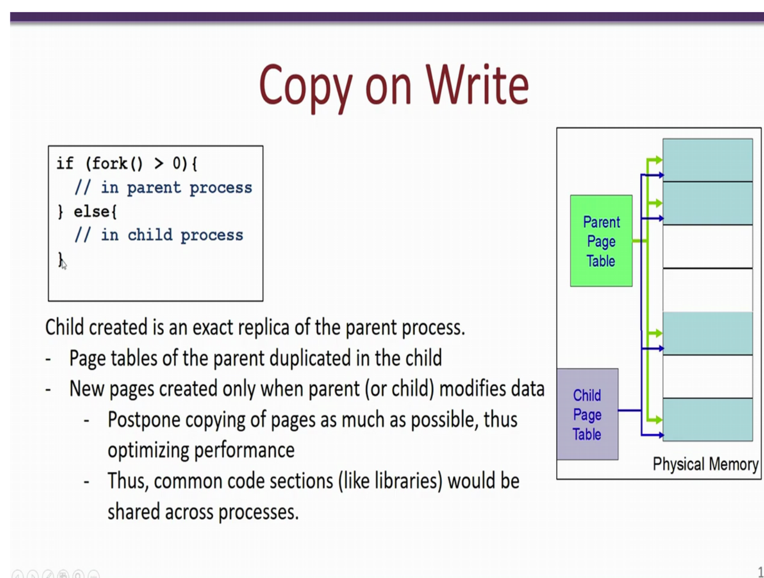
Information Security - 5 - Secure Systems Engineering
Professor Chester Rebeiro
Indian Institute of Technology, Madras
Flush Reload Attacks

Hello and welcome to this lecture in the course in Secure System Engineering. In the previous lecture we got in details to microarchitecture attacks and we looked at cache covered channels. So the aspect about cache covered channels is that we have 2 processes; process A and process B and both are sharing the same cache memory. And process A and process B are colluding with each other so that data and information from one process is transmitted to the other process in spite of all the security aspects offered by the operating system as well as the microprocessor. Now this communication through the covered channel is essentially due to the shared cache memory that is present between the process A and process B. So in this particular lecture we will be looking at another form of cache timing attacks known as the flush + reload.

For these forms of attacks target application such as cryptographic algorithms and have been used to steal secret keys or secret information about the cryptographic algorithm through the cache memory. So to understand the flush and reload attacks we would 1st need to understand something known as copy on Write would and this would take us back to the operating system where we would look at the execution of something of the fork system called, so typical fork system called as you must be quite aware of would look something like this.

(Refer Slide Time: 1:58)



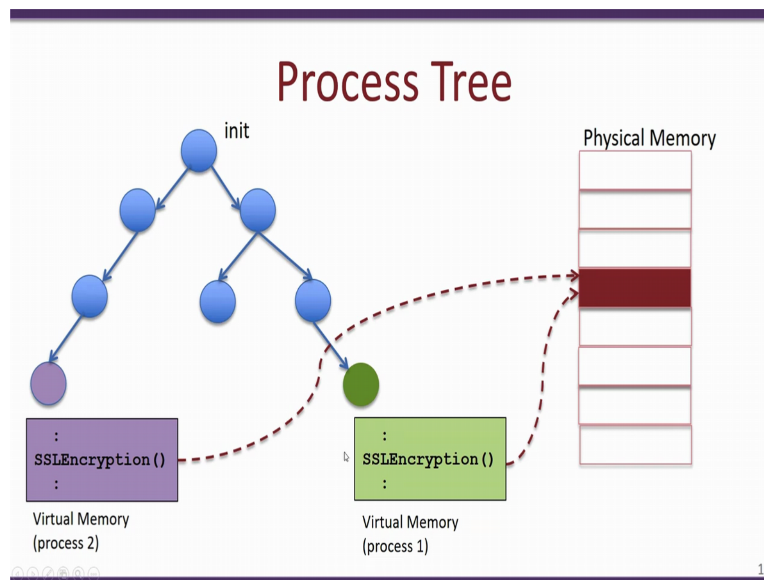
The fork system called which is used by the user processes would invoke the operating system and then the operating system would create a child process. Now when the fork system call returns, it could return with different values. In the parent process the fork will return with the P ID of the child process, while in the child process the fork would return with value of 0, so thus in these codes limit the parent process would execute over here in the if part, while the child process which has just been created would execute over here in the "else" apart.

Now, important for us with respect to the flush and reload is what happens in the operating system, so whenever the fork system gets executed, the OS would duplicate the page directory and page table of the parent process so does what we would obtain would look something like this so we would have the parent page directory and page tables and just cloned child page directory and page table. Although virtual addresses for parent process and the child process would get mapped in a very similar way, so this is the physical memory present in the RAM and what we see over here is that the page tables of the parent as well as the child would essentially map to the same regions in the physical memory.

So thus when the fork system calls returns, we have the child process which is exactly duplicate of the parent process. These pages in the RAM between the parent and the child continue to be shared until either the parent or the child modifies some particular data. When this happens, then the operating system gets invoked again and a new page gets created for example, let us say we have one variable which is shared between the parent and the child process and let us say after the child gets created, the child process modifies that particular data when that instruction gets executed it would result in a fault in operating system and a new page corresponding to that modified data gets allocated to the child process in the physical memory.

So the advantage we obtained from this is that a large portion between the parent and the child is shared. So for example, if you consider let us say a very common library like the G let us see which is used to at least stored common C functions such as print F and scan F because of this copy and write which is used the most systems, each and every process in that system would use the same copy of the print F and scan F functions which are present in RAM, so it would not do the case that each process would have a different version of the print F stored in RAM. The advantage to we achieve with this is that a large portion of the RAM gets saved.

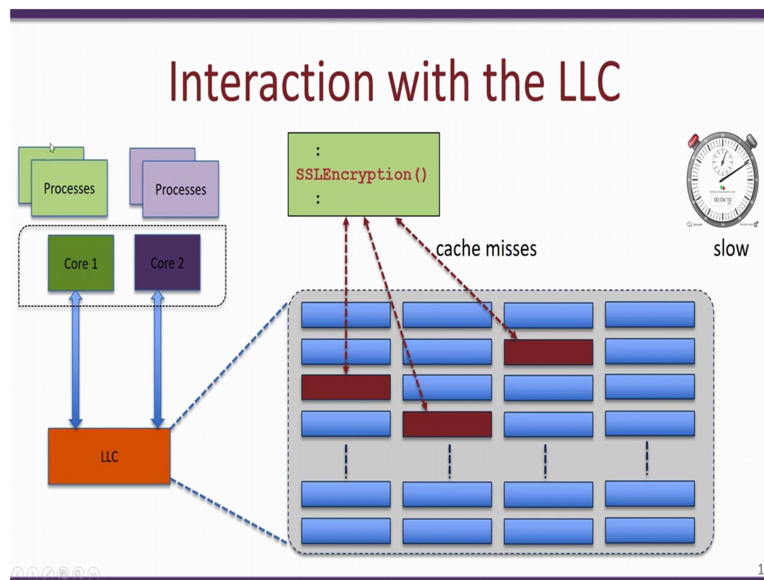
(Refer Slide Time: 5:45)



Now we will also look at something known as the process tree, which is again a very common model for most modern-day operating system. What we do know is that all processes in an operating system are created using the fork system, except for the 1st process. The 1st process in every typical unique system is known as innate and this particular process is created by the operating system. From here this process would then fork various child processes and thus in this way creates an entire process tree. So as we know if this is a process, this is the parent of that process, this is the parent of the 2nd process and so on, so all processes while we go back to the innate process.

Now as a result of the copy on write, the library codes are eventually shared in the physical memory. So for example if we have two processes, this purple process and the green process over here and we used the same library function, which in this case is SSL encryption. Eventually although these 2 processes are at different virtual memory spaces, eventually when they get mapped to physical memory they would eventually use the same copy of this SSL encryption which is stored in the RAM.

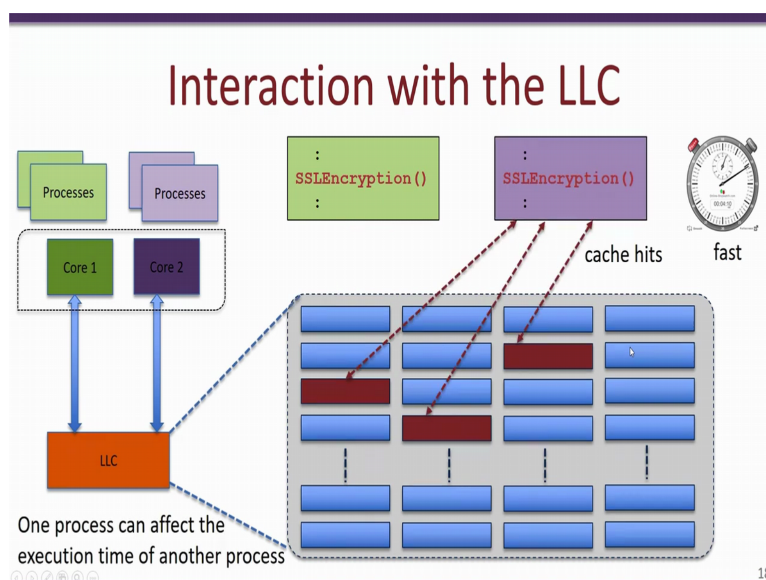
(Refer Slide Time: 7:33)



So let us see how we have these 2 processes; both executing SSL encryption, now we will look at what happens when these 2 processes share the same last level cache memory or the LLC cache memory. So the architecture we are looking at is something like this, we have to cores; core 1 and core 2, the process is executing in core 1 and process two is executed in core 2. And as we have discussed before, both of these processes execute the same SSL encryption, also what we assume is that both core 1 and core 2 share the same last level cache, so this is a grown-up picture of what the cache memory looks like.

Now, when the process one the green process executes the SSL encryption, as we know that there would be a certain number of cache misses that occurs, and as a result there will be parts of the instruction and data corresponding to this SSL encryption, which gets loaded into the cache memory. So in the 1st one we will obtain a large number of cache misses and the cash would then contain some aspects of the SSL encryption. The 1st execution which in this example corresponds to the process one would thus be very slow due to the large number of cache misses that occurs. Now let us see what would happen when the 2nd process executes the exact same function.

(Refer Slide Time: 8:54)



So when this process executes, note that since the data is already present in the last level cache, note that since the instructions are already present in the last level cache due to the prior execution by process 1, the 2nd process would then get a lot of cash hits when SSL encryption is executed, thus the execution time for the 2nd process would be considerably faster than the execution time for the 1st process even though the function is exactly identical. With this we have actually seen how one process could influence the execution time of other process. Now we will look a little more detail and we would see how one process can leak secret information from another process.

(Refer Slide Time: 9:53)

Flush + Reload Attack on LLC

Part of an encryption algorithm

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb          } executed only when e_i = 1
9        x ← x mod m     }
10     endif
11   done
12   return x
13 end
```

clflush Instruction

Takes an address as input.
Flushes that address from all caches
clflush (line 8)

Flush+Reload Attack, Yuval Yarom and Katrina Falkner (https://eprint.iacr.org/2013/448.pdf)

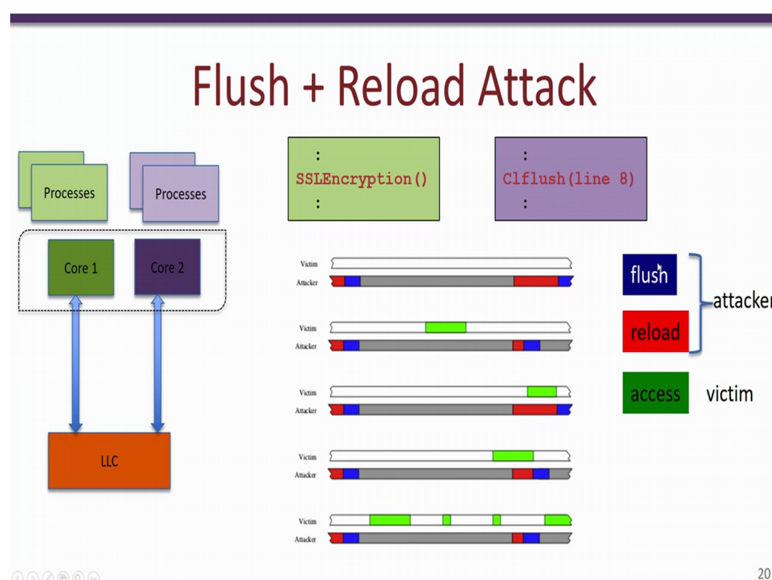
19

Now let us say we have SSL encryption and one function in that SSL encryption looks like this, this function is known as exponent, it takes 3 values b, e and m and this is a very common function used in public key ciphers such as the RAC encryption. So in a typical RAC like encryption, this second argument the key argument is secret.. Now it features go through at a very high level, we just look at this invocation, we see that the e value is used over here, essentially e i would indicate the ith be present in e.

So what is done in this particular function is that there is a for loop from the more significant bit in e down to 0 that is the least significant bit, and in every iteration of this for loop there is a condition check to determine whether the ith bit in e is 1 or 0. If ith is 1, then we do multiplication followed by modular reduction, if the ith bit is 0 then these 2 steps in 8 and 9 are skipped. Now we will see how in other process which shares the same last level cache could use the flush plus reload attack in order to extract one bit of information above the secret E I. So what is important for us is this X 86 supported instruction known as CLF flush, so this CLF flush instruction takes an address as input and flushes that particular address from all the caches present in that system.

So for example, you execute CLF flush and provide the address of the line 8, and what would be guaranteed from here is that the line 8 all the data corresponding to line 8 in this code would be flushed from all the caches present in the system, so the flush and reload attack works as follows.

(Refer Slide Time: 12:19)



Essentially the attacker has 2 phases; there is a flush phase and then there is a reload phase. During the flush phase the attacker executes a line like this, during the flush phase the attacker executes an instruction such as this and ensures that instructions corresponding to line 8 are flushed from the cache memories. During the reload phase, the attacker would access the instructions present in line 8 and therefore, these instructions would then be reloaded into the cache memory thus what is happening is that the attacker is constantly toggling between 2 modes; flush mode where this CLF flush gets executed and data is moved out of the cache, then there is a wait for some time. And then there is a reload mode where the recently flush data is accessed taken so that it is reloaded back into the cache.

And during the 2nd step when the data is reloaded into the cache, the attacker also measures the time taken for that particular instruction to execute. So, note that if there is a cache miss, then the execution time will be high, on the other hand if a cache hit occurs then the execution time during the reload phase would be much lower.

So while the attacker keeps toggling between these 2 steps of flush and reload, we would see what happens as time progresses. So the red part over here shows the reload step, and the blue part over here shows the time for the flush step. So over here for example, there is a flush that has happened and the attacker has used CLF flush to flush out the instructions corresponding to line 8 from the cache, it waits for some time and then the reload step is triggered. And as we would expect since the instructions corresponding to line 8 has been flushed from the cache, we would obtain a cache miss and therefore the execution time to reload would be considerably large.

Now let us assume that during one of the flush and reload phases, there is also the victim that has executed. And the victim has executed this particular for loop and for a particular value of E I which was set to 1, it has executed line 8 and line 9 so as a result, the victims execution would result in a cache miss, instructions corresponding to line 8 and line 9 would get loaded into the cache memory. Now consider the attacker's reload phase, since the instructions corresponding to line 8 are present in the cache the attacker during the reload phase would witness cache hits and therefore the execution time during this particular reload phase would be considerably shorter thus the attacker would be able to infer that the instructions corresponding to line 8 in the process 1 has executed, and as a result he could infer that the E Ith bit happens to be 1.

So there are many other possibilities which are depicted in these 3 figures below; for example, the eviction could occur exactly at that time when attacker's reload phase is occurring or slightly before, or there could be also multiple accesses by the victim before the reload phase executes. Now the important take away from this particular slide is the fact that by monitoring the execution time of the reload, the attacker would be able to identify whether certain instructions were executed by the victim process or not, and from this particular information the attacker would then be able to infer secret information about that victim process.

(Refer Slide Time: 16:37)

Flush + Reload Attack on LLC

Part of an encryption algorithm

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb         } executed only when e_i = 1
9        x ← x mod m    }
10     endif
11   done
12   return x
13 end
```

clflush Instruction

Takes an address as input.
Flushes that address from all caches
clflush (line 8)

Flush+Reload Attack, Yuval Yarom and Katrina Falkner (https://eprint.iacr.org/2013/448.pdf) 19

Example over here, if these instructions have executed then the attacker would infer a value of 1 for that E I bit of key, if these instructions have not been executed then the attacker will infer that the E I bit is 0.

(Refer Slide Time: 16:56)

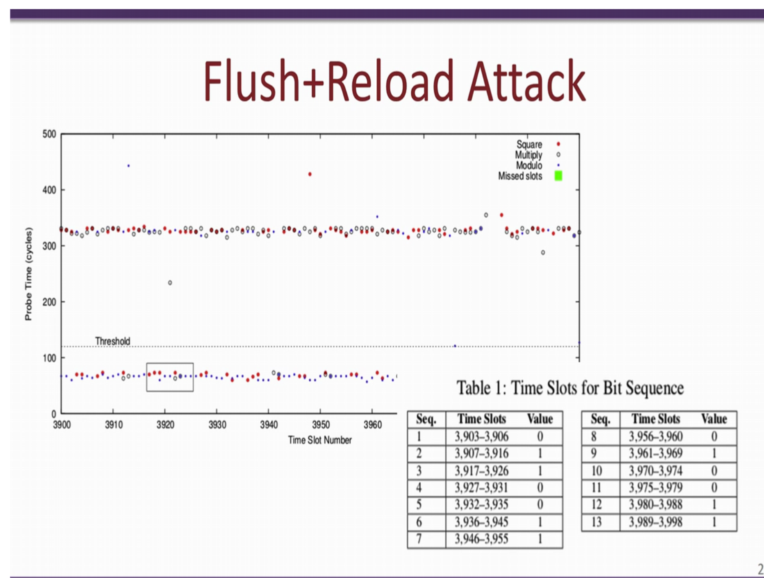


Table 1: Time Slots for Bit Sequence

Seq.	Time Slots	Value
1	3,903–3,906	0
2	3,907–3,916	1
3	3,917–3,926	1
4	3,927–3,931	0
5	3,932–3,935	0
6	3,936–3,945	1
7	3,946–3,955	1

Seq.	Time Slots	Value
8	3,956–3,960	0
9	3,961–3,969	1
10	3,970–3,974	0
11	3,975–3,979	0
12	3,980–3,988	1
13	3,989–3,998	1

This particular slide show the reload time as the vitamins executing, we can clearly see that there is significant difference when there is a cache hit which is corresponding to these areas over here when there is a cache miss. So from this the attacker can infer what instructions were actually executed by the victim process. So how we would actually counter this flush and reload attack?

(Refer Slide Time: 17:23)

Countermeasures

- Do not use copy-on-write
  - Implemented by cloud providers
- Permission checks for clflush
  - Do we need clflush?
- Non-inclusive cache memories
  - AMD
  - Intel i9 versions
- Fuzzing Clocks
- Software Diversification
  - Permute location of objects in memory (statically and dynamically)

22

The most obvious way to actually prevent such an attack is to not to use copy and write principle which is implemented by most operating systems, however this is easier said than done because copy and write is a very efficient way to utilise the fixed amount of RAM that is present in the system.. However, as we see even though there are security wonderbilities

with respect to copy and write, it is still a very preferred way for doing things. So however, very recently cloud providers have prevented copy on write from one virtual machine to another virtual machine.

Another way of preventing this particular attack is to redesign the instruction CLF flush and make it a privilege instruction. As such, a typical application do not use a function like CLF flush, such an instruction is typically used to achieve memory consistency and which is not typically needed by many applications. One possible countermeasure is to for example, is to make CLF flush a privilege instruction and only super users or root users would be able to invoke CLF flush. Another alternative is to make CLF flush accessible only through a system call. A 3rd countermeasure which is now adopted by Intel and has always been followed by AMD is to create cache memories which are nonintrusive, we will not go further into these things.

So modern Intel machine especially from Intel I9 and so on have non-inclusive caches which can prevent the flush and reload attacks, other solutions are by fuzzing clocks present in the system, typical clock that is used in such schemes the RDTSC timestamp counter. Another countermeasure is by fuzzing clocks so that the attacker will not be able to make precise timing measurement. Another alternative is by software diversification where you permute the locations of objects in memory so that by monitoring the hits and misses or by obtaining the hits and misses, the attacker will not be easily able to identify what instruction was actually being executed at that location, thank you.