6 Exploiting Out-of-Order-Execution; or, Processor Side Channels to Enable Cross VM Code Execution

by Sophia D'Antoine

In which Sophia uses the `MFENCE` *instruction on virtual machines, just as Joshua used trumpets on the walls of Jericho. —PML*

At REcon 2015, I demonstrated a new hardware side channel that targeted co-located virtual machines in the cloud. This attack exploited the CPU's pipeline as opposed to cache tiers, which are often used in side channel attacks. When designing or looking for hardware-based side channels—specifically in the cloud, I analyzed a few universal properties that define the "right" kind of vulnerable system as well as unique ones tailored to the hardware medium.

The relevance of these types of attacks will only increase—especially attacks that target the vulnerabilities inherent to systems that share hardware resources, such as in cloud platforms.

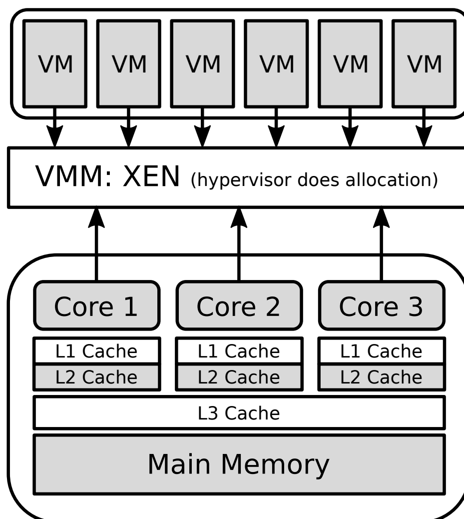

Figure 1: Virtualization of physical resources

6.1 What is a Side Channel Attack?

Basically a side channel is a way for any meaningful information to be leaked from the environment running the target application, or in this case the victim virtual machine (as in Figure 6). In this case, a process (the attacker) must be able to repeatedly record this environment "artifact" from inside one virtual machine.

In the cloud, this environment is the shared physical resources on the service used by the virtual machines. The hypervisor dynamically partitions each physical resource—which is then seen by a single virtual machine as its own private resource. The side channel model in Figure 6.1 illustrates this.

Knowing this, the attacker can affect that resource partition in a recordable way, such as by flushing a line in the cache tier, waiting until the victim process uses it for an operation, then requesting that address again—recording what values are now there.

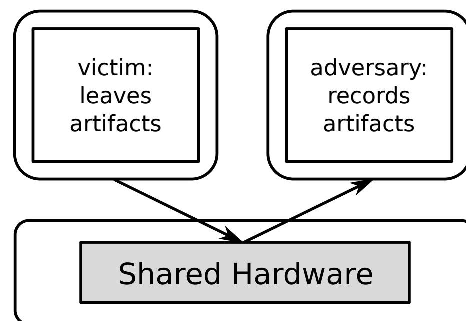

Figure 2: Side channel model

6.2 What Good is a Side Channel Attack?

Great! So we can record things from our victim's environment—but now what? Of course, some kinds of information are better than others; here is an overview of the different kinds of attacks people have considered, depending on what the victim's process is doing.

Crypto key theft. Crypto keys are great, private crypto keys are even better. Using this hardware side channel, it's possible to leak the bytes of the private key used by a co-located process. In one scenario, two virtual machines are allocated the same space in the L3 cache at different times. The attacker flushes a certain cache address, waits for the

victim to use that address, then queries it again—recording the new values that are there.[1]

Process monitoring. What applications is the victim running? It will be possible for find out when you record enough of the target's behavior, i.e., its CPU or pipeline usage or values stored in memory. Then a mapping between the recording to a specific running process could be constructed—up to some varied degree of certainty. Warning, this does rely on at least a rudimentary knowledge of machine learning.

Environment keying. This attack is handy for proving co-location. Using the environment recordings taken off of a specific hardware resource, you can also uniquely identify one server from another in the cloud. This is useful to prove that two virtual machines you control are co-resident on the same physical server. Alternatively, if you know the behavior signature of a server your target is on, you can repeatedly create virtual machines in the targeted cloud, recording the behavior on each system until you find a match.[2]

Broadcast signal. This attack is a nifty way of receiving messages without access to the Internet. If a colluding process is purposefully generating behavior on a pre-arranged hardware resource, such as purposefully filling a cache line with 0's and 1's, the attacker (your process) can record this behavior in the same way it would record a victim's behavior. You then can translate the recorded values into pre-agreed messages. Recording from different hardware mediums results in a channel with different bandwidths.[3]

6.3 The Cache is Easy; the Pipeline is Harder

Now all of the above examples used the cache to record the environment shared by both victim and attacker processes. It is the most widely used resource in both literature and practice for constructing side channels, as well as the easiest one to record artifacts from. Basically, everyone loves cache.

However, the cache isn't the only shared resource. Co-located virtual machines also share the CPU execution pipeline, as illustrated in Figure 3. In order to use the CPU pipeline, we must be able to record a value from it. Unfortunately, there is no easy way for any process to query the state of the pipeline over time—it is like a virtual black-box.

The only thing a process can know is the instruction set order it gives to be executed on the pipeline and the result the pipeline returns. This is the information source we will mine for a number of effects and artifacts, as follows.

Out of order execution: a pipeline's artifact. We can exploit this pipeline optimization as a means to record the state of the pipeline. The known input instruction order will result in two different return values—one is the expected result(s), the other is the result if the pipeline executes them out-of-order.

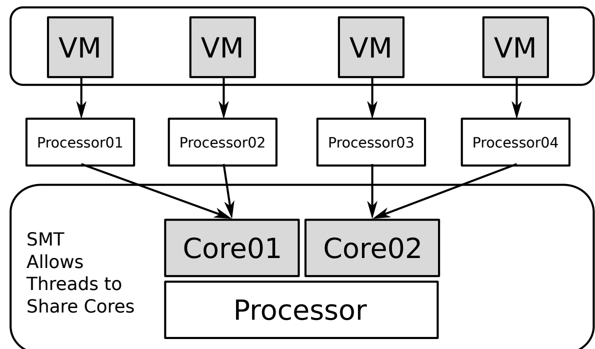

Figure 3: Foreign processes can share the same pipeline

Strong memory ordering. Our target, cloud processors, can be assumed to be x86/64 architecture—implying a usually strongly-ordered memory model.[4] This is important, because the pipeline will optimize the execution of instructions, but will attempt to maintain the right order of stores to memory and loads from memory.

However, the stores and loads from different threads may be reordered by out-of-order-execution. Now, this reordering is observable if we're clever enough.

Recording instruction reorder (or, how to be clever). In order for the attacker to record these reordering artifacts from the pipeline, we must record two things for each of our two threads: *input instruction order* and *return value*.

Additionally, the instructions in each thread must contain a `STORE` to memory and a `LOAD` from memory. The `LOAD` from memory must reference the location stored to by the opposite thread. This setup ensures the possibility for the four cases illustrated in Figure 4. The last is the artifact we record; doing so several thousand times gives us averages over time.

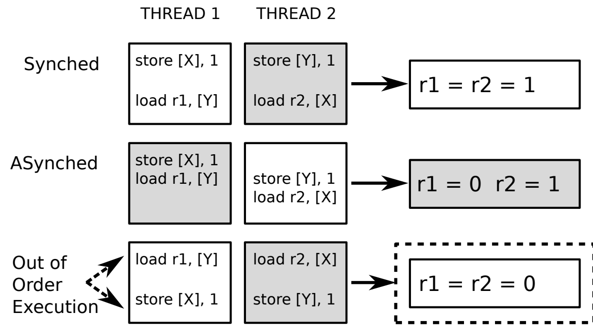

Figure 4: The attacker can record when its instructions are reordered

Sending a message. To make our attacks more interesting, we want to be able to force the amount of recorded out-of-order-executions. This ability is useful for other attacks, such as constructing covert communication channels.

In order to do this, we need to alter how the pipeline optimization works—by increasing the probability that it either will or will not reorder our two threads. The easiest is to enforce a strong memory order and guarantee that the attacker will receive fewer out-of-order-executions. This is where memory barriers come in.

Memory barriers. In the x86 instruction set, there are specific barrier instructions that stop the processor from reordering the four possible combinations of `STORE`'s and `LOAD`'s. What we're interested in is forcing a strong order when the processor encounters an instruction set with a `STORE` followed by a `LOAD`. The `MFENCE` instruction does exactly this.

By getting the colluding process to inject these memory barriers into the pipeline, the attacker ensures that the instructions will not be reordered, forcing a noticeable decrease in the recorded averages. Doing this in distinct time frames allows us to send a binary message, as shown in Figure 5. More details are available in my thesis.[19]

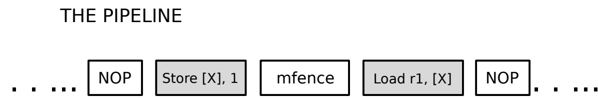

Figure 5: `MFENCE` ensures the strong memory order on pipeline

The takeaway is that—even with virtualization separating your virtual machine from the hundreds of other alien virtual machines!—the pipeline can't distinguish your process's instructions from all the other ones, and we can use that to our advantage.

References

[1] *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*, Yuval Yarom, Katrina Falkner, USENIX Security 2014

[2] *Cross-Tenant Side-Channel Attacks in PaaS Clouds* Yinqian Zhang, Ari Juels, Michael K. Reiter, Thomas Ristenpart ACM CCS 2014

[3] *Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud*, Zhenyu Wu, Zhang Xu, Haining Wang USENIX Security 2012

[4] *Weak vs. Strong Memory Models*, Preshing on Programming, `http://preshing.com/20120930/weak-vs-strong-memory-models/`


```
1  '''
2
3  TRANSMITTER
4  sophia.re
5  07/06/15
6
7  '''
8
9  from time import time, sleep
10 import os
11
12 # takes a binary string as input
```

[19] `unzip pocorgtfo09.pdf crossvm.pdf`

```
13 def send (Message, roundLength):
        for x in Message:
15              # Run a single busy loop to represent a 0
                if( x == '0'):
17                      print('sending', x)
                        # change the time of this busy loop to match receiver round length
19                      start_time = time()
                        end_time = time() + roundLength #this number is loop time in seconds
21                      while( start_time < end_time):
                                start_time = time() #do nothing
23              else:
                        # send a 'hi' bit in a given time frame
25                      # by reducing the received out of order executions
                        # this is done using the sender exe
27                      print('sending', x)
                        start_time = time()
29                      end_time = time() + roundLength
                        while( start_time < end_time):
31                              os.system("C:\\CPUSender.exe")
                                # do nothing until sending c process terminates
33                              start_time = time()

35
   def main():
37      # measured receiver time frame length in seconds - (for one bit)
        roundLength = 1.08
39      message = ''

41      # enter binary string
        while( message != 'exit'):
43              message = raw_input('Enter Binary String: ')
                start_t = time()
45              if( message != 'exit'):
                        send(message,roundLength)
47              print "\nTotal execution time: "
                print time() - start_t
49
   if __name__ == "__main__":
51      main()
```

```
1  '''

3  RECEIVER
   sophia.re
5  07/06/15

7  '''

9  from time import time,sleep
   import os
11 import sys, subprocess
   import msvcrt as m
13 import matplotlib
   import matplotlib.pyplot as plt
15
   def main():
17
     while True:
19     start_time = time()
       end_time = time() + 12
21     print "Receiving Bits in Words (8 bit blocks)....\n"

23     # records out of order executions and writes averages to file
```

```python
    p = subprocess.Popen("C:/Receiver.exe "+"1 "*8)
    while start_time < end_time:
        start_time = time()
    print time()

    # wait because of system latency
    p = subprocess.Popen("C:/nop.exe")
    p = subprocess.Popen("C:/nop.exe")

    # read all recorded out of order executions from file
    f = open("C:/Python27/BackupCheck.txt")
    txt = f.readlines()
    f.close()
    txt = txt[0]
    print "Received Bits\n"
    print txt

    # trigger a picture to appear
    bits = txt.split(":")
    if "11" in bits[0]:
        print "\n [+] trigger detected "
        exe = "C:/Users/root/Downloads/JPEGView_1_0_29/JPEGView.exe"
        args = '"C:/pics"'
        p = subprocess.call([exe,args])
        sys.exit(0)
        quit()
    else:
        print "\n [+] trigger not detected"


    # plot received out of order executions to view step signal
    print "\n\nEnter to Plot...."

    p.kill()
    m.getch()

    # plot recorded OoOE step signal to png file
    with open("BackupCheck2.txt") as f:
        data = f.read()
    data = data.split("\n")

    y = [float(x) for x in data[0].split(' ')[:-1]]
    x = list(xrange(len(y)))
    print "There are ", len(y), " elements to plot."

    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.set_title("Plot Received OoOE")
    ax1.set_xlabel("iterations")
    ax1.set_ylabel("out-of-order-execution averages")
    ax1.fill_between(x,y,color='yellow')
    ax1.plot(x,y, marker='.',lw=1,label='the data', alpha=0.3)
    leg = ax1.legend()

    plt.savefig('plot.png', bbox_inches='tight')

    # repeat
    print "\n\nEnter to Continue...."
    m.getch()

if __name__ == "__main__":
    main()
```