INDIAN INSTITUTE OF TECHNOLOGY, MADRAS

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MICRO-ARCHITECTURAL ATTACK AWARE LINUX SCHEDULER
CS3500 COURSE PROJECT REPORT

Krishna Sharma - CS18B021,
Sarthak Kapoor - CS18B066

December 7, 2020

Contents

ABSTRACT

With the increasing usage of computer systems, the need for efficient system security is increasing. Handling side channel attacks is a crucial aspect for building secure systems. In this report we present an approach to use Hardware Performance Counters (HPCs), which are registers in the hardware that store important event frequencies, to build a scheduler for Linux kernel (version 4.19.160) which can handle side channel attacks.

We start with finding out the relevant HPCs which can distinguish an attacking process from a normal process. We then find an effective heuristic by cleverly using a Machine Learning approach. We finally implement this in the existing Linux kernel code by making several modifications to it.

Through this project, we found out that L1 cache miss rate is one of the best performance statistic that can be used to differentiate between attacking and non attacking processes. We saw that the execution of an attacking process is notably different from a normal process. We can exploit these differences to build a scheduler that can detect such attacks and slow them down.

INTRODUCTION

`Side channel attacks` are the attacks based on the poor implementation of the computer system (micro-architecture or operating system) that can cause an attacker to gain information about a program. Side channel attacks are a threat to any secure system as they can be exploit the inconsistencies of the system to gain information about other protected programs running on the system. Such shortcomings of micro-architecture are not trivial to handle in hardware as well as in the operating system even though it is critical to handle such security threats. Popular side channel attacks include cache attacks, timing attacks and power-monitoring attacks. We here provide a scheme for handling such cache attacks by making modifications in the Linux kernel.

Modern computers have special purpose registers built into their architecture that monitor several hardware activities called Hardware Performance Counters (HPCs). These are registers that can be used to store counts of hardware events like no. of cycles, instructions executed, cache misses and loads, etc. These HPCs can be used to identify patterns in execution of processes. Notably, the execution patterns for normal process and a process doing side channel attacks are different. We exploit these differences to identify whether a process is doing a side channel attack in the kernel and handle it accordingly.

Some of the related work in this field that we referred to are [1] for understanding the meaning of side channel attacks and various possible solutions to it, [2] which gave us inspiration to use performance counters and machine learning models to classify processes as attacking or not.

In this project, we tried mainly two approaches to handle an attacking process once it is

identified.
The first is process migration. Many processes doing side channel attacks, exploit the L1 cache data to retrieve information of other processes. Therefore, we can migrate the attacking process to some other CPU. Since different processors have different L1 cache, the attacking process can no longer access the L1 cache of the victim process.
The second solution is reducing the priority of the attacking process to slow down its execution. We can handle this by changing the dynamic priority of the process. If a process is identified as attacking its dynamic priority is reduced, so that it gets scheduled late and its dynamic time slice for execution is also small. Since the attacking process is rarely running, it won't be able to retrieve data from cache.

DISCUSSION

We divided the project work into 3 major steps, data collection, data visualisation and heuristic learning, and modifying kernel code. We discuss the above steps in detail in this section:

Data Collection

We started with acquiring several processes which will be used for collecting HPC data. We took different processes from the MiBench collection. The processes that we used are basicmath, bitcount, blowfish, CRC32, dijkstra, FFT, patricia, and sha. These processes comprised the data set for normal programs, i.e, programs that do not do side channel attacks. We used the programs Rowhammer and Meltdown which do side channel attacks. The meltdown contains programs kalsr, reliability, memfiller, secret, and test.

We now read the performance counters corresponding to several runs of these programs. To read these counters, we used the library perftools. To read the different performance statistics we used the following commands:

```
$ sudo perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -o out1.txt proc_name
$ sudo perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores -o out2.txt proc_name
$ sudo perf stat -e dTLB-loads,dTLB-load-misses -o out3.txt proc_name
```

The first command reads the number of cycles, instructions, cache references, cache-misses and bus cycles during the execution of the program named proc_name. The second command gives the L1 data cache misses, L1 data cache loads and stores, L1 data cache miss percentage. Similarly, the third command reads the dTLB loads, dTLB misses, and dTLB miss percentage. We stored all these values corresponding to several runs of each program. The next step is to visualise these values to figure out the best suited features for heuristic learning.

Data Visualisation and Heuristic Learning

One of the most important steps in the whole project is to learn and visualise how different are the performance counters for different processes. We need to visualise how the values are different for processes that do side channel attacks and the ones that don't.
We plotted the different performance counter statistics that we read in the first step of data collection for different programs, in their several runs. We show the plots of number of runs v/s 3 different performance statistics for the processes in figures 1, 2, and 3:

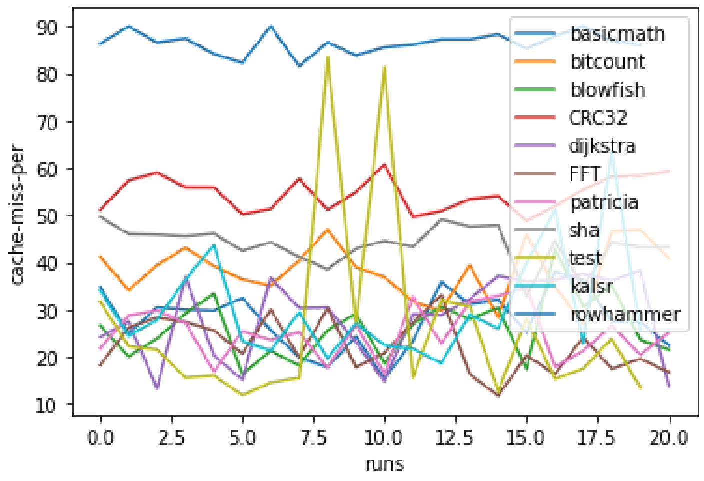


Figure 1: Cache Miss %

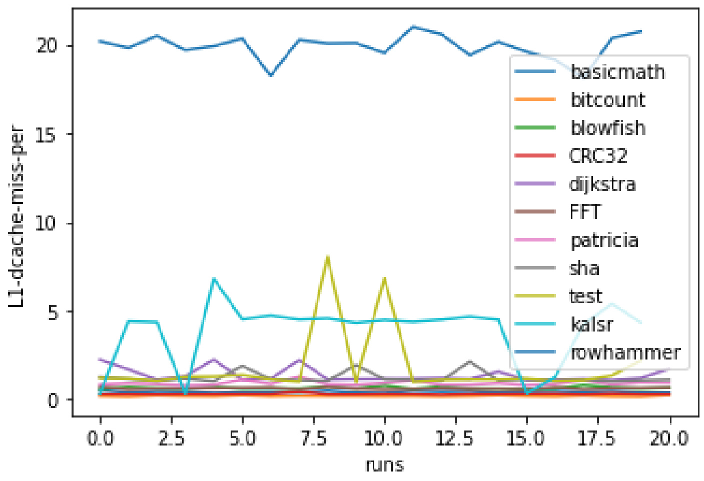


Figure 2: L1 dCache Miss %

We observe that Rowhammer program has the highest number of cache miss %, L1 dcache miss % and cache references. The other side channel attack meltdown's two programs kalsr and test are not very different from the normal programs that are not side channel attacks. We also see that L1-dcache miss percent is strikingly different for rowhammer and the other programs making it one of the best features to use in the heuristics. We also see in figure 2 that kalsr and test attacks have higher values of L1 dcache misses in some runs. We see that cache references are also higher for rowhammer and kalsr.

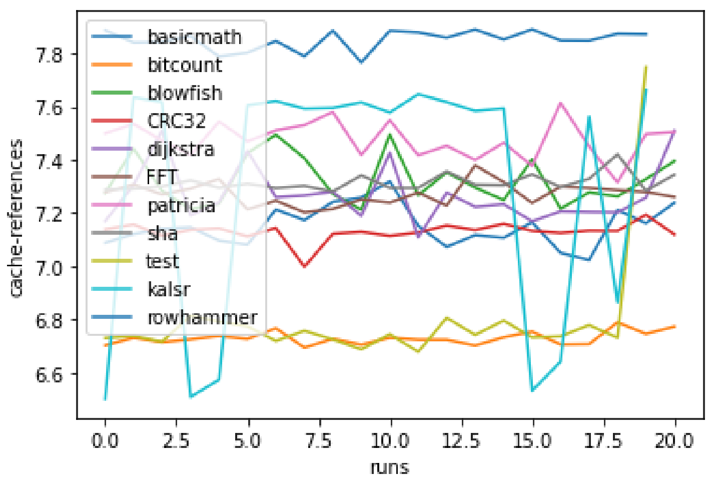


Figure 3: Cache references per unit time

We chose L1 dcache miss % and cache miss % as the two features for heuristics. The first reason for not choosing any other statistics like cache references is that they are time dependent. Here we have shown the plot per unit time. When deploying in the kernel, counting time that a process takes to run in user mode would further complicate the scheduler making it slow, inaccurate and inefficient. The other reason is from the observations that it is not a clearly distinguishing statistic. The values are close for programs doing side channel attacks and the normal programs. It is in good sense to drop such features. We use the `Select-K-Best` feature from the `sklearn` library to find out the most important performance characteristics amongst all. In the figure 4 we can see the scores of each feature. We see that L1 dcache miss percentage is the feauture with highest score. Therefore, our motivation to use it is justified.

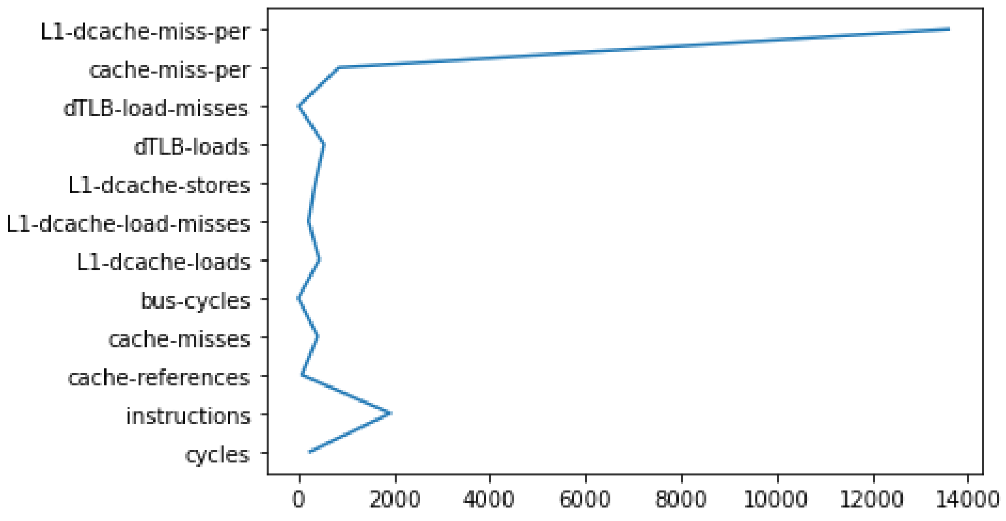


Figure 4: Feature importance

The next step involves coming up with a heuristic. Instead of coming up with a heuristic of our own, we took a clever and more accurate approach of learning the heuristic using machine learning techniques. We used a SVM classifier with linear kernel and trained it for a

2-dimensional data set. The 2 features were cache miss % and L1 dcache miss % as discussed above. We did not involve meltdown programs in our training data set as their execution is not very distinguishable from the normal programs. We trained the SVM over our dataset by labelling the values corresponding to normal programs as class 0 and the side channel attacks as class 1. The decision boundary obtained and few randomly chosen points in the data set is shown in the figure 5. The magenta points correspond to class 0 and yellow points correspond to class 1. The axes represent the statistics as labelled. Since, the data is very clearly linearly separable, we got a validation accuracy of 100%.

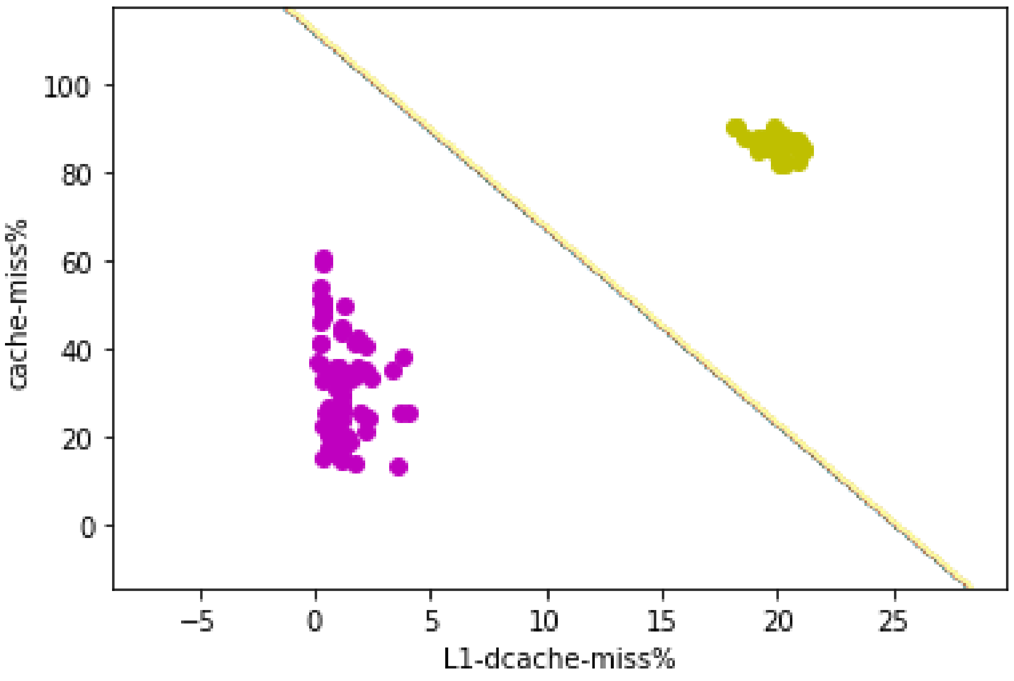


Figure 5: Decision boundary for classification

The equation of line shown in figure 5 is $5x + y - 120 = 0$. Using this we get the heuristic to be used in the kernel

$$f(x) = \begin{cases} \text{side channel attack,} & \text{if } 5x + y - 120 > 0. \\ \text{not a side channel attack,} & \text{otherwise.} \end{cases} \tag{1}$$

where x is L1 dcache miss % and y is cache miss %.

Kernel Development

The changes made to the kernel's scheduler are based on a relatively simple idea.
The region of interest for any process is the time when it has just finished execution and we are about to pick a new process for execution.
Hence most of the logic changes that have been incorporated into the scheduler are limited to the function `context_switch()` in `kernel/sched/core.c`.
The whole procedure of finding an attacking process and taking counter measures against it are summarized as below -

1. When we enter the function `context_switch()` we come with information of the previous running process with a pointer `prev` to it and of the next process that is selected to be executed with a pointer `next` to it.

2. At this stage we check whether the previous process has shown attacking behaviour in its past execution.
   This is done by reading the required performance counter values now and comparing it with the previous recorded performance counter values. By finding the difference between these we will know the performance counter values corresponding to the latest execution of the previous process.

3. To find the value of the required performance counters, we have defined our own functions. All they do is to give us the current value stored in the performance counter of our choice. These functions internally call `write_msr()` and `read_msr()` which have also been defined by us.
   The `write_msr()` and `read_msr()` further use `rdmsr` and `wrmsr` which is predefined by the kernel in <asm/msr.h>. For further information on the usage of `rdmsr` and `wrmsr` refer to the Intel Software Development Manual Volume 3 Chapter 19.

4. We also need to store the previous recorded value of the performance counters. For this we have defined a new `struct proc_msr` in `include/linux/sched.h`. This stores the performance counters that are relevant to us from the previous execution. We can easily extend this structure to hold even more performace counters if required. Hence a new field `struct proc_msr proc_msr_` is added to the `struct task` so that every process now has a `proc_msr_` field.

5. We then use our `predict_is_attack()` function to predict based on the performace counters whether the just executed process is of attacking nature or not.
   For this we use the `hypothesis()` function which uses our learned heuristic to classify the process as attacking or not based on the 2 features we considered before.

6. If we find the process to be attacking we reduce its priority. This is done by increasing its dynamic priority which will automatically affect the time it gets to execute on the processor.
   This will also slow down its execution as it will get to execute on the processor less often.

7. Now before the new process is set to execute, we populate its `proc_msr_` so that we can have the previous counter values to compare to when the process returns back to the scheduler after its execution.

In the next section we show the implementation and code modifications made to the kernel.

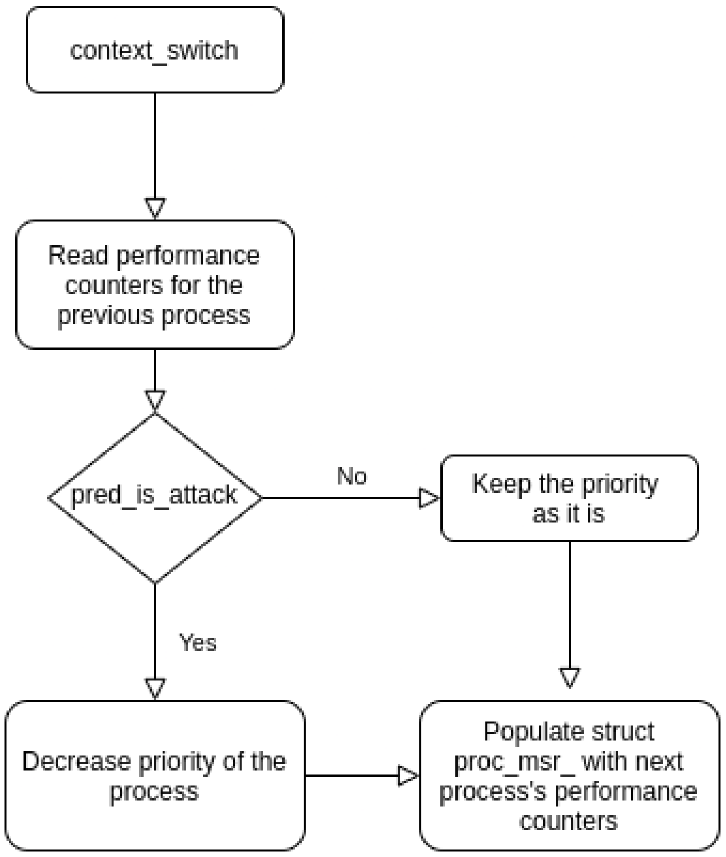

context_switch
Read performance counters for the previous process
pred_is_attack
No
Keep the priority as it is
Yes
Decrease priority of the process
Populate struct proc_msr_ with next process's performance counters

Figure 6: Flow Chart for Kernel Implementation

MATERIALS AND TOOLS

Code Script to Collect Data

The code used to collect performance counter data for various programs has been included in the submission folder. Filename - `data_collector.cpp`

Short Description

The code is a `C++` program that runs tests on the following class of programs -

1. MiBench collection - A representative of non-attacking benchmark programs.
2. Rowhammer - A program that does the standard row hammer attack.
3. Meltdown - A set of programs that do different side channel attacks.

We used the perftools API to read the various performance counters for each of the programs. All the programs were run for 20 - 40 times and each run was for some random time less than 1 second (if it took longer to execute otherwise it ran till completion). The data collected was written into files that were made available for further analysis. This data is stored in the directory `train_data`. It has 3 subdirectories, one each for Meltdown data, MiBench data, and Rowhammer data.

Scripts for Data Visualisation and Heuristic Learning

Data Extraction and Visualisation

The code used to extract information from the output of perftools files and organising is in the jupyter notebook `gendata.ipynb` included in the submission folder. It is a python file that reads all the scripts and stores the results for several different programs into a dataframe. This file also handles the data visualisation. We used modules like numpy, pandas and matplotlib for easy handling and visualisation of data. This file finally stores the final 2 performance statistics for each file, i.e, L1 dcache miss % and cache miss % into a pandas dataframe. We also created a dataframe for the target classes corresponding to each entry in the aforementioned dataframe.

Heuristic Learning

The code used to train the `SVM` model is in the jupyter notebook `training_script.ipynb` in the submission folder. This script trains the linear kernel SVM model and provides us with the heuristic line as shown in figure 5.

Kernel Code

We show here the lines of code added to the linux 4.19.160 kernel.
For each snippet we show the line number and the file to which code was added.
The full kernel is zipped and available in the submission folder.

The following code defines macros that are used for setting the MSR events.

Listing 1: MACROS used for MSR Events
Line 2786 : kernel/sched/core.c

```
#define L1_HIT 0x004101d1
#define L2_HIT 0x004102d1
#define L3_HIT 0x004104d1
#define L1_MISS 0x004108d1
#define L2_MISS 0x004110d1
#define L3_MISS 0x004120d1
#define L1D_HIT 0x00410143
#define LLC_REF 0x00414f2e
#define LLC_MISS 0x0041412e
#define MSR_BASE 0x00000186
```

The following code reads and writes in the MSRs. The function `write_msr` takes two arguments, the first is the MSR that is being used as the counter and second is the value to be written in the MSR so that it points to the corresponding event to that value. The function `read_msr` takes in only one argument which is the MSR that is to be read. Both of them use predefined macros defined in `asm/msr.h`. Since the counter values are big integers the result is returned in two 32 bit registers, `lo` and `hi`. We have used these to pass the values to rdmsr and wrmsr as can be seen in the code below.

Listing 2: Functions to read and write into MSRs
Line 2798 : kernel/sched/core.c

```

#include <asm/msr.h>

static void write_msr(u32 msr, u64 val) {
    u64 lo = val & 0xffffffff;
    u64 hi = val >> 32;
    wrmsr(msr, lo, hi);
}

static u64 read_msr(u32 msr) {
    u64 hi, lo;
    rdmsr(msr, lo, hi);
    return ((u64) lo) | (((u64) hi) << 32);
}
```

The following code just has functions that are used to read the different MSRs that we have used.

Listing 3: Functions to obtain values for specific events
Line 2810 : kernel/sched/core.c

```

void init_msr(void) {
    write_msr(MSR_BASE, L1_HIT);
    write_msr(MSR_BASE + 1, LLC_REF);
    write_msr(MSR_BASE + 2, L1_MISS);
    write_msr(MSR_BASE + 3, LLC_MISS);
}

u64 find_msr_value(int counter) {
    return read_msr(MSR_BASE + counter);
}

u64 find_L1_HIT(void) {
    return read_msr(MSR_BASE);
}


u64 find_LLC_REF(void) {
    return read_msr(MSR_BASE + 1);
}
```

```

u64 find_L1_MISS(void) {
    return read_msr(MSR_BASE + 2);
}


u64 find_LLC_MISS(void) {
    return read_msr(MSR_BASE + 3);
}
```

The following code is the definition of `struct proc_msr`. This stores the performance counter statistics when the execution of the process started.

Listing 4: struct proc_msr defined in include/linux/sched.h
Line 597 : include/linux/sched.h

```

struct proc_msr {
    u64 l1_hit;
    u64 l2_hit;
    u64 l3_hit;
    u64 llc_ref;
    u64 l1_miss;
    u64 l2_miss;
    u64 l3_miss;
    u64 llc_miss;
};
```

The function below calls the functions that we defined earlier and stores the corresponding counter values in the `struct proc_msr`.

Listing 5: Function to populate the struct proc_msr of a task
Line 2853 : kernel/sched/core.c

```

void set_proc_msr(struct proc_msr* p) {
    p->l1_hit = find_msr_value(0);
    p->llc_ref = find_msr_value(1);
    p->l1_miss = find_msr_value(2);
    p->llc_miss = find_msr_value(3);
}
```

The following function **hypothesis** has two arguments, x corresponding to L1 dcache miss % and y corresponding to cache miss %. It predicts whether these values are from a potentially attacking process or not. It uses the heuristic we found earlier to predict the result. It returns 1 if it is an attacking process or 0 otherwise.

Listing 6: Hypothesis function obtained from learning and heuristics
Line 2877 : kernel/sched/core.c

```c
u32 hypothesis(u64 x, u64 y) {
    if(5 * x + y - 120 > 0) {
        // attacking process
        return 1;
    }
    else {
        // normal process
        return 0;
    }
}
```

The following function calculates the values for different performance statistics during the execution of the program. It takes in the pointer to `struct proc_msr` before the execution started and after it finished execution and some other process is scheduled. After that it calculates the percentage values and passes the appropriate parameters to the hypothesis function explained above.

Listing 7: Function to predict if current process is an attack
Line 2887 : kernel/sched/core.c

```c
// return 1 if dangerous else return 0
u32 predict_is_attack(struct proc_msr* prev, struct proc_msr* curr) {
    // difference
    // l1_cache_miss / l1_cache_miss + l1_cache_hit
    // (total_cache_miss) / total_cache_hit + total_cache_miss

    u64 l1_hit = curr->l1_hit - prev->l1_hit;
    u64 llc_ref = curr->llc_ref - prev->llc_ref;

    u64 l1_miss = curr->l1_miss - prev->l1_miss;
    u64 llc_miss = curr->llc_miss - prev->llc_miss;

    u64 l1_miss_rate = div64_u64(l1_miss * 100 , (l1_miss + l1_hit));
    u64 total_miss_rate = div64_u64((llc_miss * 100) , (llc_ref));
    return hypothesis(l1_miss_rate, total_miss_rate);
}
}
```

The following function just picks some other CPU than the current CPU so that we can migrate the attacking process to that CPU and the attacking process doesn't affect the L1 cache for this CPU.

Listing 8: Get the next cpu for migration (simple round robin
Line 2911 : kernel/sched/core.c)

```c
// assuming we have 4 cpus
int __get_next_cpu(unsigned int curr_cpu) {
    return (curr_cpu + 1) % 4;
}
```

The below function `__msr_priority_change()` is used to reduce the priority of the process. It does some important checks which include checking that the process is not a kernel thread, if it is then it returns immediately.
It also checks that the process is not a real time process before attempting to reduce its priority. It increases the priority value of the process only after every 10 context switches (given by the sum `prev->nvcsw + prev->nivcsw`).
This decision ensures that the priority of the process does not become low too soon, but at the same time punishes a process that shows attacking behaviour too often.

Listing 9: Function To Change the priority of the process
Line 2918 : kernel/sched/core.c

```c
void __msr_priority_change(struct task_struct* prev) {
    // use these lines if migrating process to other cpu
    // int new_cpu = __get_next_cpu(prev->cpu);
    // put_prev_task(rq, prev);
    // migrate_task_rq_fair(prev, new_cpu);

    int curr_prio;
    char* kthread = "kthread";
    if(strstr(prev->comm, kthread) != NULL) {
        return;
    }
    curr_prio = prev->static_prio;
    if(curr_prio > MAX_RT_PRIO && curr_prio < 140) {
        // reweight_task(prev, new_prio - 100);
        if(prev->pid > 100000) {
            if(prev->prio < 135 && prev->static_prio < 135) {
                // printk("%s", prev->comm);
                if((prev->nvcsw + prev->nivcsw) % 10 == 9) {
                    if(prev->prio >= prev->static_prio) {
                        prev->prio = prev->prio + 5;
                        prev->static_prio = prev->prio;
                    }
                    else {
                        prev->static_prio = prev->static_prio + 5;
                        prev->prio = prev->static_prio;
    }   }   }   }   }
}
```

We make a few changes in the existing function `context_switch()`. We check using the function `predict_is_attack` whether the past execution of the previous task was potentially an attack. If it is an attack then we increase its priority value (reduce its priority) by calling `__msr_priority_change()`.
When we are picking up the next task to schedule we set the `struct proc_msr` using the function `set_proc_msr` to the current counter values.

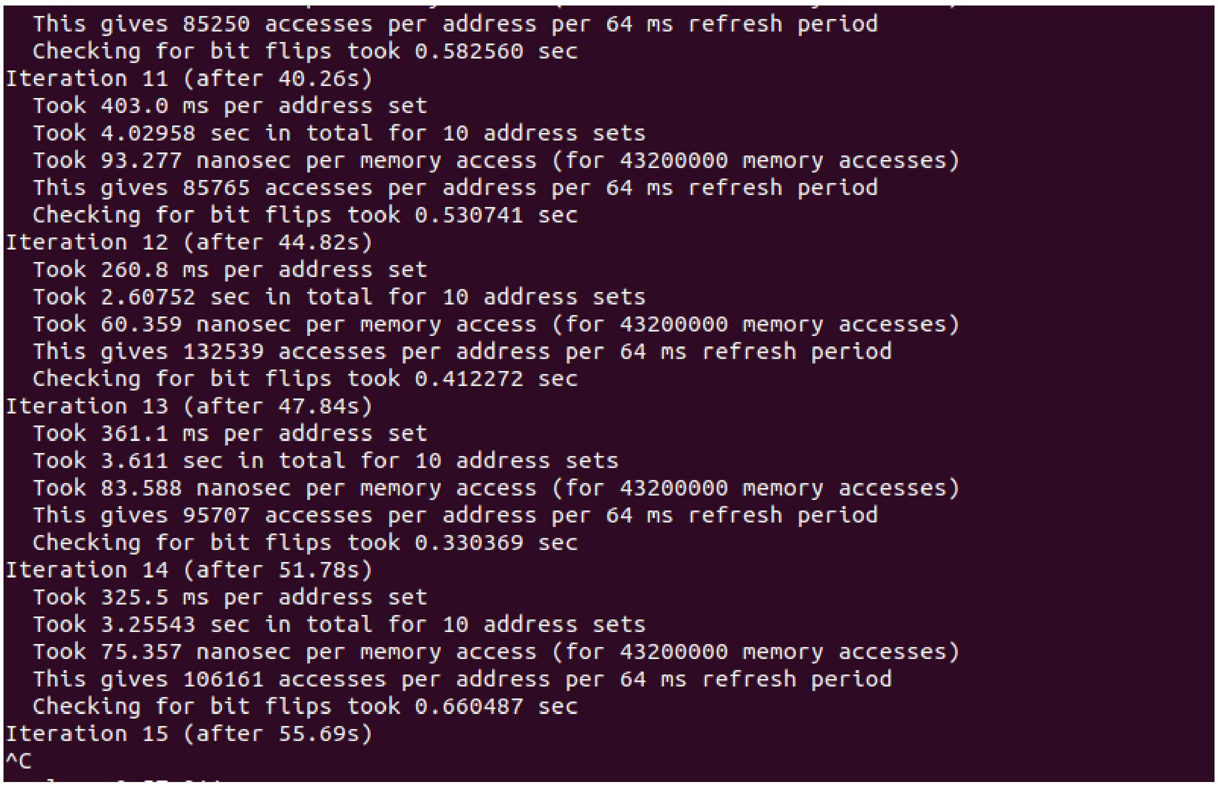
Listing 10: Checking the previous process in context_switch
Line 2951 : kernel/sched/core.c

```
 /*
 * context_switch - switch to the new MM and the new thread's register↩
     state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev,
              struct task_struct *next, struct rq_flags *rf)
{
    struct mm_struct *mm, *oldmm;
    struct proc_msr curr_msr;
    u32 pred;
    bool msr_reweight;
    int curr_prio, new_prio;
    /**changes start here**/
    if(prev->pid > 100000) {
        set_proc_msr(&curr_msr);
        pred = predict_is_attack(&prev->proc_msr_, &curr_msr);
        prev->proc_msr_ = curr_msr;
        msr_reweight = false;
        if(pred) {
            // do something, either migrate or change vruntime
            msr_reweight = true;
        }
        if(msr_reweight) {
            __msr_priority_change(prev);
        }
    }
    /**changes end here**/
    ...
    /**changes start here**/
    if(next->first_exec == 0) {
        init_msr();
    }
    next->first_exec = 1;  // this is the 1st execution
    set_proc_msr(&next->proc_msr_); // set current values of the counters, to be compared with when it comes back from
                                  //  execution
    /**changes end here**/

    // .... function continues (unchanged)
}
```

RESULTS

We used the `stress` program and ran it along with the `rowhammer` attack program on the modified kernel and on the original `Linux v4.19.160` kernel. We see a significant slowdown when running it on our modified kernel. We show example runs below on both the kernels here as a proof of concept.

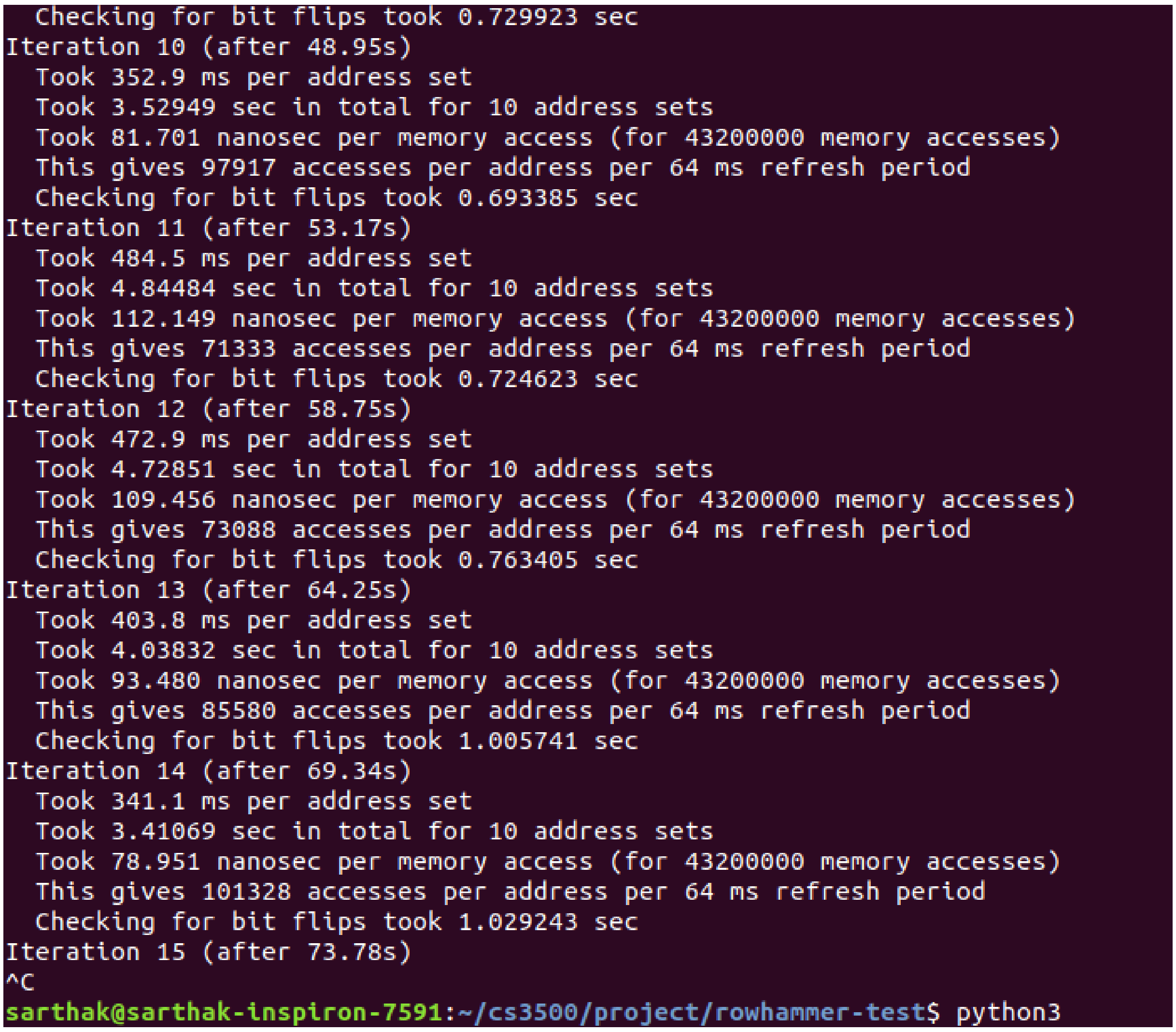


```
  This gives 85250 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.582560 sec
Iteration 11 (after 40.26s)
  Took 403.0 ms per address set
  Took 4.02958 sec in total for 10 address sets
  Took 93.277 nanosec per memory access (for 43200000 memory accesses)
  This gives 85765 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.530741 sec
Iteration 12 (after 44.82s)
  Took 260.8 ms per address set
  Took 2.60752 sec in total for 10 address sets
  Took 60.359 nanosec per memory access (for 43200000 memory accesses)
  This gives 132539 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.412272 sec
Iteration 13 (after 47.84s)
  Took 361.1 ms per address set
  Took 3.611 sec in total for 10 address sets
  Took 83.588 nanosec per memory access (for 43200000 memory accesses)
  This gives 95707 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.330369 sec
Iteration 14 (after 51.78s)
  Took 325.5 ms per address set
  Took 3.25543 sec in total for 10 address sets
  Took 75.357 nanosec per memory access (for 43200000 memory accesses)
  This gives 106161 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.660487 sec
Iteration 15 (after 55.69s)
^C
```

Figure 7: Running Rowhammer on original Linux v4.19.160 kernel

The figure 7 shows the result of running rowhammer program on the original Linux v4.19.160 kernel. As we see in the figure it takes 55.69 seconds to run 15 iterations. Therefore, it takes 3.712 seconds per iteration.

The figure 8 shows the result of running rowhammer program on our modified Linux v4.19.160 kernel. As we see in the figure it takes 73.78 seconds to run 15 iterations. Therefore, it takes



```
  Checking for bit flips took 0.729923 sec
Iteration 10 (after 48.95s)
  Took 352.9 ms per address set
  Took 3.52949 sec in total for 10 address sets
  Took 81.701 nanosec per memory access (for 43200000 memory accesses)
  This gives 97917 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.693385 sec
Iteration 11 (after 53.17s)
  Took 484.5 ms per address set
  Took 4.84484 sec in total for 10 address sets
  Took 112.149 nanosec per memory access (for 43200000 memory accesses)
  This gives 71333 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.724623 sec
Iteration 12 (after 58.75s)
  Took 472.9 ms per address set
  Took 4.72851 sec in total for 10 address sets
  Took 109.456 nanosec per memory access (for 43200000 memory accesses)
  This gives 73088 accesses per address per 64 ms refresh period
  Checking for bit flips took 0.763405 sec
Iteration 13 (after 64.25s)
  Took 403.8 ms per address set
  Took 4.03832 sec in total for 10 address sets
  Took 93.480 nanosec per memory access (for 43200000 memory accesses)
  This gives 85580 accesses per address per 64 ms refresh period
  Checking for bit flips took 1.005741 sec
Iteration 14 (after 69.34s)
  Took 341.1 ms per address set
  Took 3.41069 sec in total for 10 address sets
  Took 78.951 nanosec per memory access (for 43200000 memory accesses)
  This gives 101328 accesses per address per 64 ms refresh period
  Checking for bit flips took 1.029243 sec
Iteration 15 (after 73.78s)
^C
sarthak@sarthak-inspiron-7591:~/cs3500/project/rowhammer-test$ python3
```

Figure 8: Running Rowhammer on modified Linux v4.19.160 kernel

4.918 seconds per iteration. This implies that there is a slowdown of approximately 32.5% per iteration.

CONCLUSION

The project makes it clear that the execution is different and distinguishable for processes that are attacking and non-attacking. We can use HPCs to make this distinction and identify attacking programs. L1 cache miss rate is one of the most important features in this classification. We also included total cache miss rate in our heuristic to be more precise. We came up with a linear heuristic which is not complicated and does not increase the overheads in kernel very much. The attacking programs can be handled by either migrating the process to a different core or reducing its priority.

We have shown in the results section that reducing the dynamic priority of attacking process slows it down significantly. Further developments and fine tuning can help improving the slowdown and in better detection.

There were several other shortcomings which limited the deployment of our work. Our local systems contain only 4 performance counters and storing all necessary events in only 4 of them at one runtime limits our analysis. These registers are not available to read from user processes (`rdmsr` and `wrmsr` can only be used from inside the kernel) and debugging kernel events is tricky and time consuming.

References

1. Professional Linux Architecture by Wolfgang Maurer
2. Linux Kernel Development by Robert Love
3. Intel Software Developer Manual Vol.3
4. Reference For Hardware Performance Counters
5. Irazoqui Cache Side Channel Attack Exploitability And Countermeasures.pdf