CS3500: OPERATING SYSTEM
COURSE PROJECT

Security Aware Scheduling

Pragyan Chakraborty — CS18B038

Ravi Gupta — CS18B043

1 INTRODUCTION

Security is a major concern in modern applications, especially when a system is shared by many users. Various types of attacks are carried out by malicious processes which, for example can read restricted content like passwords or can change memory content of a location to which it should not have access, and these attacks are on the rise in the past few years.

These processes typically behave differently from normal user processes. By default, the scheduling in Linux systems does not check for the security aspect while scheduling different processes.

In this project we modify the Linux Scheduler to detect a few micro-architectural attacks and reduce the priorities of these processes to slow down the detected attacks. Slowing down the attacks can sometimes be a viable defense mechanism against them.

Intel provides a set of Performance Monitoring Registers which can help profile the performance of the programs running on the CPU. Generally the Linux kernel does not make any use of these counters and therefore one can use them to record the performance statistics of the process running on the CPU.

Then based on these statistics we reduce the priorities of the processes that have statistics similar to those of Malware / Attacks.

2 APPROACH

We modified the Linux Scheduler to slow a process down, if detected as malware. To detect a process we used the Performance Monitoring registers provided by Intel to record:

1. num_load : Number of Load Instructions

2. $l1\_miss$: Number of L1 cache misses

3. $l3\_miss$: Number of L3 cache misses

4. $l3\_hits$: Number of L3 cache hits

The version of the processor we were using only had 4 `Model-specific registers (MSRs)`, therefore it only allowed recording of maximum of 4 events. But we also wanted to make use of the Number of L2 cache misses, therefore we estimated it as follows:

$$l2\_miss = \sqrt{l1\_miss \cdot l3\_miss}$$

Through our profiling we discovered that these 5 values are sufficient to detect an attack.

2.1 Profiling Attacks and Benchmarks

In this section we will profile the attacks and the tests in `mibench` benchmark to figure out the correct thresholds to detect the attacks. The numbers presented are mean over three runs of the process.

2.1.1 Attacks

1. **Rowhammer Attack : 12 Iterations** (repo)
   a) num_load : 6,52,92,50,833
   b) $l1\_miss$: 62,48,92,472
   c) $l3\_miss$: 62,24,81,092
   d) $l3\_hits$: 6,01,746

2. **Meltdown Attack (** `./test` **)** (repo) (all meltdown attack tests are from this repo)
   a) num_load : 1,55,10,827
   b) $l1\_miss$: 11,25,594
   c) $l3\_miss$: 10,16,300
   d) $l3\_hits$: 41,830

3. **Meltdown Attack (** `./kaslr` **)**
   a) num_load : 33,46,56,788
   b) $l1\_miss$: 1,26,71,261
   c) $l3\_miss$: 1,04,29,189
   d) $l3\_hits$: 7,57,606

4. **Meltdown Attack (** `./secret` **)**

a) num_load : 31,15,48,609

b) $l1\_miss$: 11,68,377

c) $l3\_miss$: 10,12,974

d) $l3\_hits$: 59,511

5. **L1 covert channel attack(** `./receiver`**)** (repo)

   a) num_load : 7,80,43,92,948

   b) $l1\_miss$: 2,96,69,787

   c) $l3\_miss$: 2,83,326

   d) $l3\_hits$: 10,30,719

6. **L1 covert channel attack(** `./sender` **)**

   a) num_load : 80,22,03,09,969

   b) $l1\_miss$: 4,01,251

   c) $l3\_miss$: 35,386

   d) $l3\_hits$: 2,38,331

2.1.2 MiBench

1. **automotive/basicmath_large**

   a) num_load : 1,05,51,50,718

   b) $l1\_miss$: 2,27,61,294

   c) $l3\_miss$: 6,532

   d) $l3\_hits$: 4,71,564

2. **automotive/qsort_large**

   a) num_load : 2,88,10,255

   b) $l1\_miss$: 4,93,580

   c) $l3\_miss$: 65,705

   d) $l3\_hits$: 2,58,799

3. **Consumer/jpeg-large**

   a) num_load : 3,51,29,837

   b) $l1\_miss$: 1,68,396

   c) $l3\_miss$: 5,819

   d) $l3\_hits$: 22,283

4. **Office/StringSearch-large**

a) num_load : 2,61,213

b) $l1\_miss$: 11,751

c) $l3\_miss$: 1,786

d) $l3\_hits$: 3,553

5. **Network/Dijkstra-large**

a) num_load : 7,80,43,92,948

b) $l1\_miss$: 2,96,69,787

c) $l3\_miss$: 2,83,326

d) $l3\_hits$: 10,30,719

6. **telecomm/gsm-large**

a) num_load : 30,27,93,349

b) $l1\_miss$: 1,17,348

c) $l3\_miss$: 19,360

d) $l3\_hits$: 30,647

7. **security/sha**

a) num_load : 1,18,75,612

b) $l1\_miss$: 13,420

c) $l3\_miss$: 3,894

d) $l3\_hits$: 3,057

2.2 Inferring Thresholds

Now we will use the above profiles to differentiate between the MiBench programs and the attacks. First let's look at some relevant ratios in the following 2 tables. Let $total\_cache\_misses = l1\_miss + l2\_miss + l3\_miss$. Also, let

$$r_1 = \frac{num\_loads}{total\_cache\_misses}$$

$$r_2 = \frac{l3\_misses}{l3\_hits}$$

Now we see that for programs in MiBench $r_2 < 2$ and for the meltdown and rowhammer attacks $r_2 > 13$. So we can say a program is an attack if $r_2 > 5$.

Now for L1 covert channel attack both sender and receiver run simultaneously. We can see that the sender has an incredibly large r_1. So can say that a program is an attack if $r_1 > 5,000$.

For computing the the ratios we use cumulative vales of **num_loads**, **l1_miss**, **l3_miss** and **l3_hits**. We don't declare a process as malware when $l1\_miss < 100,000$ because that means too few samples to get good data.

Process	r_1	r_2
Rowhammer Attack	3.489	1034.46
Meltdown Attack (./test)	4.82	24.30
Meltdown Attack (./kaslr)	9.67	13.77
Meltdown Attack (./secret)	95.29	17.02
L1 covert channel attack(./receiver)	237.559	0.275
L1 covert channel attack(./sender)	144334	0.145

Table 2.1: Statistics of Attacks

Process	r_1	r_2
Basic Math Large	249.604	0.274882
Qsort Large	38.966	0.253884
Consumer - jpeg - Large	170.933	0.261141
Office StringSearch Large	12.0015	1.98936
Telecom - gsm - Large	1455.99	1.58301
Network Dijkstra Large	240.862	1.38996
Security - sha	518.994	0.785054

Table 2.2: Statistics of MiBench programs

Moreover we explicitly ignore process named `kworkers` and `stress`. This is because `kworkers` are kernel threads and we don't want to slow them down, and `stress` is the program we are using as a dummy background task. We don't want to slow that down either.

2.3 Slowing the attack down

Once we infer a process is an attack, we start to increase it's priority by 3 after every 16 context switches. Moreover we also start to increase a parameter process specific parameter `sched_factor` by 50 after every 2 context switches. We then update the $vruntime$ as follows

```
if (p is a normal process)
    p->se.vruntime +=  delta_fair
else if (p is infered as an attack)
    p->se.vruntime +=  p->se.sched_factor * delta_fair
```

So in affect both of these changes slow down a process if it is repeatedly detected as an attack.

3 Results

Now we will show that the modification in the kernel indeed do lead to a slow down of the attacks. We compare the runtime of processes in the modified with its runtime in the default kernel and find the change. We compare the change in the execution times of MiBench on the modified kernel with the change in the execution times of the attacks.

We have recorded all the times with the a dummy stress program running in the background.

```
$stress -i 4 -m 4 -c 4
```

The aim of this stress is to ensure that the CPU always remains busy, and any changes made to the priority of the processes are reflected in their execution time.

All of the times reported are average over 10 successive runs of the program.

Process	**Time in Default Kernel (in seconds)**	**Time in Modified Kernel (in seconds)**	**% Change**
Basic Math Large	8.8	9.4	6.81%
Qsort Large	1.15	1.24	7.83%
Consumer - jpeg - Large	0.12	0.11	-8.33%
Office StringSearch Large	0.051	0.052	1.96%
Telecom - gsm - Large	0.63	0.67	6.35%
Network Dijkstra Large	0.131	0.134	2.29%
Security - sha	0.096	0.098	2.08%

Table 3.1: MiBench program times in Default and Modified Kernels

The above table shows that for normal programs there is minimal performance drop, with a maximum performance drop of 7.83%.

Process	**Time in Default Kernel**	**Time in Modified Kernel**	**Performance Drop**
Rowhammer Attack (20 iterations)	60 seconds	105 seconds	75%
Meltdown Attack (./test)	0.62 seconds	>5 minutes	-
Meltdown Attack (./kaslr)	4.3 seconds	>5 minutes	-

Table 3.2: Time taken by attacks to execute in Default and Modified Kernels

While we can't measure the exact runtime for the L1 cache covert attack, we did find the program to be noticeably slower with the modified kernel.

In case of meltdown attacks, we notice that the processes almost stop working with no progress even after a span of 5 minutes in the modified kernel whereas they take < 5 seconds to complete execution in the default kernel.

For the case of rowhammer attack also we notice a very significant slowdown by 75%, although it is not as dramatic as the slowdown for meltdown attacks.

4 Contributions

Most of the ideas that are implemented were shared ideas. While working on the project, we both used to be on a meet almost always. We also wrote the report together and made equal contributions to it.

Ravi created the slideshow used in the video presentation. He did the user space profiling of the various attacks and benchmarks.

Pragyan installed the version of the Linux kernel used in the project on his laptop and thus made most of the kernel modifications.

Overall we feel the work was evenly shared.

5 Appendix

5.1 Profiling the processes

We use the `perf` tool to profile programs in the user space. One can either use `perf` command line utility or the `perf_events_open` C API for this purpose in the user space.

5.1.1 User Space: Using the perf_events C API

The following C Code access the performance counters using the `perf` wrapper and prints the results. Here we have sampled statistics for number of cpu cycles, number of cache references, number of cache misses and number of branch instructions.

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>
#include <sys/ioctl.h>
#include <linux/perf_event.h>
#include <linux/hw_breakpoint.h>
#include <asm/unistd.h>
#include <errno.h>
#include <stdint.h>
#include <inttypes.h>
#include <time.h>

struct read_format {
  uint64_t nr;
  struct {
    uint64_t value;
    uint64_t id;
  } values[];
};

int main(int argc, char* argv[]) {
  struct perf_event_attr pea;
  int fd1, fd2, fd3, fd4, fd5;
  uint64_t id1, id2, id3, id4, id5;
  uint64_t val1, val2, val3, val4, val5;
  char buf[4096];
  struct read_format* rf = (struct read_format*) buf;
  int i,j;
  struct timespec time, time2;
```

```
time.tv_sec = 1;
time.tv_nsec = 0;

memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HARDWARE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_CPU_CYCLES;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd1 = syscall(__NR_perf_event_open, &pea, 0, -1, -1, 0);
ioctl(fd1, PERF_EVENT_IOC_ID, &id1);


memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HW_CACHE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_CACHE_L1D |
             PERF_COUNT_HW_CACHE_OP_READ << 8 |
             PERF_COUNT_HW_CACHE_RESULT_MISS << 16;;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd2 = syscall(__NR_perf_event_open, &pea, 0, -1, fd1 /*!!!*/, 0);
ioctl(fd2, PERF_EVENT_IOC_ID, &id2);



memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HARDWARE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_CACHE_MISSES;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd3 = syscall(__NR_perf_event_open, &pea, 0, -1, fd1 /*!!!*/, 0);
ioctl(fd3, PERF_EVENT_IOC_ID, &id3);


memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HARDWARE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_BRANCH_INSTRUCTIONS;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd4 = syscall(__NR_perf_event_open, &pea, 0, -1, fd1 /*!!!*/, 0);
ioctl(fd4, PERF_EVENT_IOC_ID, &id4);
```

```c
memset(&pea, 0, sizeof(struct perf_event_attr));
pea.type = PERF_TYPE_HARDWARE;
pea.size = sizeof(struct perf_event_attr);
pea.config = PERF_COUNT_HW_CACHE_REFERENCES;
pea.disabled = 1;
pea.exclude_kernel = 1;
pea.exclude_hv = 1;
pea.read_format = PERF_FORMAT_GROUP | PERF_FORMAT_ID;
fd5 = syscall(__NR_perf_event_open, &pea, 0, -1, -1 /*!!!*/, 0);
ioctl(fd5, PERF_EVENT_IOC_ID, &id5);


ioctl(fd5, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
ioctl(fd5, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);

ioctl(fd1, PERF_EVENT_IOC_RESET, PERF_IOC_FLAG_GROUP);
ioctl(fd1, PERF_EVENT_IOC_ENABLE, PERF_IOC_FLAG_GROUP);

while (1) {
  nanosleep(&time, &time2);

  read(fd5, buf, sizeof(buf));
  for (i = 0; i < rf->nr; i++) {
    if (rf->values[i].id == id1) {
      val1 = rf->values[i].value;
    } else if (rf->values[i].id == id2) {
      val2 = rf->values[i].value;
    }
    else if (rf->values[i].id == id3) {
      val3 = rf->values[i].value;
    }
    else if (rf->values[i].id == id4) {
      val4 = rf->values[i].value;
    }
    else if (rf->values[i].id == id5) {

      val5 = rf->values[i].value;
    }
  }

  read(fd1, buf, sizeof(buf));
  for (i = 0; i < rf->nr; i++) {
    if (rf->values[i].id == id1) {
      val1 = rf->values[i].value;
    } else if (rf->values[i].id == id2) {
      val2 = rf->values[i].value;
    }
    else if (rf->values[i].id == id3) {
      val3 = rf->values[i].value;
    }
    else if (rf->values[i].id == id4) {
      val4 = rf->values[i].value;
    }
    else if (rf->values[i].id == id5) {
```

```c
            val5 = rf->values[i].value;
        }
    }

    printf("cpu cycles: %"PRIu64"\n", val1);
    printf("cache misses: %"PRIu64"\n", val2);
    printf("cache misses2: %"PRIu64"\n", val3);
    printf("branch instructions: %"PRIu64"\n", val4);
    printf("cache references: %"PRIu64"\n", val5);
    printf("\n");
  }

  return 0;
}
```

5.1.2 User Space: Using the perf command line tool

To profile a program which is run using `./program` one can run the the following command on the terminal

```
$ sudo perf stat -e mem_inst_retired.all_loads,
mem_load_retired.l1_miss,mem_load_retired.l2_miss,
mem_load_retired.l3_hit,mem_load_retired.l3_miss ./program

Performance counter stats for './program':

          4,33,034      mem_inst_retired.all_loads   (70.55%)
            14,651      mem_load_retired.l1_miss
             7,410      mem_load_retired.l2_miss
             4,379      mem_load_retired.l3_hit
             2,153      mem_load_retired.l3_miss     (29.45%)

       0.002840371 seconds time elapsed

       0.002829000 seconds user
       0.000000000 seconds sys
```

5.2 Kernel Modifications:

5.2.1 Modifying struct sched_entity

Whenever we wanted any value related to each process we did so by adding additional fields in the `struct sched_entity`. Each process has a `struct task_struct` which stores all of the information regarding the process. `sched_entity` is a member of the `task_struct` which stores all the process specific information regarding scheduling (like `vruntime`). Here are the fields we added in `sched_entity`.

```
struct sched_entity {

    ...

    u64        all_loads;        // Retired loads in the user space of the process
    u64        l1_miss;       //  L1 cache misses
    u64        l2_miss;       //  L2 cache misses
    u64        l3_miss;       //  L3 cache misses
    u64        l3_hits;       //  L3 cache hits
    int        pmc_has_been_set;    // Have the counters been set for the process
    __u32      cycles_acc;          // context switches mod 8
    u64        times_detected;   // times the scheduler detected an anamoly
    u64        sched_factor;        // vruntime += sched_factor * delta_sched_fair

    ...
}
```

The function written below is present in kernel/sched/core.c and is used to update the statistics of a process which is collected in the performance monitoring registers to the fields in the corresponding `task_struct`. This function also checks the stats to classify a process as malicious or otherwise. If malicious, this function increases the `prio` values and weight to update vruntime at regular intervals. This function is called from `__schedule` in kernel/sched/fair.c. As mentioned, this function explicitly does not perform the checks for "kworker" and "stress".

```
#define MAX_SCHED_FACTOR 5000
#define SCHED_FACTOR_INCREASE 50
#define CHECK_PRIO_AFTER_CYCLES 16
#define CHECK_FACTOR_AFTER_CYCES 2
#define MIN_L1_MISS 100000
#define MIN_PID 500
#define MAX_MISS_RATIO 5000
#define MIN_L3_HIT_RATIO 8
#define MAX_PRIO_NUM 139

void __update_msr_stats(struct task_struct* prev)
{
  static const char kworker[] = "kworker";
  static const char stress[] = "stress";

  int reg;
  u64 stats;
  int low,high;
  u64 val;
  u64 l1,l2,l3h,l3m,all;

  if(!pmc_has_been_set_ts(prev))
  {
    set_pmc_ts(prev);
  }

  set_and_read_msr(0x186,ALL_LOADS,0xc1,prev->se.all_loads);
  set_and_read_msr(0x187,L1_MISS,0xc2,prev->se.l1_miss);
  set_and_read_msr(0x188,L3_HITS,0xc3,prev->se.l3_hits);
  set_and_read_msr(0x189,L3_MISS,0xc4,prev->se.l3_miss);
```

```
  l1 = prev->se.l1_miss;
  l3m = prev->se.l3_miss;
  l3h = prev->se.l3_hits;
  l2 = int_sqrt(l1)*int_sqrt(l3m);
  all = prev->se.all_loads;
  prev->se.cycles_acc = (prev->se.cycles_acc+1)%CHECK_PRIO_AFTER_CYCLES;

  if(prefix_compare(prev->comm,kworker))  return;
  if(strcmp(prev->comm,stress)==0)  return;

  if(l1>MIN_L1_MISS && prev->pid > MIN_PID && prev->se.cycles_acc%CHECK_FACTOR_AFTER_CYCES==0)
  {
    if((l1+l2+l3m)*MAX_MISS_RATIO < all)
      prev->se.sched_factor = min_int_msr(prev->se.sched_factor+SCHED_FACTOR_INCREASE,MAX_SCHED_FACTOR);

    else if(MIN_L3_HIT_RATIO*l3h<l3m)
      prev->se.sched_factor = min_int_msr(prev->se.sched_factor+SCHED_FACTOR_INCREASE,MAX_SCHED_FACTOR);
  }

  if(l1>MIN_L1_MISS && prev->pid > MIN_PID && prev->se.cycles_acc==0)
  {
     if((l1+l2+l3m)*MAX_MISS_RATIO < all)
    {
      prev->prio = min_int_msr(MAX_PRIO_NUM,max_int_msr(prev->prio,prev->static_prio)+3);
      prev->static_prio = prev->prio;
    }

    else if(MIN_L3_HIT_RATIO*l3h<l3m)
    {
      prev->prio = min_int_msr(MAX_PRIO_NUM,max_int_msr(prev->prio,prev->static_prio)+3);
      prev->static_prio = prev->prio;
    }
  }
}
```

Given below are some #define which are used in the function mentioned above. The first few define the flags to be passed to MSRs to so that corresponding counters count the required value. `set_and_read_msr` reads the counter values and updates the fields of the struct accordingly.

```
//Flag to count all load instructions from user space
#define ALL_LOADS 0x004181d0
//Flag to count all L1 cache misses from user space
#define L1_MISS 0x004108d1
//Flag to count all L2 cache misses from user space
#define L2_MISS 0x004110d1
//Flag to count all L3 cache misses from user space
#define L3_MISS 0x004120d1
```

```
//Flag to count all L3 cache hits from user space
#define L3_HITS 0x004104d1

#define pmc_has_been_set_ts(ts)
(ts->se.pmc_has_been_set == PMC_SET_FLAG)
#define set_pmc_ts(ts) {\ts->se.pmc_has_been_set = PMC_SET_FLAG;\
                                    ts->se.all_loads = 0; \
                                    ts->se.l1_miss = 0;\
                                    ts->se.l2_miss = 0;\
                                    ts->se.l3_miss = 0;\
                                    ts->se.l3_hits = 0;\
                                    ts->se.times_detected = 0;\
                                    ts->se.sched_factor = 1;\
}\

#define set_and_read_msr(flag_reg,flag,count_reg,field)reg = flag_reg;\
                                              val = flag;\
                                              write_msr(reg,val);\
                                              reg = count_reg;\
                                              rdmsr(reg,low,high);\
                                              stats = ((u64)high<<32)+low;\
                                              field += stats;\
                                              write_msr(reg,0);\
```

After every context switch, we have

$$vruntime+ = delta\_fair$$

Given below is the updated version of the function which calculates delta_fair of a process. This function is present in kernel/sched/fair.c. Here we see that if the `sched_entity` of a process is set, then we use the field `sched_factor` to weight delta_fair. Since in the above function this is regularly incremented for malicious processes, for malicious processes vruntime increases at a higher rate, thus pushing it further

```

static inline u64 calc_delta_fair(u64 delta, struct sched_entity *se)
{
  if (unlikely(se->load.weight != NICE_0_LOAD))
    delta = __calc_delta(delta, NICE_0_LOAD, &se->load);

  if(se->pmc_has_been_set == PMC_SET_FLAG)
    return se->sched_factor*delta;

  return delta;
}
```

Below we have updated version of `__sched_fork` which is used to set the default values in `sched_entity`.

```
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
    ...

  //Setting default values for newly added fields in sched_entity
```

```
    p->se.pmc_has_been_set = PMC_SET_FLAG;
    p->se.all_loads = 0;
    p->se.l1_miss = 0;
    p->se.l2_miss = 0;
    p->se.l3_miss = 0;
    p->se.l3_hits = 0;
    p->se.times_detected = 0;
    p->se.sched_factor = 1;

    ...
}
```