the sender and the receiver 1st agree upon specific set of ports.

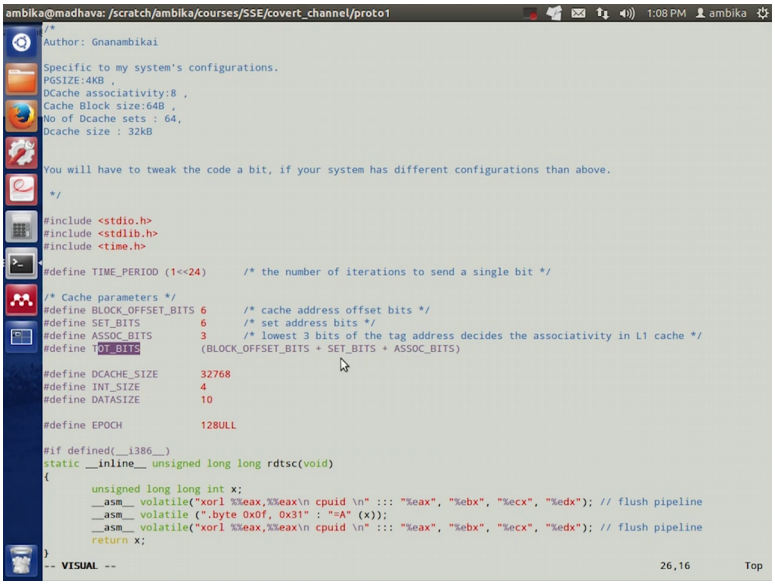
So let us call these ports as 10, 20, 30 and 40, so we call 2 these ports as the data ports, so it is called data ports size and 30 and 40 as the control ports. So these ports are prior to actually starting the cover channel, the sender and receiver agree upon these ports and essentially in our program the demonstration that we would see we would use 1 port say port 30 to send the odd bits and port 40 to send the even bits, so for example in this particular case we would have these ports sending the port number 10.

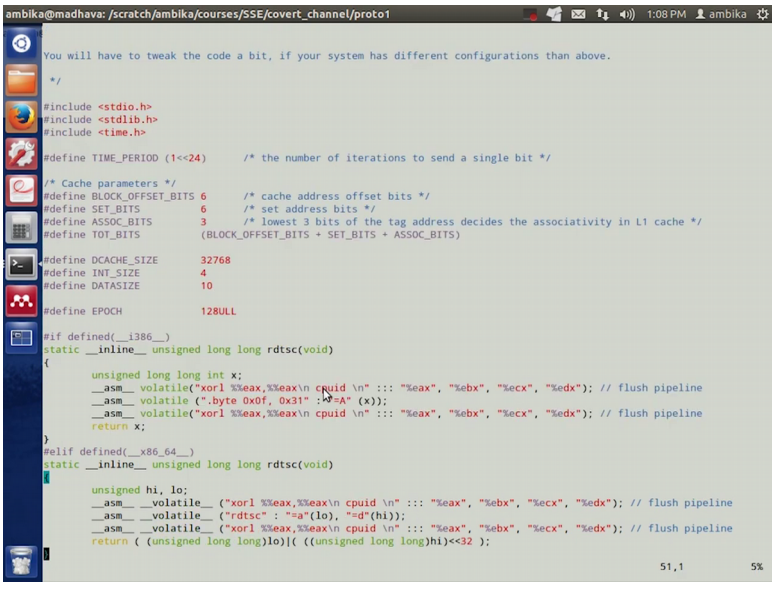
So since we start with a 0^{th} bit this particular bit this thing will be actually sent to this 1 and the next bit which is 1 again would be sent on this port 10, the 3^{rd} bit which is 0 would be sent here with the control line even and the 4^{th} bit which is here would be having the control of 30 and sent on this port, so in this way what would happen is that we are sending bit by bit from the sender to the receiver, so let us see how this program actually works.

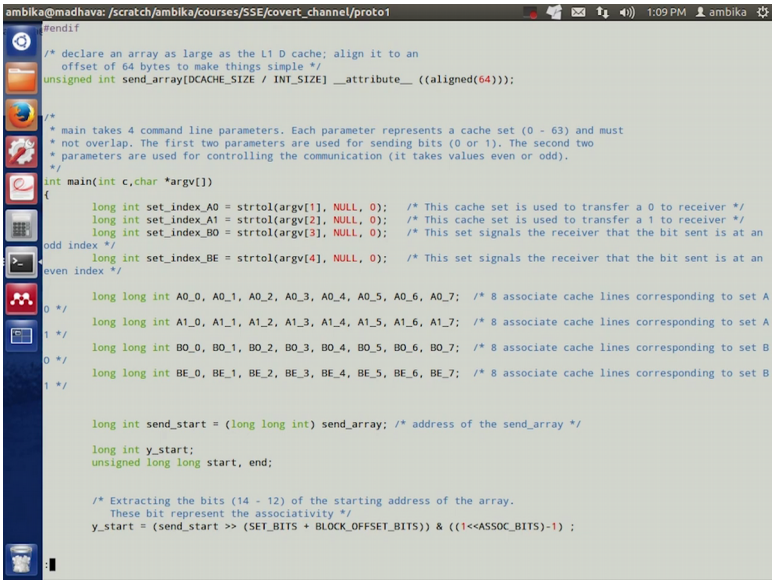
(Refer Slide Time: 11:38)

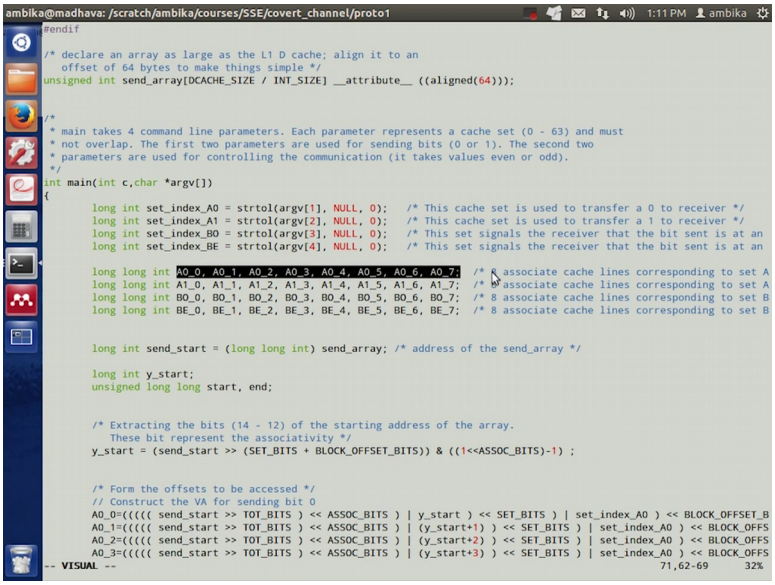


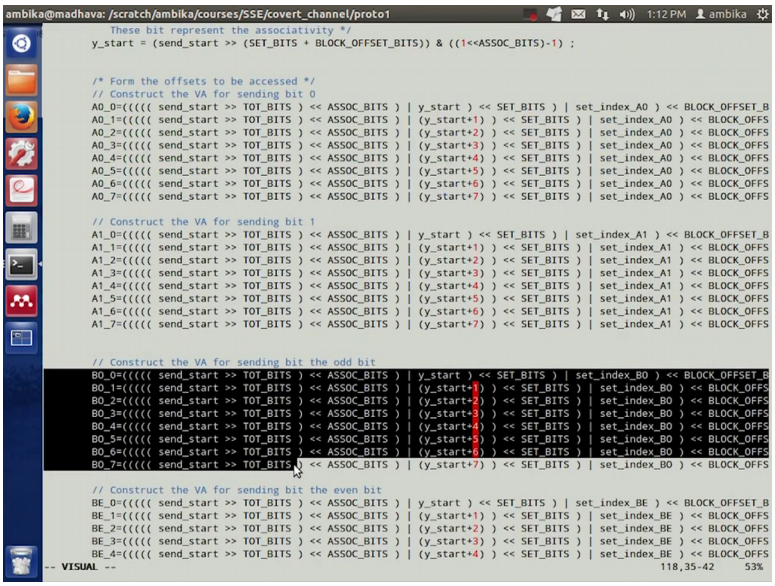

Terminal
1:07 PM ambika
@madhava:/scratch/ambika/courses/SSE/covert_channel/proto1$
ambika@madhava: /scratch/ambika/cours
t_channel/proto1$ make
gcc sender.c -o sender
gcc receiver.c -o receiver
ambika@madhava:/scratch/ambika/courses/SSE/cover
t_channel/proto1$ taskset -c 0 ./receiver 10 20
30 40
0: Reciever: 1 (even)
0: Reciever: 1 (even)
0: Reciever: 1 (even)
0: Reciever: 1 (even)
0: Reciever: 1 (even)
0: Reciever: 0 (even)
0: Reciever: 0 (even)
1: Reciever: 1 (odd)
1: Reciever: 1 (odd)
1: Reciever: 1 (odd)
2: Reciever: 1 (even)
2: Reciever: 1 (even)
3: Reciever: 1 (odd)
3: Reciever: 0 (odd)
3: Reciever: 1 (odd)
4: Reciever: 1 (even)
4: Reciever: 1 (even)
5: Reciever: 1 (odd)
ambika@madhava: /scratch/ambika/cour
ambika@madhava:/scratch/ambika/courses/SSE/cov
ert_channel/proto1$ taskset -c 4 ./sender 10 2
0 30 40
---------Starting to send --------
0 : Sender : Tx 0 Bit : even
1 : Sender : Tx 1 Bit : odd
2 : Sender : Tx 1 Bit : even
3 : Sender : Tx 0 Bit : odd
4 : Sender : Tx 1 Bit : even
5 : Sender : Tx 1 Bit : odd
6 : Sender : Tx 0 Bit : even
7 : Sender : Tx 0 Bit : odd
8 : Sender : Tx 0 Bit : even
^C
ambika@madhava:/scratch/ambika/courses/SSE/cov
ert_channel/proto1$

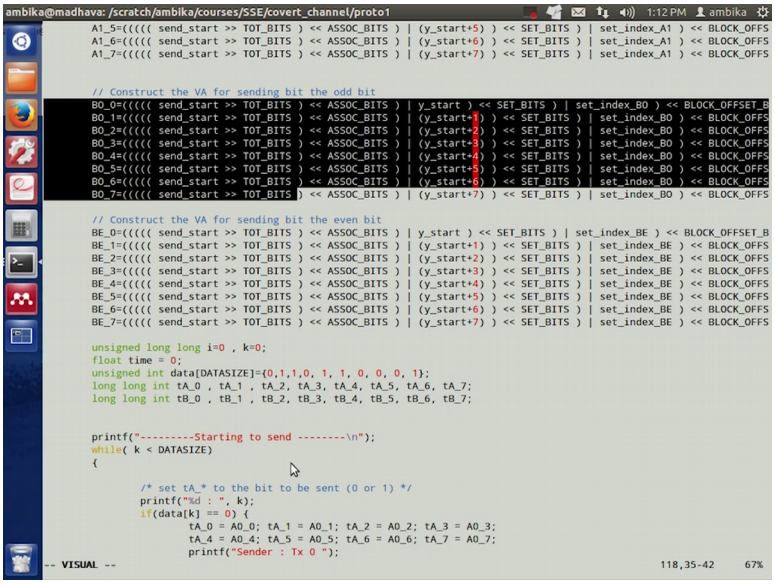

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:08 PM ambika
/*
Author: Gnanambikai
Specific to my system's configurations.
PGSIZE:4KB ,
DCache associativity:8 ,
Cache Block size:64B ,
No of Dcache sets : 64,
Dcache size : 32kB
You will have to tweak the code a bit, if your system has different configurations than above.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TIME_PERIOD (1<<24) /* the number of iterations to send a single bit */
/* 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
#define DATASIZE 10
#define EPOCH 128ULL
#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;
}
-- VISUAL --
26,16 Top

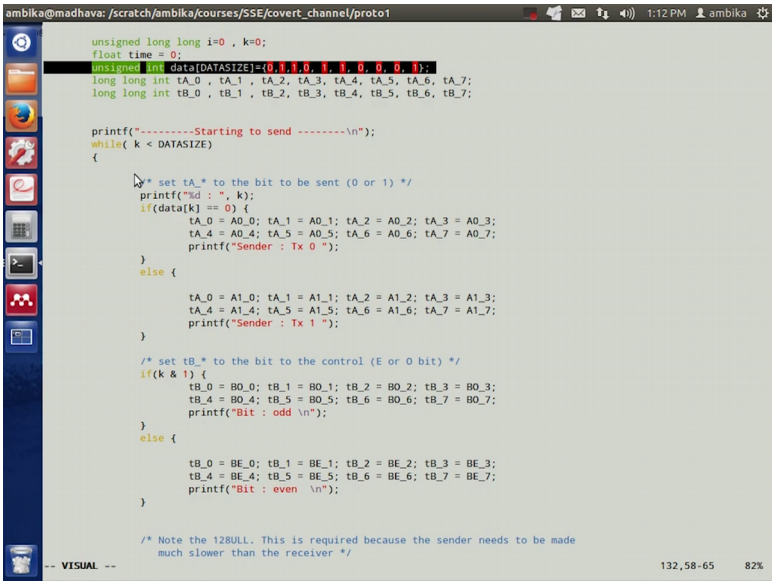

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:08 PM ambika
You will have to tweak the code a bit, if your system has different configurations than above.
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TIME_PERIOD (1<<24) /* the number of iterations to send a single bit */
/* 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
#define DATASIZE 10
#define EPOCH 128ULL
#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);
}
51,1 5%

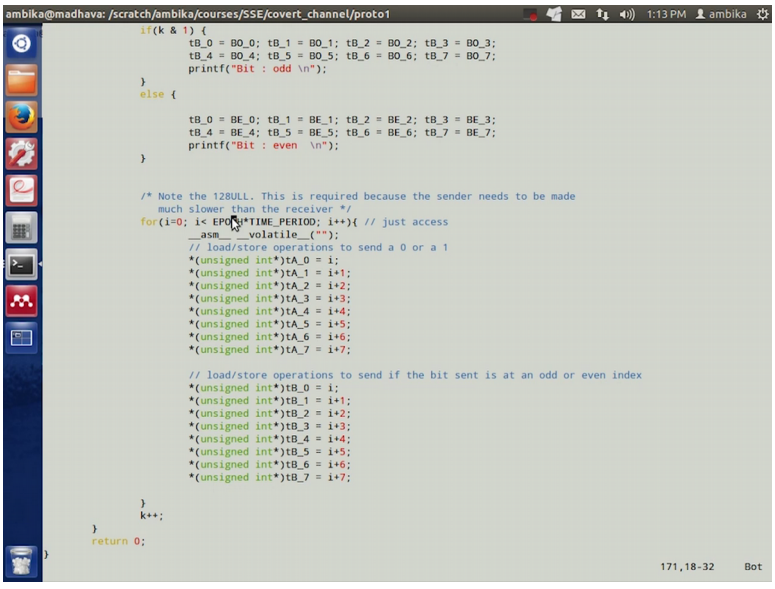

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:09 PM ambika
#endif
/* declare an array as large as the L1 D cache; align it to an
offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));
/*
* main takes 4 command line 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[])
{
long int set_index_A0 = strtol(argv[1], NULL, 0); /* This cache set is used to transfer a 0 to receiver */
long int set_index_A1 = strtol(argv[2], NULL, 0); /* This cache set is used to transfer a 1 to receiver */
long int set_index_BO = strtol(argv[3], NULL, 0); /* This set signals the receiver that the bit sent is at an
odd index */
long int set_index_BE = strtol(argv[4], NULL, 0); /* This set signals the receiver that the bit sent is at an
even index */
long long int A0_0, A0_1, A0_2, A0_3, A0_4, A0_5, A0_6, A0_7; /* 8 associate cache lines corresponding to set A
0 */
long long int A1_0, A1_1, A1_2, A1_3, A1_4, A1_5, A1_6, A1_7; /* 8 associate cache lines corresponding to set A
1 */
long long int BO_0, BO_1, BO_2, BO_3, BO_4, BO_5, BO_6, BO_7; /* 8 associate cache lines corresponding to set B
0 */
long long int BE_0, BE_1, BE_2, BE_3, BE_4, BE_5, BE_6, BE_7; /* 8 associate cache lines corresponding to set B
1 */
long int send_start = (long long int) send_array; /* address of the send_array */
long int y_start;
unsigned long long start, end;
/* Extracting the bits (14 - 12) of the starting address of the array.
These bit represent the associativity */
y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1) ;

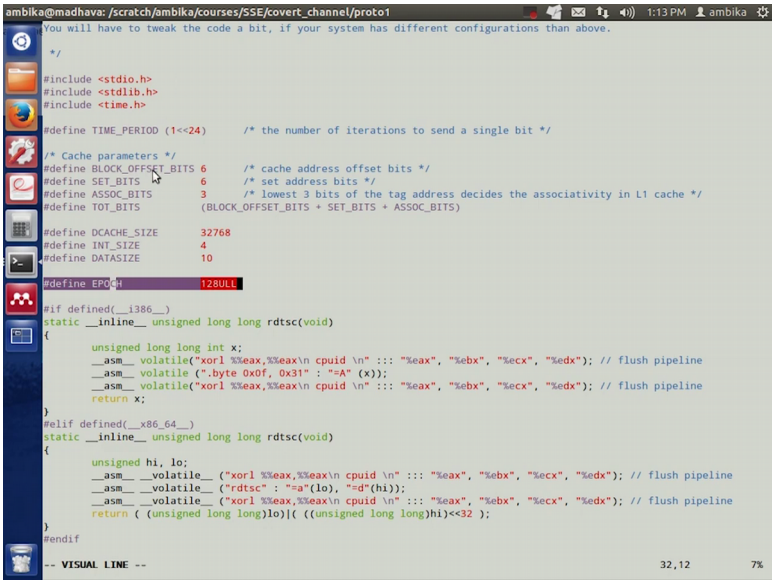

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:11 PM ambika
#endif
/* declare an array as large as the L1 D cache; align it to an
offset of 64 bytes to make things simple */
unsigned int send_array[DCACHE_SIZE / INT_SIZE] __attribute__ ((aligned(64)));
/*
* main takes 4 command line 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[])
{
long int set_index_A0 = strtol(argv[1], NULL, 0); /* This cache set is used to transfer a 0 to receiver */
long int set_index_A1 = strtol(argv[2], NULL, 0); /* This cache set is used to transfer a 1 to receiver */
long int set_index_BO = strtol(argv[3], NULL, 0); /* This set signals the receiver that the bit sent is at an
long int set_index_BE = strtol(argv[4], NULL, 0); /* This set signals the receiver that the bit sent is at an
long long int A0_0, A0_1, A0_2, A0_3, A0_4, A0_5, A0_6, A0_7; /* 8 associate cache lines corresponding to set A
long long int A1_0, A1_1, A1_2, A1_3, A1_4, A1_5, A1_6, A1_7; /* 8 associate cache lines corresponding to set A
long long int BO_0, BO_1, BO_2, BO_3, BO_4, BO_5, BO_6, BO_7; /* 8 associate cache lines corresponding to set B
long long int BE_0, BE_1, BE_2, BE_3, BE_4, BE_5, BE_6, BE_7; /* 8 associate cache lines corresponding to set B
long int send_start = (long long int) send_array; /* address of the send_array */
long int y_start;
unsigned long long start, end;
/* Extracting the bits (14 - 12) of the starting address of the array.
These bit represent the associativity */
y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1) ;
/* Form the offsets to be accessed */
// Construct the VA for sending bit 0
A0_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_A0) << BLOCK_OFFSET_B
A0_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
-- VISUAL --
71,62-69 32%

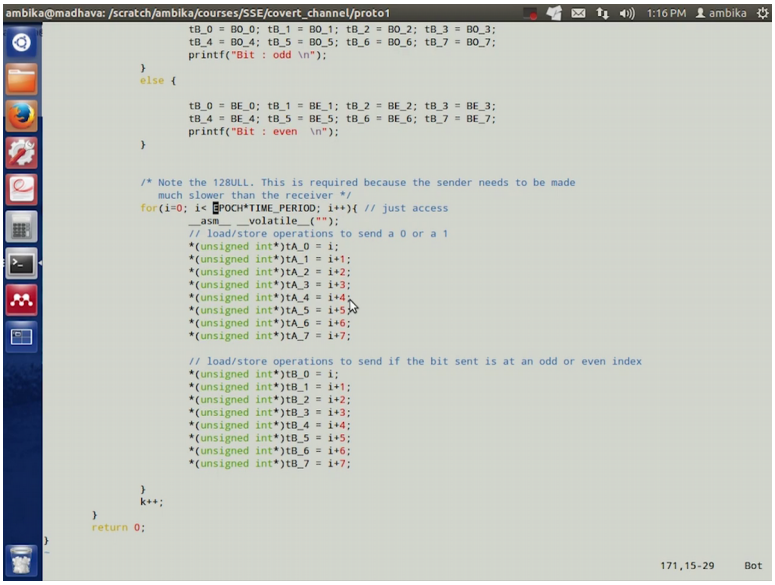

ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:12 PM ambika
These bit represent the associativity */
y_start = (send_start >> (SET_BITS + BLOCK_OFFSET_BITS)) & ((1<<ASSOC_BITS)-1) ;
/* Form the offsets to be accessed */
// Construct the VA for sending bit 0
A0_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_A0) << BLOCK_OFFSET_B
A0_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
A0_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_A0) << BLOCK_OFFS
// Construct the VA for sending bit 1
A1_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_A1) << BLOCK_OFFSET_B
A1_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
// Construct the VA for sending bit the odd bit
BO_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_BO) << BLOCK_OFFSET_B
BO_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
// Construct the VA for sending bit the even bit
BE_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_BE) << BLOCK_OFFSET_B
BE_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
-- VISUAL --
118,35-42 53%


ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:12 PM ambika
A1_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
A1_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_A1) << BLOCK_OFFS
// Construct the VA for sending bit the odd bit
BO_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_BO) << BLOCK_OFFSET_B
BO_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
BO_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_BO) << BLOCK_OFFS
// Construct the VA for sending bit the even bit
BE_0=(((((send_start >> TOT_BITS) << ASSOC_BITS) | y_start) << SET_BITS) | set_index_BE) << BLOCK_OFFSET_B
BE_1=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+1)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_2=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+2)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_3=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+3)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_4=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+4)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_5=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+5)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_6=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+6)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
BE_7=(((((send_start >> TOT_BITS) << ASSOC_BITS) | (y_start+7)) << SET_BITS) | set_index_BE) << BLOCK_OFFS
unsigned long long i=0 , k=0;
float time = 0;
unsigned int data[DATASIZE]={0,1,1,0, 1, 1, 0, 0, 1};
long long int tA_0 , tA_1 , tA_2, tA_3, tA_4, tA_5, tA_6, tA_7;
long long int tB_0 , tB_1 , tB_2, tB_3, tB_4, tB_5, tB_6, tB_7;
printf("---------Starting to send --------\n");
while(k < DATASIZE)
{
/* set tA_* to the bit to be sent (0 or 1) */
printf("%d : ", k);
if(data[k] == 0) {
tA_0 = A0_0; tA_1 = A0_1; tA_2 = A0_2; tA_3 = A0_3;
tA_4 = A0_4; tA_5 = A0_5; tA_6 = A0_6; tA_7 = A0_7;
printf("Sender : Tx 0 ");
-- VISUAL --
118,35-42
67%


ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:12 PM ambika
unsigned long long i=0 , k=0;
float time = 0;
unsigned int data[DATASIZE]={0,1,1,0, 1, 1, 0, 0, 1};
long long int tA_0 , tA_1 , tA_2, tA_3, tA_4, tA_5, tA_6, tA_7;
long long int tB_0 , tB_1 , tB_2, tB_3, tB_4, tB_5, tB_6, tB_7;
printf("---------Starting to send --------\n");
while(k < DATASIZE)
{
/* set tA_* to the bit to be sent (0 or 1) */
printf("%d : ", k);
if(data[k] == 0) {
tA_0 = A0_0; tA_1 = A0_1; tA_2 = A0_2; tA_3 = A0_3;
tA_4 = A0_4; tA_5 = A0_5; tA_6 = A0_6; tA_7 = A0_7;
printf("Sender : Tx 0 ");
}
else {
tA_0 = A1_0; tA_1 = A1_1; tA_2 = A1_2; tA_3 = A1_3;
tA_4 = A1_4; tA_5 = A1_5; tA_6 = A1_6; tA_7 = A1_7;
printf("Sender : Tx 1 ");
}
/* set tB_* to the bit to the control (E or O bit) */
if(k & 1) {
tB_0 = BO_0; tB_1 = BO_1; tB_2 = BO_2; tB_3 = BO_3;
tB_4 = BO_4; tB_5 = BO_5; tB_6 = BO_6; tB_7 = BO_7;
printf("Bit : odd \n");
}
else {
tB_0 = BE_0; tB_1 = BE_1; tB_2 = BE_2; tB_3 = BE_3;
tB_4 = BE_4; tB_5 = BE_5; tB_6 = BE_6; tB_7 = BE_7;
printf("Bit : even \n");
}
/* Note the 128ULL. This is required because the sender needs to be made
much slower than the receiver */
-- VISUAL --
132,58-65
82%


ambika@madhava: /scratch/ambika/courses/SSE/covert_channel/proto1
1:13 PM ambika
if(k & 1) {
tB_0 = BO_0; tB_1 = BO_1; tB_2 = BO_2; tB_3 = BO_3;
tB_4 = BO_4; tB_5 = BO_5; tB_6 = BO_6; tB_7 = BO_7;
printf("Bit : odd \n");
}
else {
tB_0 = BE_0; tB_1 = BE_1; tB_2 = BE_2; tB_3 = BE_3;
tB_4 = BE_4; tB_5 = BE_5; tB_6 = BE_6; tB_7 = BE_7;
printf("Bit : even \n");
}
/* Note the 128ULL. This is required because the sender needs to be made
much slower than the receiver */
for(i=0; i< EPOCH*TIME_PERIOD; i++){ // just access
__asm__ __volatile__("");
// load/store operations to send a 0 or a 1
(unsigned int)tA_0 = i;
(unsigned int)tA_1 = i+1;
(unsigned int)tA_2 = i+2;
(unsigned int)tA_3 = i+3;
(unsigned int)tA_4 = i+4;
(unsigned int)tA_5 = i+5;
(unsigned int)tA_6 = i+6;
(unsigned int)tA_7 = i+7;
// load/store operations to send if the bit sent is at an odd or even index
(unsigned int)tB_0 = i;
(unsigned int)tB_1 = i+1;
(unsigned int)tB_2 = i+2;
(unsigned int)tB_3 = i+3;
(unsigned int)tB_4 = i+4;
(unsigned int)tB_5 = i+5;
(unsigned int)tB_6 = i+6;
(unsigned int)tB_7 = i+7;
}
k++;
}
return 0;
}
171,18-32
Bot





So over here we have 2 completely different programs one is known as the receiver.c the other one is the sender.c will also be sharing these programs with you, so you can actually try this out on your machines. So the 1st thing we do is do a make clean and make and we obtain 2 executables one is a sender and the 2nd is the receiver. So what we will shows is that it is possible to actually communicate information from the sender through the receiver through the shared cache, so before going into details about how this program is actually working we will just look at the demonstration first.

The 1st thing to do is to start the receiver and we need to ensure that the cache is shared and as we have seen earlier in this video the processor 0 and the processor 4 are essentially sharing the same CPU core thus we need to ensure that both these processors the receiver and the

server are executing on exactly this core, so we can do this by the task that command taskset minus C 0.

Slash receiver and we provided the agreed-upon ports over here, so 10, 20, 30, 40 so recollect that we define the ports in the cache cover channel as essentially the different cache sets, so there were 64 cache sets in our L1 data cache and we have decided to choose these 4 cache sets, cache set 10, 20, 30 and 44 for our communication. 2 of these cache sets 10 and 20 are used for transferring data while the cache set 30 and 40 are used for control. So let us start the receiver and in a totally isolated environment start the server as follows, so we need to ensure that the server is running on the CPU 4.

So we run the tasks set specify the CPU that we want to run in this case it is 4 with exactly the same agreed-upon ports 10, 20, 30 and 40 so what is happening now is that the sender is beginning to send a 0^{th} bit it is at a time even location and we see that this sender is essentially through the cache able to communicate to the receiver, so apparently this particular 0 bit has been received over here by the receiver. The next bit which is sent by the sender is 1, so this is the odd bit and we see that the receiver has received this particular weight, so let us look for some more time to see more bits being transmitted.

The 3^{rd} bit being send is 1 again so this is an even bit and we will see that after sometime the receiver has indeed received this bit as well, so the bit number 2 is the even bit got a value of 1, so we can just wait for may be a minute or so to see more bits being transmitted, so this is an extremely slow process but what it signifies is that these 2 process sender and receiver in spite of the heavy protection provided by the operating system and hardware have been able to actually communicate with each other, so this was on 2 regular processes but we could also demonstrate this and various other infrastructures for example even with the trusted execution environment such as the SGX and (())(15:45) that we have studied earlier in this lecture such kind of cover channels can be done from a trusted environment to the untrusted appointment.

So one thing that de-markets a particular cover channel is the rate at which data can be transferred from the sender to the receiver, so over here as we see it is an extremely slow process, so typically the units would be something like bits per second and as we see over here the score is not very optimised. This seems to be much less than 1 bit per second but nevertheless we see that attackers are quite motivated and the rate of transmission is not a very critical issue for attackers as long as the information can be obtained on the other side.

Now that we have seen these programs work, so let us look a little more in detail about what is happening inside this will, so let us stop the receiver and the server. Now one thing for you think about is that recollect that…or if you just go back on the video you will see that there are some bits that gets received even though the sender has not yet started, so one thing for you to think about is why such a thing has actually happened.

Okay so let us go into the code we will open our editor and look at the sender code first okay so this code has been written by Gnanambikai who is a Ph.D. student at IIT Madras, so important thing showing here is this particular time period and we will prefer to this later on in this particular code and what we see is that various attributes about the cache memory present in the system at least the cache memory that has been targeted for this cover channel has been hashed defined over here for example the de cache size of 32 kilobytes. Number of set bits which is 6 over here because we have 64 different sets.

The offset bids within a cache line is again 6 bits because each cache line is of 64 bits and something known as total bi ts which represents the associated bits, set bits and block address bits. One thing what was important over here was that the associativity of the cache memory, so recollect that this particular memory is an 8 way associative cache and therefore in order to create conflict with a particular set there should be 8 addresses following on the same cache set.

The 8 addresses should be designed in such a way or should be crafted in such a way such that each of them feels a particular way in a cache corresponding to that set. So the arguments that are provided over here that is the control ports and the data ports that is 10, 20, 30 and 40 are taken from the command line and set into these various integers set index A0, A1, B0 and B1, so A0 and A1 are used to transfer 0 and 1 while B0 and the BE are used for the control ports and where they are used to transfer the odd bits and the even bits respectively.

The next thing we do is we craft different addresses, note that we require 4 sets of addresses, each is a collection of 8 different addresses, so this for example A00 to A07 is an 8 way corresponds to the 8 way associative cache. So I not go into details about this but giving the fact that what we discussed earlier in this video, you can see how the 8 addresses are crafted, so that all of them would access exactly the same set. We assume policy such as the least recently used to be implemented in the cache and assume the fact that all of these 8 addresses would fall in the same cache sets but in different ways of the cache.

So similarly we have A0, A1 addresses being crafted B0 addresses and the BE addresses being crafted. The next thing would be quite interesting so we would look at how the data is being transmitted. We have defined the data to be transmitted from the sender to the receiver as follows, so the data comprises of this binary bits 011 and so on and what we do over here is that we start to craft memory and start to actually send them from the sender to the receiver, so important in this particular thing is this particular fall loop.

So each iteration is run for epoch times the time period, this is to ensure that the receiver which is running (())(20:55) from the sender process has sufficient amount of time to actually lead this information, so we look at how the epoch is defined which is hash defined to 128 over here as seen over here and the time period is 2 power 24 which means that these loops are run for a total of 2 power 24 time 128 things so this would be long enough to procure the receiver sufficient amount of time to actually detect bits being transmitted. The next thing we do is create memory accesses to various locations.

Now tA underscore 0 and tB underscore 0 correspond to load and store operations to these addresses corresponding to the data being transmitted. Recollect that TB are used in order to control the data and specific odd or even bits being transmitted and tA I used to send 0s and 1s respectively, so essentially this is the communication which is happening by the sender, the sender is making memory accesses to these particular memory addresses. So each time the sender actually make such a memory access, the data if it is not present in the cache would be fetched from a lower cache in this case the L2 and L3 cache and stored in the L1 cache as well as stored in a register pointed to buy this.

Now what happens on the receiver side is that the receiver also runs a similar loop as shown over here but the receiver would also time that particular loop. Now due to the conflicts that arise due to the common cache and the agreed-upon cache sets, the conflicts may actually cause certain memory accesses to be faster and certain memory access to be slower, so the increase time would indicate that there is something being transmitted by the sender on that particular cache set and therefore a miss has occurred the data has to be retrieved from the lower cache or from the de run. Thank you.