3 Cache Reverse Engineering [60 points]

Consider a processor using a 4-block LRU-based L1 data cache. Starting with an empty cache, an application accesses three L1 cache blocks in the following order, where consecutive numbers (e.g., n, $n+1$, $n+2, ...$) represent the starting addresses of consecutive cache blocks in memory:

$$n \rightarrow n+2 \rightarrow n+4$$

3.1 Part I: Vulnerability [35 points]

A malicious programmer realizes she can reverse engineer the number of sets and ways in the L1 data cache by issuing *just* two more accesses and observing *only* the cache hit rate across these two accesses. Assume that she can insert the malicious accesses only after the above three accesses of the program.

1. [15 points] What are the next two cache block she should access? (e.g., [n+?, n+?])

There are two possible answers:

- **[n → n+4]**
- **[n → n+2]**

Explanation. There are three possible set/way configurations, shown below labeled by their respective sets/ways. Each configuration shows a drawing of the cache state after the three initial accesses. Rows and columns represent sets and ways, respectively, and the LRU address is shown for each occupied set:

(a) **(4 sets, 1 way)**

$(n+4)_{LRU}$
-
$(n+2)_{LRU}$
-

(b) **(2 sets, 2 ways)**

$n+4$	$(n+2)_{LRU}$
-	-

(c) **(1 set, 4 ways)**

n_{LRU}	$n+2$	$n+4$	-

At this point, all three caches have a 100% miss rate since they started cold. In order to differentiate the three cases with *just two* more accesses, we need to induce *different* hit/miss counts in each of the three types of caches. The only way this is possible is if one cache type experiences two hits, another two misses, and the last one hit and one miss.

Only two solutions exist to produce this case:

- **[n → n+4]**
  (a) n miss, $n+4$ miss = 100% miss rate
  (b) n miss, $n+4$ hit = 80% miss rate
  (c) n hit, $n+4$ hit = 60% miss rate
- **[n → n+2]**
  (a) n miss, $n+2$ hit = 80% miss rate
  (b) n miss, $n+2$ miss = 100% miss rate
  (c) n hit, $n+2$ hit = 60% miss rate

2. [10 points] How many L1 sets/ways would there be if the cache hit rate over the 2 extra instructions was:

There are two possible solutions due to the two possible solutions to part (1). They directly encode the hit/miss rates for the last two accesses shown in the solution above:

- Solution 1:

L1 hit rate	# sets	# ways
100%	1	4
50%	2	2
0%	4	1

- Solution 2:

L1 hit rate	# sets	# ways
100%	1	4
50%	4	1
0%	2	2

3. [10 points] What should the next two accesses be if the replacement policy had been Most Recently Used (MRU)? (e.g., [n+?, n+?])

There is no solution for just two more accesses because with an MRU policy, no permutation of two more accesses is able to assign a unique L1 hit rate to each of the three types of caches.

3.2 Part II: Exploitation [25 points]

Assuming the original cache design (i.e., with an LRU replacement policy) is using a 1-set (4-way) configuration, the malicious programmer decides to disrupt a high-security banking application, allowing her to transfer large amounts of foreign assets into her own secure Swiss account. By using a carefully designed concurrently running process to interfere with the banking application, she would like to induce slowdown, thereby exposing opportunity for her mischief.

Assume that the unmodified banking application issues the following access pattern, where each number represents X in $n + X$:

$$0 \rightarrow 6 \rightarrow 1 \rightarrow 7 \rightarrow 6 \rightarrow 1 \rightarrow 4 \rightarrow 0 \rightarrow 5 \rightarrow 0 \rightarrow 7 \rightarrow 4 \rightarrow 2 \rightarrow 7 \rightarrow 2 \rightarrow 4$$

1. [10 points] What is the unmodified banking application's cache hit rate?

$\frac{7}{16}$

2. [15 points] Now, assume that the malicious programmer knows the access pattern of the banking application. Using this information, she is able to inject a **single** extra cache access in between each of the banking application's accesses (i.e., she can interleave a malicious access pattern with the normal application's execution).

   What is the minimum cache hit rate (not counting extra malicious accesses) that the malicious programmer can induce for the banking application?

$\boxed{\frac{2}{16}}$

Explanation. Since the malicious programmer wants to cause every cache access by the banking application to miss, she must ensure that any block brought into the cache by an application access is evicted before it is accessed again. The (1 set, 4 way) configuration causes a set conflict between any two possible blocks, so in order to force an eviction of a given line, she must ensure that *four* cache accesses to different, unique cache lines are performed. This guarantees that accessing the given line again results in a miss.

Because she can only insert one extra cache access in between each pair of normal application accesses, she cannot force an eviction of the first access in the two sequences "...0 → 5 → 0..." and "...2 → 7 → 2...". In these two sequences, accesses to the same cache block are separated by only one access. This means that the malicious programmer can only cause a total of three distinct cache block accesses in between the accesses to the same cache block, and therefore, she cannot prevent a cache hit.

Thus, despite the malicious programmer's best effort, the two cache accesses shown below in bold will still hit in the cache, resulting in an application cache hit rate of $\frac{2}{16}$.

0 → 6 → 1 → 7 → 6 → 1 → 4 → 0 → 5 → **0** → 7 → 4 → 2 → 7 → **2** → 4

Here is an example cache access sequence resulting in the solution hit rate, with the malicious injected accesses shown in blue:

0 → 2 → 6 → 3 → 1 → 4 → 7 → 5 → 6 → 2 → 1 → 3 → 4 → 6 → 0 → 7 → 5 → 1 → **0** → 2 → 7 → 3 → 4 → 5 → 2 → 6 → 7 → 0 → **2** → 1 → 4 → 3

Note that this is one of many possible such sequences.