DRAMA : DRAM Addressing Attack

Sahil Shah, 160050005
Preey Shah, 160050008
Kunal Goyal, 160050026
Aman Bansal, 160050028,
Kanak Agarwal, 150050016

November 2018

Contents

1 Introduction

For a long time, side channel attacks have been the bane of computing architects. Here, we explore a cross-CPU side channel attack that is particularly interesting and has wide applications. This attack exploits DRAM addressing [1] and the presence of buffers in each bank of the DRAM module. We observe the time in accessing memory and deduce accordingly whether the memory accessed by the other process was in the same bank in DRAM module. For this, we require a mapping of physical addresses to a bank. A reverse engineering method for the same has also been described in the paper [1]. Our project involved significant amounts of both studying the paper as well as its publicly available implementations and then building upon those implementations.

In this project, we have reverse engineered and verified the mapping from the physical address space to actual DRAM banks for the 7^{th} generation of Intel's i7 processor and use this to build a covert channel of communication between 2 different processes running on the same machine. This paper provided a code base, but since it was largely undocumented, we essentially build on the ideas expressed in the paper ourselves, using a few functions from the code base.

Apart from these, we also studied Flush-Reload cache-based side channel attack and provided a demonstration of the same using existing ideas.

2 Motivation

In the past, cache-based architectural attacks have been studied extensively and several techniques like Prime Probe [2] and Flush and Reload [3] have been developed. The scope of such attacks was limited and they could not work on processes running on different CPUs. Rowhammer attack [4] was the first attack to use DRAM modules. The paper [1], popularly called DRAMA (short for DRAM Addressing Attacks), on which our project is based on was published only in 2016 and exploits DRAM based addressing for cross-CPU attacks. Further, in recent years there has been a rapid increase in cloud computing and the use of hypervisors to ensure isolation of virtual machines. This paper provides a way to build a covert channel in such a cloud computing environment and can form the basis of further research. It also provides methods to reverse-engineer the DRAM mapping which we describe in the next section. This mapping can then be used to speed up rowhammer attacks as well.

Besides this, we also implemented the shared cache-based Flush-and-Reload attack in order to spy on a program which takes input from the user and predict certain properties of the input using our probe. We can see the difference between the two attacks in terms of the certainty of their outputs. This implementation helped us clearly understand the difference between the two attacks and their respective advantages and disadvantages. Though we principally focus on the DRAMA attack, we explain the differences in the end. Reading and implementing both attacks helped give us an insight into the development of different side-channel attacks.

3 Contributions

3.1 Individual Contributions of the Team Members

To get started, all of us read up on the DRAM Addressing Attacks from various sources including an online presentation by the authors, a Masters' Thesis of Michael Schwarz (DRAMA: Exploiting DRAM Buffers for Fun and Profit) and of course, the original DRAMA paper [1]. The link of the masters thesis is https://www.blackhat.com/docs/eu-16/materials/eu-16-Schwarz-How-Your-DRAM-Becomes-A-Security-Problem-wp.pdfhere (we did not get the dblp citation). Once we had an idea about the background, we split the work in the following manner. Preey Shah worked on the Flush and Reload Attack and its implementation. Sahil Shah and Kanak Agrawal worked on reverse-engineering the mappings for 7^{th} Generation i7 and verifying them while Aman Bansal and Kunal Goyal worked on using the mappings to build a covert communication channel. However, everyone has knowledge of the different parts and gave inputs for the same. The work division was not very strict and the above division is broad.

3.2 Contribution towards Future Work

The mapping found using the reverse engineering method can be used to make the Rowhammer attack faster. Having knowledge of the mapping helps us target a narrower range of addresses.
Even the Flush-Reload attack can be improved upon by identifying a better threshold for cache hit time. Knowledge of the time taken on a row buffer hit can be used in setting the threshold. Both the above attacks can be implemented incorporating the above factors. The DRAM addressing can be used to implement a cover channel between a machine and its VM. The challenges faced have been mentioned above. Further work can be done to find the DRAM mapping for other processors and to exploit the DRAM addressing to design a side-channel attack.

4 Background

4.1 DRAM organization

- DRAM stands for Dynamic Random-Access Memory. The main memory of computers is made up of DRAM cells. Modern DRAM is organized in a hierarchy of channels, DIMMs, ranks, and banks.
- A system has one or more channels. Channels are physical links between the DRAM modules and the memory controller.
- Multiple DIMMs (Dual Inline Memory Modules) are connected to each channel. A DIMM has one or two ranks.
- Each rank has 8 or 16 banks corresponding to DDR3 and DDR4 DRAM respectively. In the case of DDR4, banks are additionally grouped into bank groups. Banks finally contain the actual memory arrays which are organized in around 215 rows and around 210 columns.
- Two addresses can only be physically adjacent in the DRAM chip if they are in the same channel, DIMM, rank and bank.
- The mapping from the physical address to the channel, DIMM, rank and bank, is completely dependent on the processor. However for the mapping to be uniform among all the channels, DIMMs, ranks and banks, the mapping is generally a linear function using an XOR of address bits. We assume that the addressing functions are also linear for newer CPU models. The manufacturers, however, do not disclose the mapping of its processors.

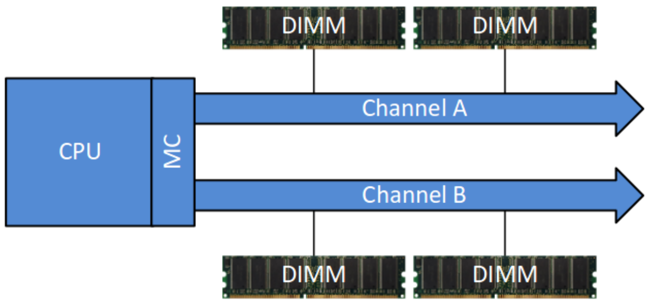


Figure 1: Hierarchy of DRAM

4.2 The Row Buffer

Each bank contain multiple rows but to access memory, the individual rows cannot be accessed directly. Between the DRAM cells and the memory bus, there is a row buffer which is unique for every bank. It stores an entire row and it basically acts like a directly mapped cache. Requests to addresses in the currently active row are served directly from this buffer. If a different row needs to be accessed, then the data of the row is first copied to the row buffer and then the request is serviced. Naturally, on such a conflict, it takes more time to access the data which will be the basis of the DRAMA attack.

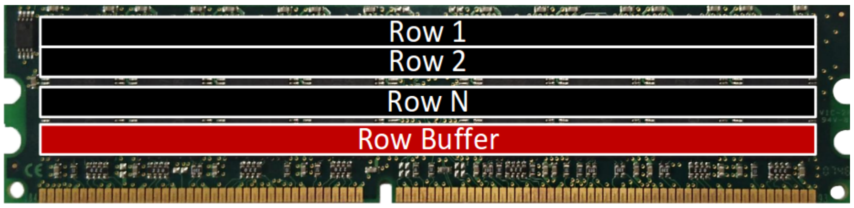


Figure 2: A Bank

4.3 Exploiting the Row Buffer

Pessl, et al. describe in [1] a method to exploit the row buffer to both, reverse engineer the mapping used by the processor as well as to build a covert communication channel. In this section, we explain this exploitation.

We alternately access elements from the DRAM at 2 different physical addresses and measure the average time taken over several such accesses. We ensure that the element is always fetched from main memory by flushing it from the cache before every access.

Now, if the two addresses happen to be mapped to the same bank in physical memory, then this average time will be high, because every access now requires us to first copy the row into the row buffer. The next access (to the other address in the same bank) will replace the row from the row buffer and so on, since the rows corresponding to the 2 elements are competing for the same row buffer. On the other hand, if we access addresses that belong to 2 different banks, the average access time will be much lesser because after an initial access to both the elements, the corresponding rows will get stored in two different row buffers and hence, each access involves simply reading the row off the row buffer (and not copying the row into the row buffer as well).

We describe how we use this to reverse engineer the mapping in Section 5 and how to use it to build a covert channel in Section 6.

4.4 Mapping Physical Addresses to DRAM rows

The actual row of the DRAM in which an element is present is determined by the physical address of the element. However, the mapping used is not a simple one - probably because the processor would like to place spatially related data elements in different banks so that repeated accesses to them would be efficient. Moreover, this mapping is determined by the processor and varies not only from processor to processor but also across generations. The mapping for Intel processors is not made public by Intel and hence, the need to reverse engineer these mappings arise. Pessl, et al. provide the mappings for a few processors, but that for the 7^{th} generation of Intel's i7 processor was not publicly known and hence, we are finding these mappings as a part of the project. In Section 5 we describe how we achieve this goal.

It is worth noting here that though the functions are not known, in most cases they have found to consists of simple linear functions, which means that a bit in the output can be expressed as the XOR of one or more bits of the physical address. Here, output of the function refers to a set of bits that uniquely identify a row in the DRAM. However, since we are interested only in the bank and not in the exact row to which a physical address maps, we expect the function to provide $log_2(n)$ bits where n is the total number of banks that exist in the DRAM. Each of the bits can further be given a semantic meaning -

one of the bits may be used to identify the channel, another to identify the rank in the channel and two to identify the bank within the channel and so on. However, determining which bit corresponds to what is beyond the scope of this project and would probably require the use of physical methods which are also described by Pessl, et al. in [1]. This method, requires the measurement of voltages in the bus that connects the processor and the DRAM and is much faster and more accurate than the software based reverse engineering. However, it requires us to physically dig into the machine and the DRAM and is hence very inconvenient.

5 Reverse Engineering the Mapping for Intel i7 7^{th} Gen

In this section, we describe how we reverse-engineered the previously unknown mapping from physical addresses to DRAM row for the i7 7^{th} generation processor. We also describe some interesting implementation details, the challenges we faced and how we verify these mappings.

5.1 The Algorithm

We first provide a high-level understanding of the method we employ to reverse-engineer the mapping.

- First, the program allocates a large chunk (usually over 70% of the entire DRAM) by declaring a very large array and initializing it (in order to prevent lazy allocation from not actually allocating physical memory for the array).
- Next, it uses the pagemap of the process to find the page tables and hence determine the virtual address to physical address translation.
- The program now creates a set $\mathcal{S}$ of a large number of random addresses (both virtual and physical). Now, we select any one address from this set $\mathcal{S}$ at random and call it A.
- Next, in order to partition the set into subsets based on which bank the addresses belong to, we compute the average time for alternately accessing A with every other element of $\mathcal{S}$ a large number of times.
- This gives us a histogram like the one in Figure 3(a), where we have the access time (in terms of number of clock cycles) on the x-axis and the fraction of accesses on the y-axis. One can see 2 clear peaks in such a histogram and these correspond to accesses from the same set and from different sets. Using the argument described in section 4.3, we deduce that the peak corresponding to lower access times are due to physical addresses from $\mathcal{S}$ that do not lie in the same set as A, while the peak corresponding to larger access times is caused by the physical addresses that lie in the same bank as A. The peak for elements belonging to the same set make up a significantly smaller fraction of total accesses, and this should be expected because we would expect only $\frac{1}{n}$ of the random addresses to lie in the same bank as A where n is the total number of banks in the DRAM.
- We manually tweak a few parameters and can then algorithmically partition $\mathcal{S}$ based on this histogram to get a set $\mathcal{S}^1$ of addresses which belong to the same bank as A. We now, update $\mathcal{S}$ to $\mathcal{S}\backslash(\{A\} \cup \mathcal{S}^1)$ and continue this procedure until we get the required number of partitions. The target number of partitions (denoted by ns) is supposed to be provided by the user of the program and computed as the the product of the number of channels, DIMMs, ranks and banks.

$$ns = DIMMs * channels * ranks * banks$$

- At the end of this, what we obtain is a partition of the set $\mathcal{S}$ into ns sets such that all the elements in a subset belong to the same bank. We then use this information to obtain a bunch of likely mapping functions.

- The assumption in calculating the function is that it is a linear function using an XOR of physical address bits. If the number of sets is a power of two, the XOR functions give a uniform distribution. Now, we can represent the addresses and their set as a linear equation system. Instead of systematically solving the equations, we generate every possible function and verify whether it yields the same result for each address in the set. Since the the maximum number of coefficients is the number of bits of an physical address, the search space is not very large. Moreover, the lower 6 bits of the address are used for indexing in the row, so we can directly equate their coefficients to zero.
- So, we iterate over all the bitmasks. Then, for each bitmask, we apply it to every address in all the sets. If the XOR of the bits of the bitmask is same for all the addresses of each set individually, then we store that function as a possible solution. We now have several candidate functions saved. From them we choose those which distribute the *ns* sets approximately equally and have lower number of set bits. To achieve this, we remove all functions that are linear combinations of other functions through the method Gauss-Jordan elimination.
- After removing the linearly dependent functions, we are left with a smaller number of functions. However, because of the random selection of addresses, we get some false functions in our result. This is possible if we did not get addresses in every set while randomly selecting a pool of addresses. The reason for this is that we either guessed too small a number of sets or we did not get enough addresses in a particular set. The former is possible because the number of sets is not easily available information adn the latter is likely in case there are some other programs running at the same time which utilize a significant amount of the DRAM. In both cases, we are missing some sets for which the computed functions might be wrong. To eliminate these spurious functions, we can run the measurement and identification phase multiple times. The intersection of the functions from all runs would result in only correct functions with a high probability.

5.2 Implementation Details

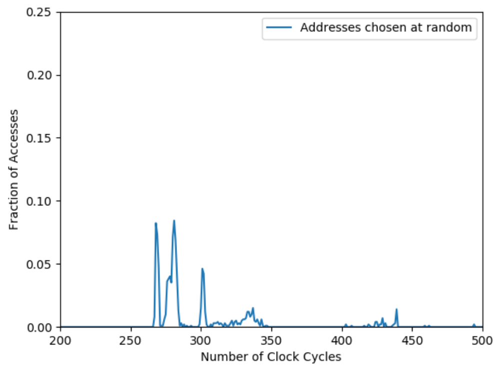
To find the translation from virtual address space to the physical address space, we make use of the file /proc/self/pagemap which gives us information about the page table of the process, which can then be used to determine the physical address corresponding to any virtual address.

The process used to measure the time is worth discussing and the most interesting points follow:

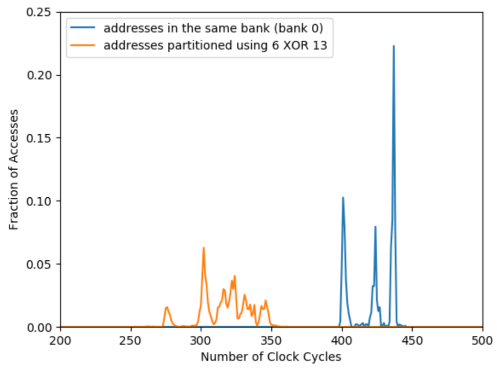
- The memory controller can reorder memory access for performance reasons. To stop it from doing it, we access random address in between to make it hard for memory controller to optimize via reordering. This may add a little noise to our calculations but on taking average over multiple attempts we get results good enough for the purpose of reverse engineering the mapping.
- Values are cached not only in row buffers but also by the CPU cache. Every access to a memory location gets cached, and further accesses are served from the cache. Hence it is important to clear the CPU cache after every memory access because we don't want to calculate the CPU cache hit time but time to access from row buffer in DRAM. For this, clflush assembly instruction is used.
- CPU can re-order instructions. We need to prevent this out-of-order execution to guarantee a noise-free correct result. For this purpose, we use the serializing instruction cpuid.
- Since the program runs in user space, scheduler can interrupt at any time and this can add noise to our calculations. We can try to avoid this by yielding several times before our timing measurements.
- Since we are accessing same memory addresses many times without modifying the data at that address, compiler can optimize and just access once knowing that further accesses will give same value. We don't want this to happen. We prevent this by declaring all our pointers and inline assembly instructions to be volatile. This tells the compiler that the value of the variable can change, even though there is no code that changes the variable.

5.3 Challenges Faced

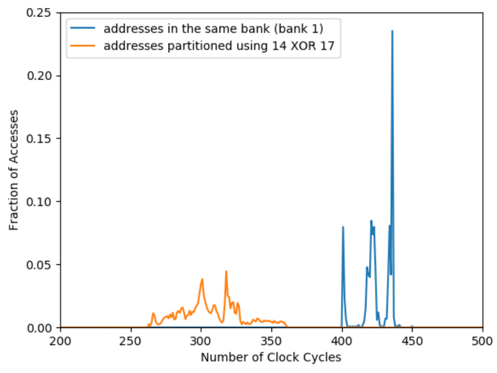
- The biggest challenge we faced was that we did not know how many sets our DRAM was divided into and we could not find any way of determining this through software-based approaches. The uncertainty persisted until we verified the functions we guessed.
- The code that was available online outputs functions that are likely (not necessarily the correct ones), along with their probability which was rarely 100%.
- If there is any other process running on the system at the same time, the output is very noisy and can often lead to incorrect contributions. We learnt the hard way, that it was essential to kill all other background processes before running our code.
- There is a lot of non-determinism as the code gives different outputs when run at different times as well as when run on different cores of the CPU.



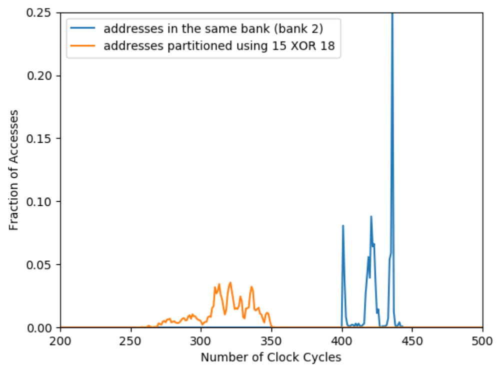
(a)



(b)



(c)



(d)

Figure 3: (a) Graph of the time between the access of two random addresses. We can observe one larger peak at lower access time and one smaller peak with higher access time. The second peak corresponds to the two addresses belonging to the same bank.
(b), (c), (d) Blue graph represents the access times between two addresses of the same bank. Orange graph represents the access times with two addresses corresponding to different values of functions that we calculated. For eg in (b), the function is 6th bit XOR 13th bit.

5.4 Results

The code available online provides as output a large number of probable functions with a probability attached to each. The problem we faced was two-fold. First, we did not know the number of banks in the DRAM and hence how many functions to actually look for. Secondly, on running the program the second time, the output was always different. To tackle this problem, we ran the code several times on different cores of the CPU and finally guestimated the following functions:

$$h_0 = p_6 \oplus p_{13}$$

$$h_1 = p_{14} \oplus p_{17}$$

$$h_2 = p_{15} \oplus p_{18}$$

$$h_3 = p_{16} \oplus p_{19}$$

where h_i represents the i^{th} bit used to determine the bank and p_i represents the i^{th} bit of the physical address. We provide a verification of these functions in the following section as well, since the procedure to obtain them as described in the paper leaves a lot of ambiguity.

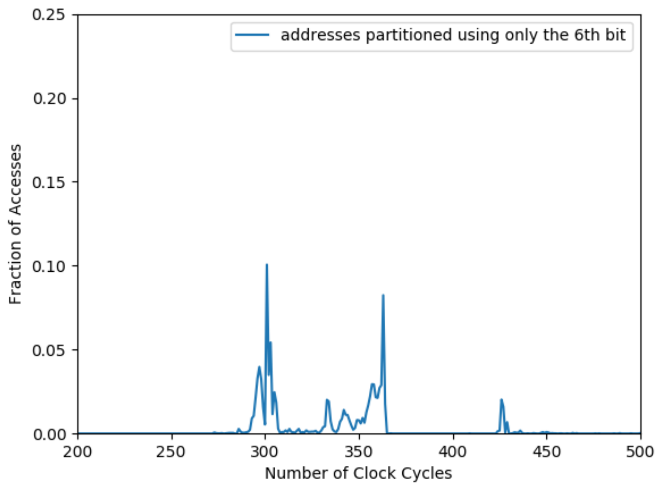


Figure 4

5.5 Verification of the Mapping

Due to the uncertainty in the number of sets and the non-determinism in the output of the reverse-engineering script available online, we came up with our own methods to verify the mappings we deduce. The method proceeds as follows.

First, we select a base address and measure the access time when alternating with elements chosen at random, that is, from any set. The histogram obtained for this is shown in Figure 3(a). In this histogram, we can see two clear peaks - the one that occurs at smaller values of access time correspond to the addresses which lie in a different bank compared to the base address. The other peak, which occurs at larger values of access time corresponds to addresses that lie in the same bank. We use this to infer a threshold that differentiates accesses that belong to the same set and those that belong to different ones.

We now proceed with the verification in the following manner: to ensure that the functions we have chosen partition the physical address space into subsets such that each subset definitely lies in one bank (but not necessarily occupying the entire branch) we plot histograms when accesses are in the same bank as determined by all the four functions. These plots can be seen as the blue curves in plots (b), (c) and (d) of Figure 3. Since the histogram of these accesses lie completely above 395 clock cycles, we can logically deduce that all accesses are to elements that are in the same set. For the sake of brevity, we consider plots corresponding to the first 3 banks and include the remaining 13 in the appendix.

Now, we need to show that each function that we have come up with is actually a partitioning function. To do so, we consider one function at a time and consider accesses to elements that give different values for the function in consideration. We now need to prove that for every function, the two elements that we consider are indeed in different banks, that is, different values on any one function ensures that the addresses are in different banks. This is evident from the histograms that are drawn in orange in plots (b), (c) and (d) of Figure 3. There is no access time that takes more than 395 clock cycles in this case, confirming our hypothesis.

Let us consider the possibility that the number of sets was 32 and that p_6 and p_{13} were functions in themselves. For this, we plotted the histogram obtained by accessing elements that have different values of the 6^{th} bit of the physical address, as shown in Figure 4. From the histogram it is clear that we have access times both above and below the threshold of 395 cycles and hence, partitioning using this method is yielding addresses in the same bank as well.

6 Building a Covert Channel

6.1 The Algorithm

- We will now describe the algorithm which we used to build a covert channel between two processes. We use the physical address to DRAM bank mappings of the processor obtained in the previous section to identify whether two addresses belong to the same bank or not. If any of the four functions give a different value then the addresses belong to different banks. First of all we generate two addresses, one in both the processes, such that they belong to the same bank but not to the same row.
- Both the receiving process (referred as reciever hereafter) and the sending process (referred as sender hereafter) agree upon a time interval. The receiver keeps accessing his address continuously during that time interval and then calculates the average time it took to access that address using the same method as used in reverse engineering the mappings. The sender transfers the messages in bits. If it has to transfer a '1' then it keeps accessing his address continuously during the time interval (after flushing the cache) otherwise it just sits idles.
- If the sender transmitted '1' then the average time is going to be higher as compared to the average time when the sender transmitted '0' because the number of row buffer misses will increase. This is due to the continuous accesses by the sender which competes for the same row buffer cache.
- The receiver compares the average time with the threshold to determine if the bit transferred was 0 or 1.

6.2 Implementation Details

- The first major obstacle was to synchronize the sender and the receiver. We used the processor's time-stamp counter (TSC) to synchronize both the processes. We defined a *syncTime* variable which stores the length of the time-interval for transmission of one bit. Both the sender and receiver agree upon this value.
- Next part was to generate two addresses, one in sender and one in receiver, such that they belong to the same bank but not to the same row. For this, both the processes create a large array and find 100 indices whose addresses belong to to a particular bank (both process must agree upon a this bank). Then both process pick any index out of these indices and choose that as the address they are going to access.

- The receiver starts recording the average access time for each time interval and waits for sender to send something. As part of the synchronization protocol, the sender sends a predefined starting sequence which lets the receiver know that the sender is starting the transmission and after he is done data transmission, he sends a null character denoting the end of communication. The receiver always keeps looking for the predefined sequence and after it receives one, it starts reading the data and until it receives the null character signifying the end of transmission.Once a transmission is complete, the receiver continues to access its address to again look for the predefined starting sequence denoting the beginning of a new transmission.
- The string which the guest has to transfer is sent character by character using ASCII encoding, i.e., 8 bits are sent for sending a single character.
- For accessing and recording the average time for particular access, we use the same method which we used for reverse engineering the DRAM mappings of the processor. After getting this reading, we feed the the average access time reading to a python file to analyze the data, plot it and print the received string.

6.3 Challenges Faced

6.3.1 Covert Channel between Host and Virtual Machine

- The first challenge we faced while making a covert channel between a host and a VM is that we couldn't use TSC for synchronizing the sender and the receiver. The *hypervisor* ensures that the TSC counter obtained by a user process in the VM is not the same as that of host. The reason being that allowing the same TSC counter in the VM can cause technical issues like jump in the TSC since the VM can get paused for some time. Also, allowing access to the Host's TSC counter can simplify various cache-based side-channel attacks because a malware can now know that he is in a VM and this can lead to various security problems.
- Running the VM on the host leads to heavy usage of DRAM which interferes with our experiment since we require that no other process affects the row buffer except the sender and receiver. If any other process competes for the same row buffer then the receiver might notice a high access latency even when the sender is sending a 0.
- Its was not clear to us whether the physical address that we used to get from the pagemap is the actual DRAM physical address or the physical address assigned by the VM page tables. We had to study a lot about the shadow page tables to understand how this page translation is done. This created a further hurdle in building a covert channel between the host and the VM.

6.3.2 Deciding Threshold

- During various runs of the experiment, we received different lower and upper limit for the access times due to noises in the measurement. We had to use techniques like taking the average of the high and low value as the threshold.

6.3.3 Memory Limitations

- As previously explained in the VM section, we require that a large amount of DRAM remains unused since other processes can interfere with our row buffer access times and we definitely don't want that.
- We had to kill other heavy processes and reboot the system various times to get the correct values so that we can minimize the DRAM usage by other processes.

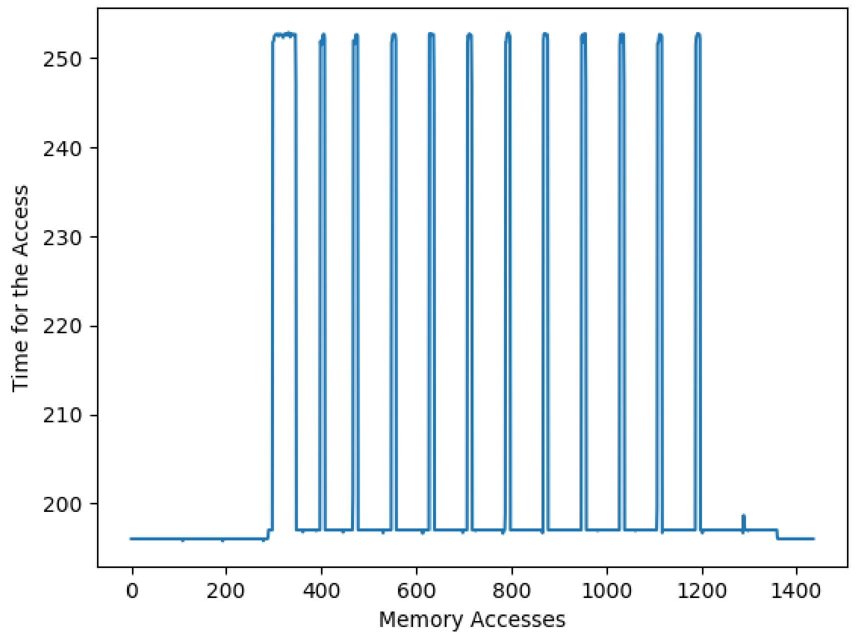


Figure 5: The Access Time Graph for the message '@@@@@@@@@@'. The transmission of the initial synchronizing sequence and only one 1-bit out of every 8 can be clearly seen.

6.4 Results

- We obtained quite good results in developing a covert channel between two processes running on the same machine. We are transmitting the message with very high reliability (at the cost of slow transmission rate).
- We transmitted the string '@@@@@@@@@@' and plotted the access time of all the access which the receiver performs. Figure 5 shows the plot. As we can see the difference between the access time when a 1 is sent and when 0 is sent is large enough and this enables us to completely remove the effect of random noise and get 100% reliability provided we set the threshold appropriately. We chose this string because '@' is 64 in ASCII which means that out of 8 characters sent for every character transfer, 7 will be zero. This is clearly visible in the graphs. Also notice that the first peak is slightly wider. This is because it is the peak of 5 consecutive ones which is present in our initial synchronising string.
- Please look at the Appendix for some more access time graphs.

7 Comparison with Flush and Reload

Flush and Reload is a cache-based CPU attack that is briefly explained. Two processes running on the same CPU have the same L3 cache. As a result, if a spy has access to victim memory, the spy can flush certain memory addresses from the cache. Note that the cc-flush library allows us to flush a memory address from all caches. Now, once certain memory addresses have been flushed from memory, if the victim program accesses those flushed memory addresses, the addresses are cached including in L3. Now, after a certain period of time, when the spy is scheduled again on the CPU, it tries to access the flushed memory addresses. If the time taken to access those memory addresses is comparable to cache access time, this indicates that the victim program accesses that address during the preceding period. We use this to design and devise particular attacks like finding the key in RSA.

Here we have demonstrated a Flush-Reload attack on a program that takes a string as input and does different things according to if the letter is uppercase or lowercase. The spy uses this to probe the instruction addresses of the respective functions.

Through this, we get an insight into the advantages and the drawbacks of the two attacks. The Flush-Reload attack requires a very high degree of control over the CPU for controlling the time that is yielded to the victim. Moreover, the threshold in Flush-Reload is also quite sensitive to the threshold time (the cache hit time) and this may get affected due to some CPU optimizations. The DRAM attack is not so sensitive to threshold times. The DRAM attack works in a more general setting and is not restricted to processes running on a single CPU. However, DRAM attack requires more pre processing in the form of finding the DRAM mapping for that processor, and hence takes much more time overall. The DRAM attack can also be used for covert channel communication with a high success rate, as has been demonstrated.

References

[1] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 565–581, 2016.

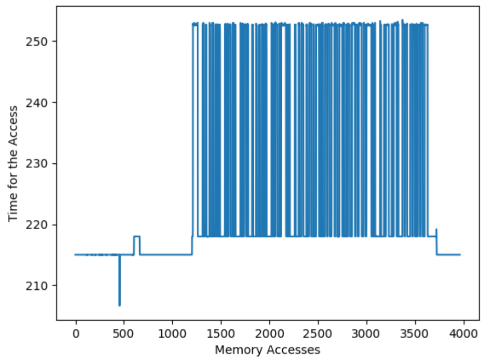
[2] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622, 2015.

[3] Yuval Yarom and Katrina E. Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

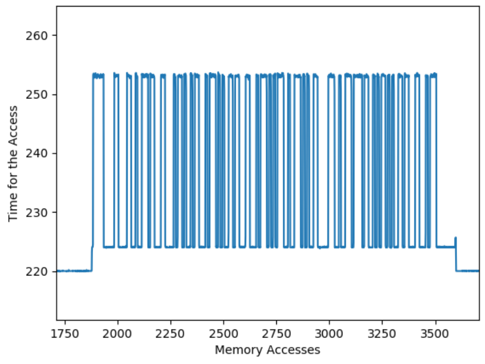
[4] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372, 2014.

8 Appendix

8.1 Messages transferred using the covert channel



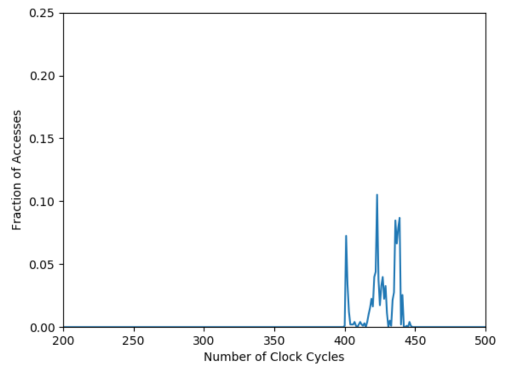
(a)



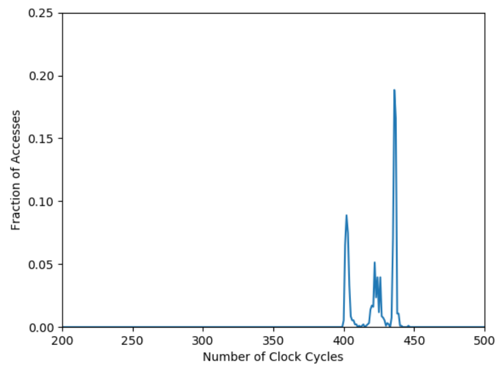
(b)

Figure 6: The access time graph in (a) corresponds to the string 'DRAMA-DRAM_Addressing_Attacks' and the graph in (b) corresponds to 'architectureproject'

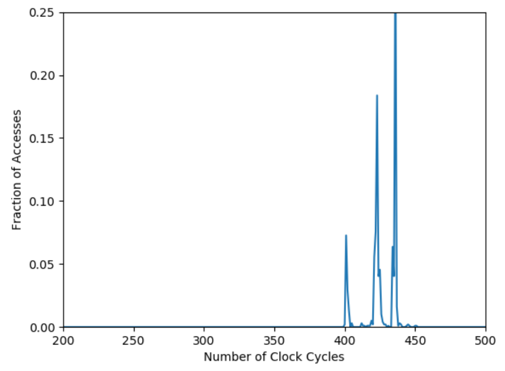
8.2 Histograms for verification of the functions



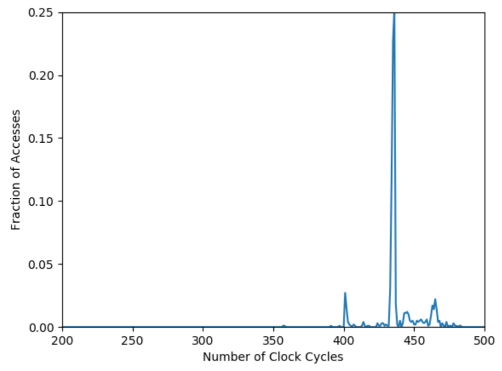
(a)



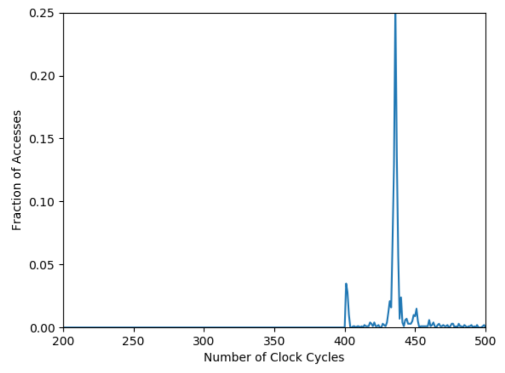
(b)



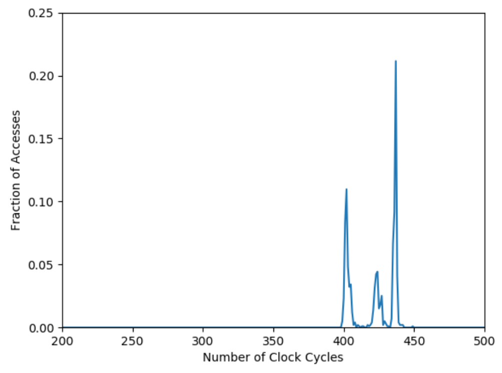
(c)



(d)

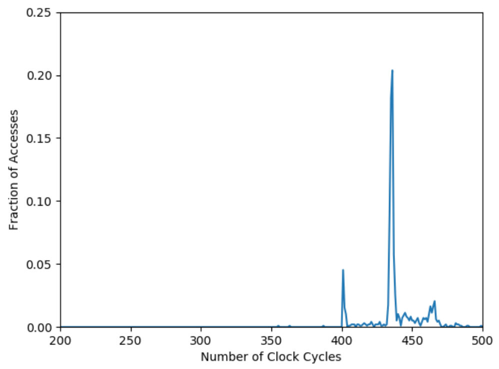


(e)

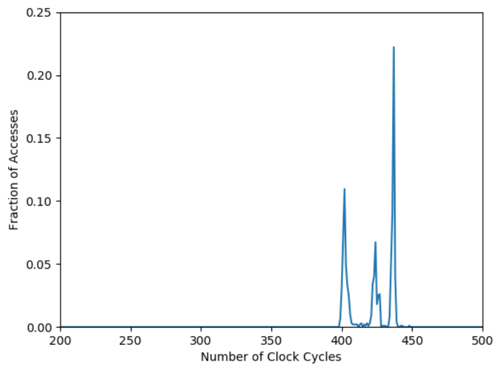


(f)

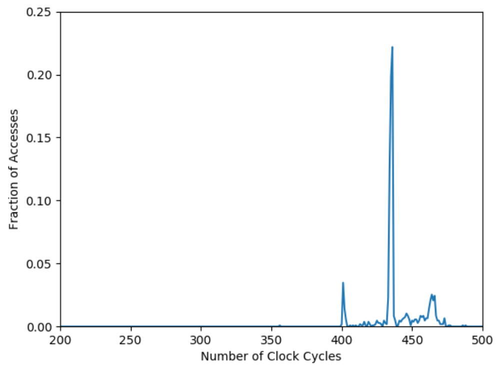
Figure 7: These are histograms corresponding to accesses from the same bank. As one can see the times are always smaller than the threshold of 395. (a), (b), (c), (d), (e) and (f) correspond to banks 3, 4, 5, 6, 7 and 8 respectively



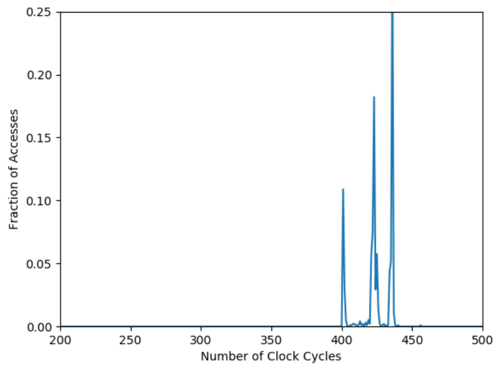
(a)



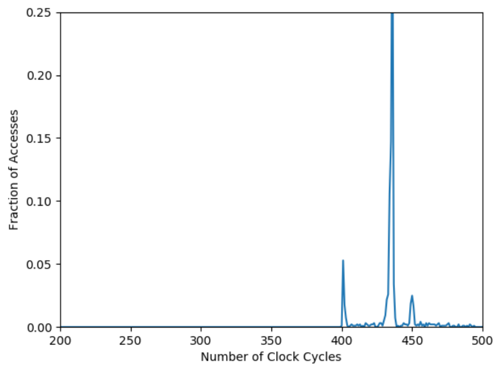
(b)



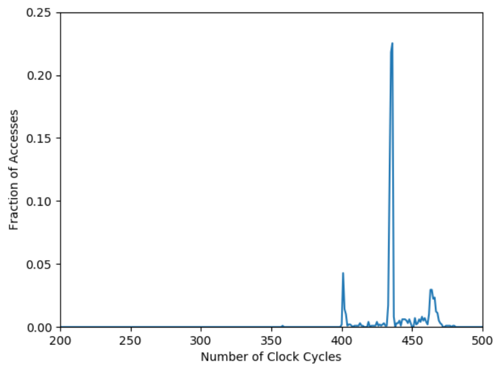
(c)



(d)

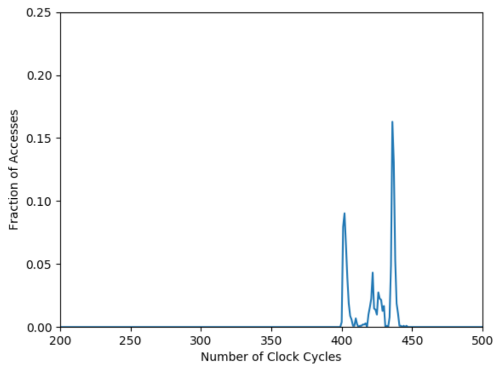


(e)

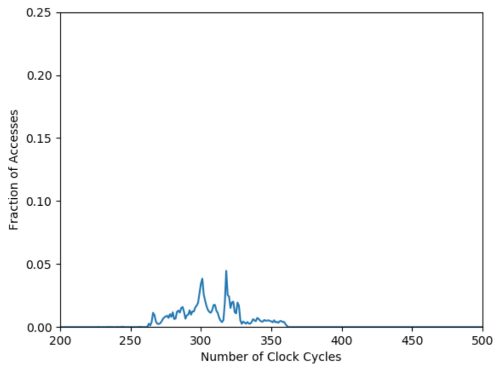


(f)

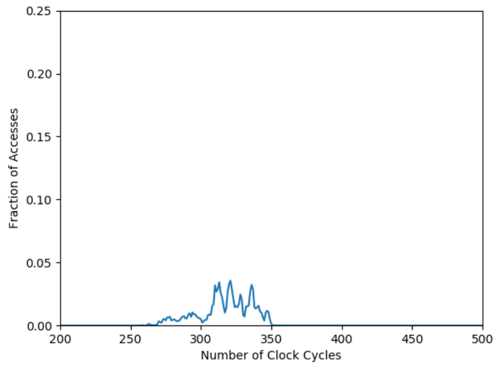
Figure 8: These are histograms corresponding to accesses from the same bank. As one can see the times are always smaller than the threshold of 395. (a), (b), (c), (d), (e) and (f) correspond to banks 9, 10, 11, 12, 13 and 14 respectively



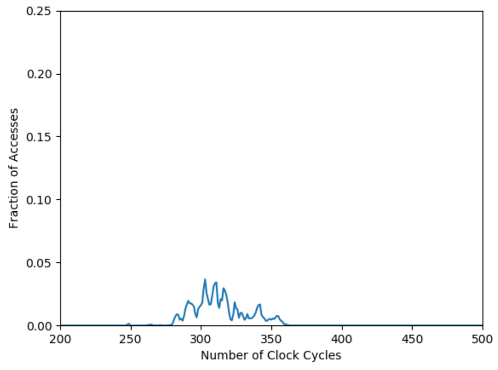
(a)



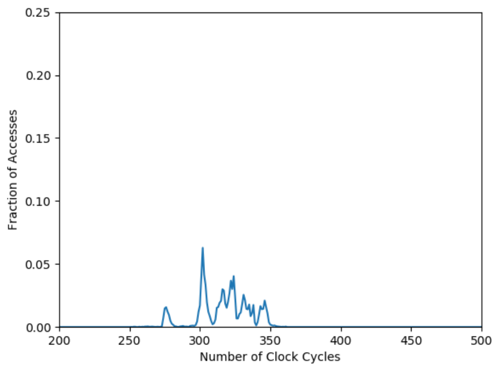
(b)



(c)



(d)



(e)

Figure 9: The histogram in (a) corresponds to accesses from bank 15. The other figures correspond to accesses from addresses that give different values on at least of the functions that we deduces. Figures (b), (c), (d) and (e) correspond to functions h_0, h_1, h_2 and h_3, respectively