Software-only Reverse Engineering of Physical DRAM Mappings for Rowhammer Attacks

Alessandro Barenghi
Politecnico di Milano - DEIB
Piazza Leonardo da Vinci, 32
20133, Milan, Italy
alessandro.barenghi@polimi.it

Luca Breveglieri
Politecnico di Milano - DEIB
Piazza Leonardo da Vinci, 32
20133, Milan, Italy
luca.breveglieri@polimi.it

Niccolò Izzo
Politecnico di Milano - DEIB
Piazza Leonardo da Vinci, 32
20133, Milan, Italy
niccolo.izzo@mail.polimi.it

Gerardo Pelosi
Politecnico di Milano - DEIB
Piazza Leonardo da Vinci, 32
20133, Milan, Italy
gerardo.pelosi@polimi.it

***Abstract*—In recent years, the ability to induce bit-flips in DRAM cells via software-only driven charge depletion has been successfully exploited to gain unauthorized privileged access to the functional resources on fixed and mobile computational platforms. The first crucial step in executing these attacks, collectively known as rowhammer attacks, concerns gaining the knowledge of how virtual memory addresses are mapped onto the geometric addresses of the physical DRAM module(s). We propose a methodology to reverse engineer such maps without direct physical probing of the DRAM bus of the target platform. We validate the correctness of the inferred maps against some publicly available data about modern Intel CPUs maps and show that they depend on the number of installed memory modules.**

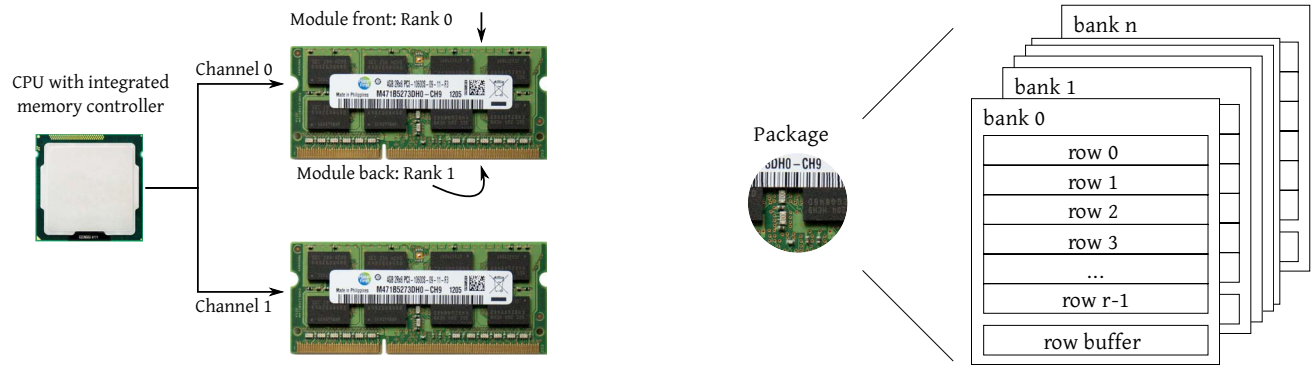
***Index Terms*—Rowhammer attacks, software-based memory mapping, timing side channel**

I. Introduction

The quest for increasing performance and energy efficiency in the design of computing systems has lead to a corresponding increase in the complexity of the optimizations that are inserted at any level, ranging from CPU microarchitecture, through computer architecture, to the structure of operating systems. Such an increase in complexity presents a pressing challenge when guaranteeing security properties, such as inter-process isolation and a hierarchical privilege structure among the system users. Indeed, the difficulty posed by predicting the effects on the security of the system, which stems from apparently harmless and performance-wise fruitful optimizations, is witnessed by the recently discovered *Meltdown* [1] and *Spectre* [2] vulnerabilities. Such vulnerabilities arise as a side-effect of an apparently harmless energy optimization that keeps as valid a cache line fetched as the result of a mis-speculated or anticipated `load` instruction. Security issues may also stem from problems that were commonly considered by architecture and system engineers only as reliability concerns. For instance, setup time violations in complex CPUs can be induced by a reduction in the power supply, or through forcing them to work outside their target clock frequency. However, if such setup time violations can be induced intentionally, their effect on the security of software implementations of cryptographic algorithms can be catastrophic, as shown in [3], [4], where the secret-key material is retrieved after observing a handful of faulty values. A component that has gone remarkably unchanged in the evolution of computing systems is the Dynamic Random Access Memory (DRAM). Since its first implementation on silicon in 1966 by Robert Dennard, DRAM modules have been increasing their capacity and read/write speed as the result of a steady shrinking in the etching technology. One key characteristic of DRAM memories is that data memorized in them fade over time naturally, as they are represented by charge levels in non-ideal capacitors. Thus every DRAM must be periodically read and rewritten, an operation known as *refresh*, to avoid data corruption. Despite the fact that a periodic *refresh* is performed, electrical phenomena such as *sub-threshold leakage* and *gate-induced drain leakage*, may still cause data alteration. Memory manufacturers have long known these phenomena, and always considered them as reliability issues, introducing mechanisms relying on error correcting codes, to allow the correction of a single-bit error and the detection of a two-bit error in the module.

While accidental memory corruptions due to fading are chiefly a reliability concern, the possibility of inducing such value changes intentionally, via software-only stimuli has proven to be a significant security problem. Indeed, Kim Yoongu et al. [5] identified and practically validated the possibility of circumventing memory protection mechanisms exploiting software-induced bit-flips in DRAM modules to change the protection map of user-accessible memory pages. This attack, known as *Rowhammer*, relies on repeatedly performing read accesses to a row of a DRAM block, causing charge depletion in the neighboring rows. The charge depletion results in flip downs in the stored values, which escape the common memory protection mechanisms enacted by the operating system and the CPU. Since its first description, the ability of Rowhammer to circumvent the access control barriers between different process domains has been extensively investigated to highlight and exploit security issues on various systems. Open literature reports attacks aimed at performing a privilege escalation of the hosting operating system from native environments on desktop [6] and mobile [7] platforms to browser sandboxes [8] and virtual machine environments running on third-party compute clouds [9]. In addition to privilege escalation, the faults induced in the stored data have also been used to thwart the security of cryptographic primitives in [10], [11]. A strict requirement for a the effectiveness of a *Rowhammer* attack, is the knowledge of the mapping between the physical addresses and the actual data location within the DRAM architecture. Such a mapping depends on the memory

978-1-5386-6544-2/IVSW18/



(a) Connection of DIMM modules to the CPU

(b) Intra-chip DRAM organization

Figure 1: Depiction of hierarchy of the components in a modern DRAM subsystem

controller being employed, and thus varies from a platform to another. The authors of [12] derived the mappings for a set of Intel CPU DRAM controllers by directly inspecting the memory elements with a digital sampling oscilloscope and decoding which banks were activated by a given `read` operation on a memory address. An alternative approach, which employs localized thermal disturbances on the DRAM module, was presented in [13].

Contributions. In this work, we describe a software-only methodology to infer the mapping of the physical memory addresses onto DRAM locations, relying uniquely on memory access times as the informative side channel. Our contribution allows to further prove the importance and effectiveness of the Rowhammer attacks, as it extends its practicality to systems where no physical access is possible to determine the physical addresses to DRAM locations map, and no clone of the attacked platform is available for probing. Moreover, we report that the physical addresses to DRAM locations map changes depending on the amount of memory modules present on the system in case of Intel x86_64 CPUs, a fact that provides an improvement over the knowledge of their maps reported in [12]. Finally, given the software-only character of our map derivation methodology, it allows to implement Rowhammer countermeasures in a unified fashion, performing the tailoring upon the specific DRAM architecture at boot time.

II. Preliminaries

In this section we summarize the main concepts concerning the structure of a modern DRAM architecture hierarchy, and provide a summary of the *Rowhammer* attacks in the existing literature, together with the strategy they employed to obtain the sequence of fast and accurate memory accesses required.

A. Background on DRAM Architecture

A modern DRAM based primary memory is connected to the memory controller residing on the CPU die via one or more *channels*, which are physically embodied by wiring in the host platform mainboard (see Fig. 1a). Each one of such channels is connected to one or more Dual Inline Memory Modules (DIMMs), depending on the electrical constraints of the motherboard. A DIMM is constituted by a Printed Circuit Board (PCB) holding a variable number of Ball Grid Array (BGA) packaged DRAM chips. The interconnection of DRAM chips is performed grouping them into *ranks*: most common DIMM modules have one or two ranks, and group all the chips belonging to a rank on the same side of the module PCB. High density DIMMs may have four or eight ranks, and exhibit no evident physical grouping of the chips belonging to the two or four ranks residing on the same side. A single DRAM chip contains one or more *banks* (Fig. 1b), i.e., two dimensional arrays of DRAM *cells*. For chips housing a large number of banks, such banks may be organized in *bank groups*, sharing the on-chip access logic. The readout from the DRAM *cells* is performed copying a *row* of the *bank* into a dedicated buffer known as *row buffer*. In case the number of rows in a bank does not allow the readout due to electrical constraints, a bank may be split into separate *subarrays*, each one of which is endowed with a separated row buffer. All the subarray row buffers are connected to the main bank row buffer. Typical memory accesses require an amount of data smaller than an entire row. To this end, the chip arranges only a set of contiguous bytes identified by a *column* index to be forwarded to the DIMM, and from there back to the CPU memory controller.

From the point of view of the program execution on the CPU side, the main memory locations are identified by a binary string representing a positive integer known as *physical address*. It is thus necessary to devise a map between the physical address and a tuple of indexes, reported in Table I, which uniquely identify the DRAM location of a given byte. We will employ the corresponding uppercase letters to indicate the size in bits of the maximum value taken by the index. The first four indexes identify in which channel the location is present (k), which rank in the said channel is involved (l), which bank in the said rank (b), and which is the row (r) to be copied in the row buffer to access the desired datum. Determining the actual position in a row may be performed in two ways: either consider an index c indicating the column in a row, and an index v pointing out the byte-accurate location in the column, or considering the row partitioned in cache line sized chunks indexed by n and having an index z point the byte-accurate location in the chunk. The memory controller designer exploits the freedom in mapping the physical address

Table I: Indexes uniquely identifying the DRAM location corresponding to a physical address in the memory hierarchy

Symbol	Description
k	Channels in a system
l	Ranks per channel
b	Banks per rank
r	Rows per bank
c	Columns per row
v	Bytes per column
n	Cachelines per row
z	Bytes per cacheline

to an actual DRAM location to maximize the performance in terms of latency and bandwidth of the memory subsystem [14]. Willing to maximize the performance of the accesses to consecutive physical addresses, the most common choice is to map consecutive physical addresses onto contiguous bytes in a single row. Such a choice maximizes the effectiveness of large burst transfers if the memory bus allows it. Therefore, the bits starting from the least significant one, in the physical address represent the binary encoding of the byte-accurate position of the datum in the row. Besides this first technique, the intrinsic parallelism of the multiple channels is exploited mapping row-sized physical address intervals on different channels. Finally, the designer should avoid as much as possible that a common physical address pattern results in a sequence of accesses performed on different rows of the same bank. Indeed, such accesses require the row buffer to be loaded with a different row from the one it is containing, and must take place in strict sequential order among them. A similar problem, although with significantly less impacting latencies, is the contention over the bank access logic in a rank; therefore subsequent accesses to different banks in a rank should also be avoided. Since there is no general technique to prevent such issues, a widely adopted solution is to combine the bits of the physical address contiguous to the ones representing the byte-accurate location via linear Boolean functions [15] to derive the l, b and r indexes. Such a technique prevents a single long sequence of accesses to consecutive physical addresses from generating a significant amount of resource contention.

The choices made by the memory controller designer are typically not publicly disclosed, with the notable exception of the memory controller employed by AMD processors belonging to Family 15h, models 00h-0Fh for which the map can be found in [16]. Given the criticality of knowing the map function to perform efficient Rowhammer attacks, different approaches to deriving it (or an approximation thereof) are reported in open literature. Besides the approaches requiring physical access to the machine described in [12], [13], successful attempts at deriving enough information to perform Rowhammer attacks were reported in [7] and in [17]. In particular, the first approach derived an alleged mapping function for an ARM based System-on-Chip performing a sequence of pairs of accesses to physical addresses. The pairs were chosen adding a variable offset, picked as a multiple of 4kiB (i.e., the operating system page size), to both the first and the second address of the pair. The authors of [7] infer that the DRAM row size is 16×4 kiB, as the access latencies spike with that same spatial period on the offsets. Such spikes are ascribed to the second access of an access pair forcing the copy of a new row in the row buffer. Finally, the software only approach reported in [12] could not be reproduced on Intel CPUs of the same generations, when endowed with a different number of DIMMs. From this fact, we deduced that the map function depends both on the specific memory controller and the memory configuration (i.e, the number of DIMMs or ranks per DIMM) installed on the mainboard at hand.

B. The Rowhammer Attack

The Rowhammer attack relies on the fact that a row readout into the row buffer accelerates the rate at which the charge is depleted in the cells of neighboring rows. Therefore, reading repeatedly and frequently the same row will decrease the amount of time for which the datum stored in the neighboring row is stable, inducing potentially a flip-down of one or more bits. If this takes place before a charge refreshing action is taken, the charge refreshing will not restore the correct value of the bit, as it has no way of detecting that it took place in the first place (save for DRAMs equipped with sufficient error-correction mechanisms). The charge depleting can be further accelerated if the repeated read accesses are performed both on two rows adjacent to the same one, a technique known as *double rowhammering*. We note that performing a repeated access to a memory location with a simple program constituted by a tight loop may not be enough to trigger the Rowhammer phenomenon. Indeed, besides the high frequency of the accesses, and the fact that they should target the same location, such accesses should also be uncached, i.e., the attacker should ensure that all `load` operations are performed from the DRAM and not from one of the available caches. Open literature reports four viable ways of attaining such frequent and uncached load operations: employing dedicated cache line flush instructions (e.g., Intel's `clflush` instruction) [5]; exploiting cache eviction strategies through forcing application mandated cache flushes [8]; exploiting instructions designed to perform memory accesses that are not stored in cache, known as *non-temporal accesses* (e.g., Intel's `movnti`, `movntdq` instructions) [18]; and exploiting memory portions marked as uncached by the operating system [7].

III. Software-only Mapping Methodology

In the following, we describe how the measurement of the latency of pairs of physical address accesses allows to derive the Boolean function that maps them to DRAM locations.

A. Access-latency based mapping function inference

Our goal is to derive a Boolean function that maps the bits of a physical address `pa` to the locations of the DRAM architecture present on the mainboard at hand. Let us denote with `P` the number of bits of a physical address provided to the on-die memory controller (i.e., $\texttt{pa} = \langle \texttt{pa}[1], \texttt{pa}[2], \ldots, \texttt{pa}[\texttt{P}] \rangle$, where $\forall\, j \in \{1, \ldots, \texttt{F}\}$, $\texttt{pa}[j] \in \{0, 1\}$), and with `K`, `L`, `B` the number of bits needed to select one channel, one rank and one bank in the available DRAM, respectively (i.e., the bit sizes of the indexes k, l, b reported in Table I).

The Boolean function employed by the on-die memory controller to translate the physical memory addresses to the DRAM location of a bank, $f : \{0,1\}^{\mathtt{P}} \rightarrow \{0,1\}^{\mathtt{K+L+B}}$, is supposed to be a linear function in canonical Algebraic Normal Form (ANF), i.e., $f(x_1, x_2, \ldots, x_{\mathtt{P}}) = x_{i_1} \oplus x_{i_2} \oplus \cdots \oplus x_{i_s}$, with $x_j \in \{0,1\}$, $1 \leq j \leq \mathtt{P}$ and $0 \leq i_l < \mathtt{K+L+B}$, $1 \leq l \leq s, s \leq \mathtt{P}$.

The hypothesis of keeping a linear Boolean function f constituted only by a `xor` combination of literals, x_{i_l}, to model the combination of physical address bits to select the DRAM location of a bank (i.e., LITERALS(f)= $\{x_{i_1}, x_{i_2}, \ldots\}$, where x_{i_l} is in one-to-one correspondence with $\mathtt{pa}[i_l]$), is consistent with the functions reported in [12], and the fact that it is desirable for the output of the said Boolean function to be uniform over its codomain. The knowledge of such a function would allow to perform accurate Rowhammer attacks, as it would allow to deterministically determine the physical addresses mapped to the same bank. Indeed, picking physical memory addresses $\mathtt{pa}_1, \mathtt{pa}_2, \mathtt{pa}_3 \ldots$ mapped to different rows in the same bank would be accomplished by *1)* checking that $f(\mathtt{pa}_1) = f(\mathtt{pa}_1) = f(\mathtt{pa}_3) = \ldots$, and *2)* properly choosing the values of the remaining $\mathtt{P-K-L-B}$ bits of the physical address. Given the knowledge of the function f, it is possible to completely map a DRAM bank exploiting the fact that the Rowhammer effect acts only on neighboring rows. Therefore, given f it is possible to fill the cells of an entire bank with ones and systematically hammer its rows one by one. The adjacency relation among the rows can be derived observing where the bit flips take place. As reported in the previous section, the design and configuration of the function f present in the on-die memory controller is not publicly known for most CPUs. Our methodology to derive such a map is articulated in two phases: *(i)* finding pairs of physical addresses mapped onto the same bank, and *(ii)* deriving f from the said address pairs.

Finding Same-Bank Different-Row Accesses. Considering two subsequent accesses to two random physical memory addresses, the Same-Bank Different-Row (SBDR) accesses should occur with a probability that depends on the number of overall banks. Therefore the first step of our technique is to measure the access latency of pairs of randomly chosen, valid, physical addresses and cluster such pairs in sets S_τ according to the measured access latency value τ. Once a statistically significant amount of access latencies is collected, they are partitioned in sets S_τ clustering them by their access latency τ. Drawing the mentioned pair of accesses uniformly over the entire memory, it is expected that the sets S_τ with the highest values of τ (i.e., the ones closer to the actual SBDR latency), contain a fraction of the total pairs of accesses approximately equal to $\frac{1}{\mathtt{Tot.num.of banks}}$. We collect all such sets in a collection $\mathcal{S}$, $\mathcal{S} = \bigcup_\tau \{S_\tau\}$, where τ is a suspected SBDR latency. To match the scenario of a Rowhammer attack, the measurements of the aforementioned accesses are performed from a test running as a user-space process and the measured latencies are affected by some amount of noise w.r.t. the same accesses running on the bare-metal.

Deriving Boolean Functions. Once the collection $\mathcal{S}$ is computed, we proceed to generate a set of linear Boolean functions and to choose among those the ones that best fit the mapping applied by the on-die memory controller according to the procedure reported in Algorithm 1. We note that, to the end of reducing the computational load of generating all the functions $f : \{0,1\}^{\mathtt{P}} \rightarrow \{0,1\}^{\mathtt{K+L+B}}$, it is possible to consider a reduced amount of bits of the physical address. More precisely, it is convenient to discard the bits of the physical address that allow to select the intra-row location of the datum, i.e., the ones identifying a column (c) in the row of a bank and the ones selecting a byte (v) in the identified data chunk. This effectively reduces the computational effort needed for searching for the desired linear Boolean function to the one required for searching for $f : \{0,1\}^{\mathtt{P-C-V}} \rightarrow \{0,1\}^{\mathtt{K+L+B}}$, where `C` denotes the number of bits needed to select a column in a row, while `V` denotes the number of bits needed to uniquely identify a byte in the chunk of data readout with the column index. Algorithm 1 takes as input the collection $\mathcal{F}$ of all the linear Boolean functions $f : \{0,1\}^{\mathtt{P-C-V}} \rightarrow \{0,1\}^{\mathtt{K+L+B}}$, i.e., $\mathcal{F} = \langle f_1, f_2, \ldots f_i, \ldots \rangle$, with $|\mathcal{F}| = 2^{\mathtt{K+L+B}} \cdot 2^{\mathtt{P-C-V}}$.

Algorithm 1: Boolean map function computation

Input: $\mathcal{S} = \{S_{\tau_1}, S_{\tau_2}, \ldots, S_{\tau_{|\mathcal{S}|}}\}$, where $S_{\tau_j} = \{(\mathtt{pa}_1, \mathtt{pa}_2)_1, \ldots\}$ includes all pairs of physical addresses with a τ_j SBDR access time, $1 \leq j \leq |\mathcal{S}|$;
$\mathcal{F} = \langle f_1, f_2, \ldots f_i, \ldots \rangle$, list of linear Boolean functions.
Output: $\mathcal{M}$ is the set of candidate Boolean map functions.

```
1  R ← ∅  // list of pairs ⟨f, score⟩, where
2         // f ∈ F, score an integer number
3  foreach f ∈ F do
4      score ← 0
5      foreach S_τ ∈ S do
6          foreach (pa_1, pa_2) ∈ S_τ do
7              if f(pa_1) = f(pa_2) then
8                  score ← score + 1
9      R ← R ∪ ⟨f, score⟩
10 R_1 ← SortDescByScore(R)
11 R_2 ← Head(R_1, truncateThreshold)
12 D ← ∅  // Set of functions f∈R_2
13        // dominated by a g∈R_2
14 foreach f ∈ R_2 do
15     foreach g ∈ R_2 do
16         if LITERALS(f) \ LITERALS(g) = ∅ ∧
             LITERALS(g) \ LITERALS(f) ≠ ∅ then
17             D ← D ∪ {f}
18 M ← R_2 \ D
19 return M
```

Algorithm 1 examines each function $f \in \mathcal{F}$ (line 3) computing a score corresponding to the number of pairs of physical addresses $(\mathtt{pa}_1, \mathtt{pa}_2)$ in all $S_\tau \in \mathcal{S}$ for which $f(\mathtt{pa}_1) = f(\mathtt{pa}_2)$ (lines 5–8). This score corresponds to the number of address pairs that are likely to be SBDR according to the measured latencies, and are deemed as such by the function f. Each function is paired with the computed score, and a set $\mathcal{R}$ of pairs $\langle f, \mathtt{score} \rangle$ containing them is built (lines 1–9). The function-score pairs are sorted in decreasing score order and the set $\mathcal{R}_2$ of the highest scoring ones is selected (lines 10–11), employing a threshold (`truncateThreshold`) that is heuristically chosen (e.g., `truncateThreshold = 100`). The reason to pick multiple candidates is to provide resilience against measurement errors. The last step in the procedure

(lines 14–18) takes into account the fact that, given two functions $f, g \in \mathcal{R}_2$, it is possible that g contains all the literals contained in f. Such a case takes place when g is the actual correct map function, and f attains a high score only on the basis of the fact that it evaluates identically to g whenever the literals of g missing from f evaluate to zero on the given input. To prevent a function such as f, which is ignoring the contribution of some address bits, to be considered as the correct map function, Algorithm 1 detects functions such as f in lines 14–16, storing them in the set $\mathcal{D}$, and computes the set of candidate functions $\mathcal{M}$ as $\mathcal{R}_2 \setminus \mathcal{D}$.

B. Achieving Reliable Memory Access Measurements

The procedure described in the previous section relies on the ability to gauge the latency of a pair of accesses executed in a user-space process taking care to reduce as much as possible the measurement noise. The first point to collect accurate measurements is to employ the architectural support provided by the x86_64 Instruction Set Architecture in terms of the Time Stamp Counter (TSC): a 64-bit register counting the number of clock cycles since the bootstrap of the CPU. Intel CPUs provide the `rdtsc` instruction that performs a readout of the TSC into a pair of registers. However the `rdtsc` instruction can be reordered with neighboring ones by the out-of-order execution policy of the CPU, effectively allowing the execution of the `load` operations to be measured before or after the TSC is read. To the end of preventing such a behavior, we pair the `rdtsc` instruction with a `cpuid` one, which acts as a microarchitecture-level barrier according to [19]. Alternatively, on modern platforms, the availability of the `rdtscp` instruction, which is equivalent to `rdtsc` except for the fact that it enforces serialization with respect to all other instructions in flight in pipeline. However, in both the case of `rdtsc` and `rdtscp`, the TSC is read before pending `load` instructions complete the fetch of their value from the main memory, except in the case another instruction using it is executed before `rdtsc[p]`. To prevent this phenomenon from polluting our measurements, we insert a memory fence (`mfence`) instruction before the `rdtsc[p]`, so that all the `load` operations are completed before the TSC is read.

Precise access timing measurements cannot be obtained if memory accesses are cached, thus we obtain uncached accesses by issuing a `clflush` instruction on the two target addresses just before reading them. The `clflush` instruction flushes all the cache levels and the CPU write-buffers; thus, all the subsequent read commands are sent to the DRAM. Since we are performing our measurements in a GNU/Linux Operating System (OS) process running in user-space, the measurement procedure may be subject to preemption and CPU migration, adding further noise to the measurements. We minimize those phenomena by issuing a `sched_yield` *system call* (syscall) before our critical code sections and by pinning our process to a single CPU, respectively. The `sched_yield` syscall makes our process relinquish the remainder of its time quantum, so that execution will continue with a full one, making preemption more unlikely to happen.

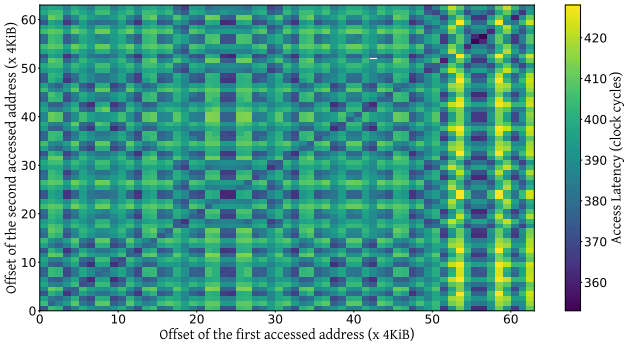
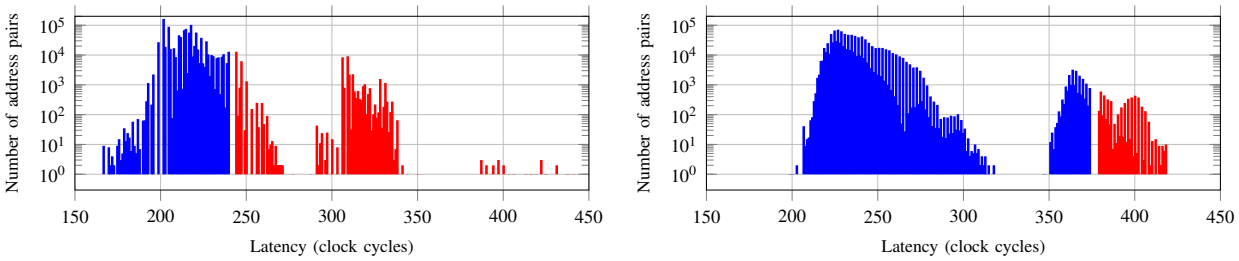


Figure 2: Access latency heatmap of our Skylake platform, derived applying the method in [7]

Finally, all the DRAM modules are periodically refreshed at least every 64ms. If we attempt at accessing a row during *refresh*, the measurement can be thrown off by an order of magnitude, as a row cannot be transferred while refreshing it. Since the likelihood of accessing a row during refresh is rather low, we perform three accesses to the same pair of physical addresses and repeat all of them until the three TSC binary values agree up to the third least significant bit.

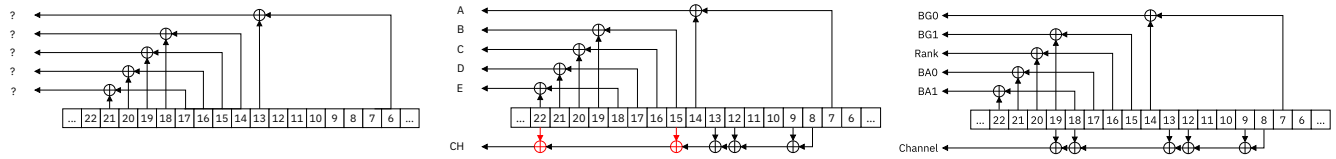
IV. Experimental Validation

We validated our technique on two different Intel CPUs, an Intel Sandy Bridge i5-2520M clocked at 2.5GHz with two, 4GiB, M471B5273DH0-CH9 DDR3 DIMM modules, and an Intel Skylake i5-6500 clocked at 3.2Ghz where we tested both a configuration with four, 8GiB KHX2133C14D4 DDR4 DIMM modules in dual channel configuration, and a configuration with a single channel/DIMM module. The two computers were running ArchLinux (kernel version 4.13) and Gentoo Linux (kernel version 4.8), respectively. To validate the need for an accurate methodology to compute the physical address to DRAM location map, we conducted a first experiment trying to reconstruct the map on our Intel Skylake i5-6500 employing the method in [7]. Figure 2 reports the results of such an attempt, showing that the complex nature of the Skylake map function prevents a straightforward deduction from being made as in [7]. We move onto the evaluation of our mapping technique. We report that the number of Boolean functions $|\mathcal{F}|$ to be materialized in our setup is at most $|\mathcal{F}|=2^{\mathtt{K+L+B}}\cdot 2^{\mathtt{P-C-V}}=2^{18}\approx 2.56\cdot 10^5$, considering the fact that a row is 2^{13} bits long in the modules at hand (i.e., `C+V`=13), the effective physical address bits are `P`=35 (as the installed memory size is 32GiB), the modules are single-ranked (`L`=0), and the number of banks is at most 32 (`B`=5), with a dual channel configuration (i.e., `K`=1). Figure 3 reports the histograms of the size all sets S_τ for both the Sandy Bridge and Skylake platforms, where the red bars represent the sets chosen for inclusion in $\mathcal{S}=\bigcup_\tau\{S_\tau\}$, recalling that $\sum_{S_\tau\in\mathcal{S}}|S_\tau| = \frac{\mathtt{Total\ accesses}}{\mathtt{Num.\ of\ banks}}$. We note that the SBDR accesses are set apart by a rather clean cut in the Skylake platform, while the Sandy Bridge appears to present rather noisier measurements. Figure 4 reports the best candidate functions obtained by a Python implementation of Algorithm 1



(a) Sandy Bridge platform (b) Skylake platform

Figure 3: Histograms of the number of physical address access pairs (y-axis) exhibiting a given latency (x-axis)



(a) Our derivation (single channel) (b) Our derivation (dual channel) (c) Map provided in [12]

Figure 4: Skylake map functions derived by us for single channel (a) and dual channel (b) memory configurations, and from [12]

for the Skylake platform, both in a single and dual channel configuration (Fig. 4a and Fig. 4b, respectively) compared to the function reported in [12] for a dual channel configuration. We note that, in our derived mapping, we cannot assign a semantic to the output Boolean values, as there is no way to distinguish, nor need to for rowhammering purposes, the ones dedicated to channel, rank or bank index from our point of observation. We observe that the function we retrieved matches the one obtained via direct oscilloscope probing in [12], up to two swaps in the contribution of the input bits (in red in Fig. 4). Indeed, such bit contribution swap is transparent from the SBDR access point of view, as it changes the role of the contribution of physical address bits to the channel and bank index. Figure 4a shows that the map function between physical addresses and DRAM locations effectively changes in case a different memory configuration (a single channel instead of a dual channel one) is employed; indeed, the physical address bits involved in the computation of the channel, rank and bank indexes are different from both Fig. 4b and Fig. 4c.

V. Concluding Remarks

We showed the feasibility of retrieving a physical address to DRAM location map function exploiting the information of the access latency timing side channel. Our methodology is purely software and requires the same attacker model of Rowhammer (user-space execution on to the host). We have shown that the map function depends on the memory configuration of the host and thus is not uniquely dependent on the memory controller.

References

[1] M. Lipp et al., "Meltdown," *CoRR*, vol. abs/1801.01207, 2018.

[2] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution," *CoRR*, vol. abs/1801.01203, 2018.

[3] A. Barenghi, G. M. Bertoni, L. Breveglieri, and G. Pelosi, "A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1864–1878, 2013.

[4] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management," in *26th USENIX Sec. Sym.* USENIX Association, 2017, pp. 1057–1074.

[5] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA 2014*. IEEE.

[6] M. Seaborn, contributed by T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," http://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, March 2015.

[7] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS 2016*. ACM.

[8] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, ser. LNCS, vol. 9721. Springer, 2016, pp. 300–321.

[9] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation," in *25th USENIX Sec. Sym.*, 2016, pp. 19–35.

[10] S. Bhattacharya and D. Mukhopadhyay, "Curious Case of Rowhammer: Flipping Secret Exponent Bits Using Timing Analysis," in *CHES*, ser. LNCS, vol. 9813. Springer, 2016, pp. 602–624.

[11] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a Needle in the Software Stack," in *25th USENIX Sec. Sym.*, 2016, pp. 1–18.

[12] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Sec. Sym.*, 2016, pp. 565–581.

[13] M. Jung, C. C. Rheinländer, C. Weis, and N. Wehn, "Reverse engineering of drams: Row hammer with crosshair," in *MEMSYS 2016*. ACM.

[14] D. T. Wang, "Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm," Ph.D. dissertation, University of Maryland, College Park, MD 20742-7011 (301)314-1328., 2005.

[15] Z. Zhang, Z. Zhu, and X. Zhang, "Breaking Address Mapping Symmetry at Multi-levels of Memory Hierarchy to Reduce DRAM Row-buffer Conflicts," *J. Instruction-Level Parallelism*, vol. 3, 2001.

[16] Advanced Micro Devices, "BIOS and Kernel Developer's Guide for AMD Family 15h Models 00h-0Fh Processors," http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf, 2013.

[17] M. Seaborn, "How physical addresses map to rows and banks in DRAM," http://lackingrhoticity.blogspot.it/2015/05/how-physical-addresses-map-to-rows-and-banks.html, March 2015.

[18] R. Qiao and M. Seaborn, "A new approach for rowhammer attacks," in *HOST*. IEEE CS, 2016, pp. 161–166.

[19] G. Paoloni, "How to Benchmark Code Execution Times on Intel® IA-32 and IA-64," Intel Corp., Tech. Rep., 2010. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf