BiasP: A DVFS based Exploit to Undermine Resource Allocation Fairness in Linux Platforms

Harshit Kumar, Nikhil Chawla, Saibal Mukhopadhyay
Dept. of Electrical and Computer Engineering, Georgia Institute of Technology
hkumar64@gatech.edu,nchawla6@gatech.edu,saibal.mukhopadhyay@ece.gatech.edu

ABSTRACT

Dynamic Voltage and Frequency Scaling (DVFS) plays an integral role in reducing the energy consumption of mobile devices, meeting the targeted performance requirements at the same time. We examine the security obliviousness of CPUFreq, the DVFS framework in Linux-kernel based systems. Since Linux-kernel based operating systems are present in a wide array of applications, the high-level CPUFreq policies are designed to be platform-independent. Using these policies, we present BiasP exploit, which restricts the allocation of CPU resources to a set of targeted applications, thereby degrading their performance. The exploit involves detecting the execution of instructions on the CPU core pertinent to the targeted applications, thereafter using CPUFreq policies to limit the available CPU resources available to those instructions. We demonstrate the practicality of the exploit by operating it on a commercial smartphone, running Android OS based on Linux-kernel. We can successfully degrade the User Interface (UI) performance of the targeted applications by increasing the frame processing time and the number of dropped frames by up to 200% and 947% for the animations belonging to the targeted-applications. We see a reduction of up to 66% in the number of retired instructions of the targeted-applications. Furthermore, we propose a robust detector which is capable of detecting exploits aimed at undermining resource allocation fairness through malicious use of the DVFS framework.

CCS CONCEPTS

• **Security and privacy → Malicious design modifications**.

KEYWORDS

Security, power management, DVFS, P-states

ACM Reference Format:
Harshit Kumar, Nikhil Chawla, Saibal Mukhopadhyay. 2020. BiasP: A DVFS based Exploit to Undermine Resource Allocation Fairness in Linux Platforms. In *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '20), August 10–12, 2020, Boston, MA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3370748.3406549

ISLPED '20, August 10–12, 2020, Boston, MA, USA

ACM ISBN 978-1-4503-7053-0/20/08…$15.00
https://doi.org/10.1145/3370748.3406549

1 INTRODUCTION

A diverse group of tasks that fulfill various functionalities operates concurrently on a commercial System-on-Chip (SoC). All these tasks are contending over resources such as cache memory, random access memory, internal buses, and running time on CPU. It is the responsibility of the OS kernel to ensure fair allocation of the hardware and software resources. The scheduler is the kernel entity that handles CPU resource allocation for executing tasks and ensures that each running task gets a small fraction of the CPU resources at regular time intervals [7]. This "fairness" is necessary to warrant that no particular process hogs up the CPU resources and degrades the Quality-of-Service of other tasks [3]. Therefore, processor performance is highly dependent on the extent of fairness imposed by the scheduler. An unfair scheduler has the power of directing resources away from a particular set of tasks degrading their Quality-of-Service (QoS).

The Linux-kernel provides different techniques that can be used for controlling resource allocation in CPU-cores and regulating the execution time of the processes. Control Groups (commonly known as CGroups) allows the user to allocate resources among a user-defined group of tasks running on a system [4]. One can also change the scheduling priority of a process that controls the amount of time the process gets to execute instructions on the CPU core [7]. While these techniques are useful for skewing allocation away from resource-hungry tasks, they can be used to undermine the fairness in resource allocation to regular tasks, degrading their QoS. However, since such techniques have been designed for controlling resource allocation, and they directly influence the scheduler's behavior, they are primarily monitored for any violation in resource allocation fairness [15].

A different kernel entity, called Dynamic Voltage and Frequency Scaling (DVFS) system, aims to find the right balance between performance and power consumption [25]. DVFS reduces the power consumption of a computing device by scaling down the voltage and frequency based on the targetted performance requirements of the task. Due to the efficacy of DVFS power management in reducing both dynamic and static power, it has become ubiquitous in mobile platforms [18].

In this paper, we present BiasP, which exploits DVFS platform to adversely impact the fairness in resource allocation inside commercial SoCs running on Linux-based kernel. We restrict CPU resource allocation to a set of targeted tasks running on the CPU core by modifying the userspace attributes of the DVFS policies. In particular, we use the upper layers of the CPUFreq, the DVFS framework in Linux-kernel, to create a first-of-its kind scenario that undermines microarchitectural fairness through DVFS. Since current schedulers do not monitor the impact of DVFS policies on the computation of a process, it is much more challenging to discover such scenarios

using current detection systems. Hence, we propose a lightweight detector, which is capable of detecting compromises in resource allocation fairness arising from the DVFS framework. We make the following contributions in this paper:

- We formulate a robust task detection methodology, on commercial Linux-kernel based platforms, which is capable of accurately detecting targeted-tasks executing instructions on the CPU-core. We then modify the CPUFreq core attributes to restrict CPU resource allocation to a set of targeted tasks degrading their performance.
- We propose a lightweight detector which is capable of detecting any skewness in resource allocation achieved by exploiting DVFS platforms, including intelligent variations of the proposed technique.

Furthermore, we demonstrate the practicality of our exploit by running it on a commercial smartphone, which runs the Linux-kernel based Android OS. We show that given a list of target applications, the exploit can specifically slow down the processor when those apps are actively executing instructions on the CPU core, defeating the "neutrality" or fairness in allocation of resources to the targeted apps.

The remainder of the paper is structured as follows. We provide a brief background on Linux scheduler and DVFS in Section 2. Next, we explain our threat assumptions and BiasP exploit in Section 3. In Section 4, we demonstrate the exploit by running it on an Android OS-based smartphone. We discuss a detection-technique in Section 5. We discuss related works in Section 6 and conclude in Section 7.

2 BACKGROUND

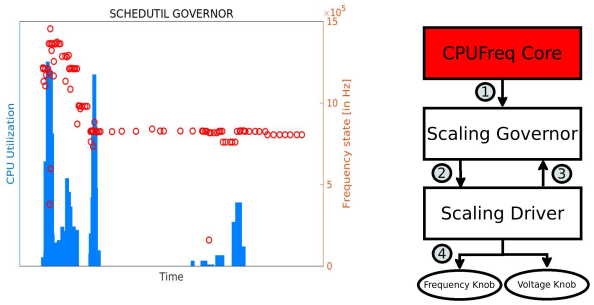
We discuss the function of Linux scheduler and the calculation of CPU load utilization in Section 2.1, followed by the dynamic scaling of voltage and frequency employed in mobile platforms in Section 2.2 and its implementation in Linux-kernel in Section 2.3.

2.1 Linux Scheduler and CPU Utilization

Scheduler is the kernel entity which is responsible for deciding the order of execution of runnable threads on a CPU core. It maintains a list of all runnable threads and selects the thread with the highest priority. Since the scheduler has information about all the runnable threads and their respective demand, it can calculate the CPU utilization at any given time using this information. Various load tracking schemes [14, 20] enable the scheduler to track the aggregate demand from all threads which are in the run queue of the processor. The secondary goal of the scheduler is to inform the CPUFreq governor (Section 2.3) of the aggregate task demand and guide the governor on the need to change the CPU frequency.

2.2 Dynamic Voltage and Frequency Scaling

DVFS is a power-saving technique used in IoT/mobile platforms. It minimizes the amount of energy required to perform each task by scaling the operating voltage and frequency of a CPU core. The frequency and voltage supply values for a given DVFS based power management are discrete points rather than continuously varying values. The discrete tuples of frequency and voltage pairs supported by the system is called Operating Performance Points (OPPs) or P-states. For example, (600MHz, 0.8V) is an OPP. High-level policies are used to optimize the P-state selection process [21].



(a) (b)

Figure 1: (a) Time series of CPU utilization (from scheduler) and frequency states decided by Schedutil governors of Pixel 3.
(b) CPUFreq subsystem: (1) Tunables and modifiable attributes communicated from CPUFreq Core to the scaling governor (2) Scaling Governor transmits decisions about the selected P-states (3) Scaling Driver communicates the available P-states for the hardware platform (4) Scaling Driver writing in the memory mapped registers which control the scaling of supply voltage and clock frequency.

2.3 DVFS in Linux-kernel based platforms

Linux uses a standard infrastructure called CPUFreq (CPU Frequency Scaling) to implement DVFS [5]. CPUFreq dissociates the CPU frequency controlling mechanisms (like device-level drivers) from high-level system policies. The policies are instantiated as "governors," and different kinds of governors exist for various power and performance requirements. Figure 1.b explains the generic structure of the CPUFreq subsystem. In this work, we are only concerned with CPUFreq core which provides the common code infrastructure containing all the tunables and interfaces. It is accessed through the files in: `/sys/devices/system/cpu/cpux/cpufreq/`

CPUFreq Governors We summarize the operation of a few generic scaling governors[1] provided by CPUFreq, which have been considered in this work, and can be used with all scaling drivers of any platform:

- Schedutil : This governor uses the CPU utilization metric from the scheduler. Based on the utilization it scales up the frequency to provide necessary processing power.
- Powersave : This governor statically sets the P-state of the CPU to its lowest point.

Figure 1.a illustrates the frequency[2] and CPU utilization values for the schedutil governor. One can observe that the frequency state is a monotone function of the CPU utilization ,*i.e.*, an increase in the CPU utilization leads to the governor scaling up the frequency.

[1]We only consider the generic governors used in commercial Linux-based systems and not those proposed in the literature.
[2]We only consider the frequency values as representation of the P-state since for a given P-state, there is a one-to-one mapping between the voltage state and the frequency state.

Table 1: % Increase in the number of microarchitechtural events between low and high P-states

Event	*Scenario A*	*Scenario B*	*% Increase*
retired-instructions	417400438	1085778527	160.13
cpu-cycles	496874308	1336644722	169.01
cache-references	168271841	440688625	161.89

3 EXECUTING BIASP

Threat model assumptions. Modifying the attributes of the CPUFreq framework requires privileged access. Hence, BiasP can be executed by embedding it inside kernel-level malwares. Kernel-level malwares are pretty common in Linux based platforms as well as in Android OS (Linux kernel-based) which is considered in this paper [12, 23]. Therefore, the gray box vulnerability testing environment can be used to assess the implications of BiasP.

Motivational example. Consider the following two scenarios of Task X operating on a CPU core with the specified clock frequency:

- Scenario A: Clock frequency of 576 Mhz
- Scenario B: Clock frequency of 1.76 GHz

The number of retired instructions for a given time in scenario A would be a fraction of its value in scenario B. The reduction in clock frequency leads to an increase in cache-hit time, memory-access time, IO wait time, and a general increase in the duration of all microarchitectural events. This essentially translates to a scenario wherein Task X does not have as much access to resources in scenario A as it had in case of scenario B, undermining the objectives of a fair OS kernel. Thus, lowering the P-state of a CPU affects the entire spectrum of microarchitectural events. Table 1 shows a few microarchitectural events and their corresponding increase from Scenario A to B. The data has been collected using performance counters on a Google Pixel 3 smartphone.

Principle. BiasP aims at degrading the performance of specific tasks in a Linux-kernel based platform. Henceforth, the specific tasks which are candidates for the exploit, and whose performance we want to degrade, are referred to as target-task. For achieving this goal, we need to perform the following steps:

(1) Detect when a target-task is running on a CPU core.
(2) If a target-task is detected, we lower the P-state of the CPU core to its lowest possible value.

Since all the other tasks, apart from the target-task, will have P-state allocation proportional to their utilization of CPU core, we are depriving the target task of adequate CPU resources required for satisfactory performance. Next, we state the challenges which need to be surmounted for effective execution of this scenario.

Challenge 1: Robustly detecting when a target-task is running. Context-switching of processes occur at a very high frequency, in the order of μs. Furthermore, for an N core CPU (without hyperthreading [19]), a maximum of N processes can be executed simultaneously, one process per core. Apart from context-switching, a process may migrate to another core. All these factors make robust detection of target-task from the userspace quite challenging.

Challenge 2:Minimizing spill-over performance degradation Upon successful detection of a target-process running on a CPU core, BiasP lowers the P-state of the core. However, changing the P-state (as explained in Section 3.0.2) requires time, and during that interval, the target-process may context-switch in and out of the CPU core multiple times. This leads to spill-over performance degradation of the processes which were executed during this window of lowered P-state and were not the target of the exploit.

Besides this, in modern multi-core processors, CPUFreq governors operate on a per-cluster basis ,i.e., a cluster of cores (> 1) share the same CPUFreq governor, and therefore the same P-state. Therefore lowering the P-state of a cluster will have spill-over degradation for all the non-target-processes which are running on the same cluster. This case of spill-over degradation cannot be circumvented from userspace; therefore, we do not discuss this further. However, in day-to-day practical scenarios, whenever a specific-task is running on a Linux-based mobile platform, we can expect the processes associated with that task to be running most of the time on the CPU-core.

3.0.1 ***Task detection on Linux-kernel based platforms.*** Every process in Linux-kernel has a unique identifier assigned to it, called the process pid. We can obtain this unique pid by using the command: `pidof <task-name>`. pidof returns *NULL* if the task is neither in the running or not-running state, and returns a unique pid otherwise [13]. In case of an Android device we can get the pid of an application by using its publicly available application ID [1].

With the acquired target-pid, the current objective is to detect whenever the target-task starts running on the CPU. We mention the conventional methodologies of detecting actively running tasks on a CPU from userspace in Linux-kernel. Furthermore, we discuss their drawbacks and their inefficacy in robust task-detection.

Method 1: Read from the file `/proc/[pid]/stat`. As explained in the Linux manual, the state field inside the stat file contains the process state [6]. If the state field reads "R," then the process is running.

Drawback. Linux classifies both running and runnable processes as "R." A runnable process has all the resources required for execution and is waiting for a slot on the CPU core; a running process is executing instructions on the CPU. Clearly, lowering down the P-state upon detection of a runnable target-process undermines our goal of minimizing the spillover degradation of the exploit.

Method 2: Read from the file `/proc/sched_debug`. Linux provides global scheduling information in the file `sched_debug`. The field `.curr -> pid` for each CPU core, inside `sched_debug`, stores the pid of the current process executing on that core.

Drawback. `sched_debug` is a large file and parsing it for getting the value of `.cur -> pid` takes time (in the order of *ms*) undermining robust task detection.

Proposed method for task-detection: Use `ftrace`. *ftrace* is a userspace utility for diagnosing and debugging performance issues in the Linux kernel [22]. ftrace uses tracepoint events to track events occurring inside the kernel. The tracepoint event, which is of interest in our case, is `sched:sched_update_task_ravg`. It is logged when the window-based CPU-load stats are updated for a task [9]. Within this event lies the subevent `TASK_UPDATE`. This subevent is invoked when a running-process updates its window-based stats. The process continues to execute after this event until a

context switch occurs. Therefore, this event facilitates robust task-detection and a reduction in spill-over performance degradation. Algorithm 1 states the proposed method for target-detection.

Algorithm 1: TARGET_DETECT returns the number of times target-task is executing instructions on CPU-core

Input: List of target-tasks $TT = \{t_1, t_2, \ldots, t_n\}$ of task-names
Output: # of times target-task executes instructions on a CPU-core

$TT\_PID \leftarrow pidof(TT)$
/*Setup ftrace*/
`echo` TT_PID `> /sys/kernel/tracing/set_event_pid`
`echo "cur_pid==`TT_PID`" && "event==TASK_UPDATE" > /sys/kernel/tracing/events/sched/filter`
/*Enable tracing and clear trace buffer*/
`echo 1 > /sys/kernel/tracing/tracing_on`
`echo > /sys/kernel/tracing/trace`
/*Output the contents of trace buffer into the file t_detect*/
$t\_detect \leftarrow trace$
if *events in* t_detect **then**
  return $\#events$
else
  return $TARGET\_TASK\_NOT\_DETECTED$

Algorithm 2: BIASP lowers the P-state of the system when it detects a target-task executing instructions on the CPU-core

Input: List of target-tasks $TT = \{t_1, t_2, \ldots, t_n\}$ of task-names
Output: Selects lowest P-state from the list $P - state = \{p_1, p_2, \ldots, p_{max}\}$ of available states upon detection of TT

while $TRUE$ **do**
  $\#Target\_task\_occurrence \leftarrow$ TARGET_DETECT(TT)
  /*$threshold$ = 0 for task-detection
  $threshold$ = N for foreground application detection*/
  if $\#Target\_task\_occurrence > threshold$ **then**
    $current\ P\_state \leftarrow p_1$
  else
    $current\ P\_state \leftarrow default\_CPUFreq\_governor()$
return

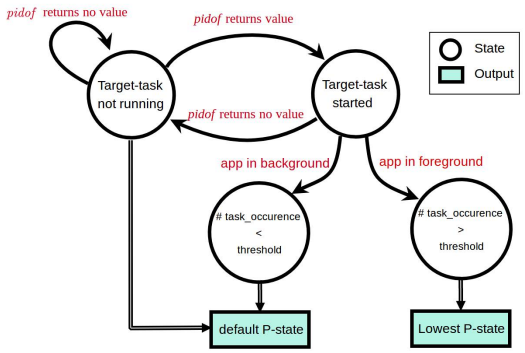


Figure 2: State diagram showing the decision-making process of BiasP

3.0.2 ***Lowering the P-state of the system.*** The second step of BiasP entails lowering the P-state of the CPU core upon successful detection of the target-task. We achieve this by changing the CPUFreq governor to powersave by writing powersave in the file `scaling_governor`. Figure 2 depicts the decision making process during the exploit. The implementation of the scenario is summarized in Algorithm 2.

4 EXPERIMENTAL DEMONSTRATION

For demonstrating the outcomes of BiasP, we select the Android Operating System (AOS). AOS is found in roughly 85% of the smartphones used in the world and is built on a modified Linux kernel.

4.1 Experimental Setup

The target device is a Google Pixel 3 smartphone, which features an octa-core Qualcomm Snapdragon 845 CPU. The four A55 cores share the same CPUFreq governor, and the four A75 cores share the same CPUFreq governor. The device is flashed with the stock kernel for Pixel 3 available from the Android Open Source Project (AOSP), i.e., Android v9.0, which is based on Linux kernel 4.9.148 [2]. During the experiments, the device was connected to the wireless network, and background processes from the OS and other applications were running, creating a real-world scenario.

4.2 Observation Vectors

BiasP results in performance degradation in the victim device. We use the following vectors to quantify the degradation:

- **gfxinfo** is used to measure the frame processing time and the number of dropped frames [11].
- **Performance Counters** to quantify performance degradation by measuring the number of retired instructions [8, 10]

4.3 Exploit Results

To simulate different workloads, we selected the top apps and games frequently downloaded by the Android community. We also use CPU Load Generator (CPU_LG) for creating CPU-bound threads. This helps us in determining the maximum performance degradation, which is achieved by lowering the P-state of the CPU.

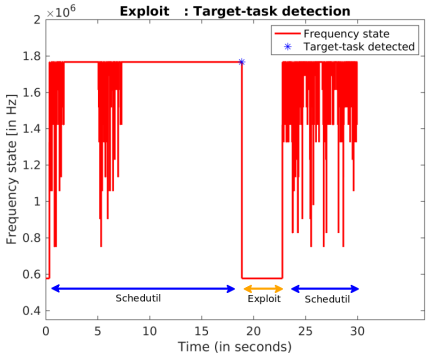


Figure 3: Lowering of P-state upon target-detection

Figure 3 shows BiasP: upon target detection, it lowers the P-state of the CPU-core; otherwise, the P-state is determined by the default governor (schedutil in case of Pixel-3).

Table 2 shows the frame processing time and the number of dropped frames with and without BiasP targeting those apps. There is a significant increase in the frame processing time and the number of dropped frames. Both these factors contribute towards a degraded QoS of the app, which was visibly "laggy". Frame processing on Android OS takes place on CPU, so even if an app has I/O bound processes, there will be degradation in the UI performance.

Figure 4 shows the reduction in the number of retired instructions. For simulated regular use of every app, 20 iterations of data were collected, each iteration of 10 seconds. The retired instructions were averaged over all the iterations. High degradation is observed

Table 2: Frame Processing time (FT) and the number of dropped frames for the baseline and the attack scenario.

Performance Parameter	*Chrome*			*Youtube*		
	Baseline	Exploit	Degradation	Baseline	Exploit	Degradation
Median FT (ms)	5	11	120%	6	14	133%
Average FT (ms)	7	17	143%	7	15	114%
Dropped Frames	24	100	316%	40	84	110%
	Amazon			*Google Maps*		
Median FT (ms)	4	15	275%	5	18	260%
Average FT (ms)	5	15	200%	9	18	100%
Dropped Frames	34	356	947%	263	975	270%

for apps which are CPU intensive, while apps which have I/O bound processes show comparatively lesser degradation in the number of retired instructions.

5 DETECTION

BiasP uses DVFS for effectuating the unfairness, which makes the detection difficult using the current framework that only focuses on cgroups [15]. Moreover, this is not a Denial-of-Service attack where the victim is locked out of the system and detection of the exploit becomes pretty straightforward. BiasP is much more subtle and difficult to detect considering it degrades the QoS of tasks by limiting the resource allocation using a new attack surface, DVFS.

Figure 6 shows the frequency distribution of the P-states observed under two different conditions, i.e., the schedutil governor and BiasP, for a given workload. In case of schedutil governor (in green), there is a heavy bias towards the higher P-states, since performance is prioritized over energy effeciency when the device is active. Under BiasP (in red), there is an unusually high frequency of the lowest P-state. Furthermore, Fig 5 shows the range of CPU utilization values observed for the lowest P-state. We can observe that only low utilization values result in the lowest P-state, as compared to BiasP scenario where a range of utilization values result in the lowest P-state, making the utilization a "don't care". Figure 1.a shows a direct correlation in CPU Utilization and the corresponding P-state. Hence we can conclude that the P-state scaling decision is correlated with the CPU Utilization, except in the case of BiasP. We use this principle to create a correlator based detector for BiasP.

Algorithm 3 explains the proposed detector that monitors the *effect* of BiasP. This enables us to tackle intelligent variations of the exploit, which might involve malicious kernel drivers directly

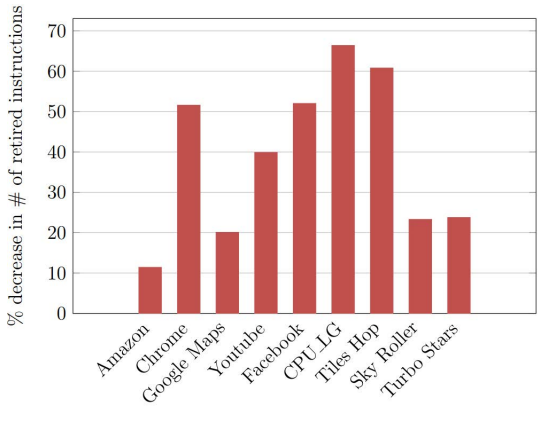


Figure 4: Performance degradation of apps due to decrease in the number of retired instructions

Algorithm 3: Detection Algorithm for BiasP

Input: P-state (P_n) and CPU Utilization (C_n) at current time t_n
Output: BiasP detection flag

```
while TRUE do
    /*P-state queue (P_Queue) of size N*/
    /*Correlation buffer (Corr_Queue) of size M : Stores correlation between
      CPU utilization and P-state*/
    P_Queue.push(P_n);
    corr_n = Correlation(C_n,P_n);
    Corr_Queue.push(corr_n);
    if P_n == Lowest_Pstate then
        if all elements of P_Queue are same then
            if corr_n > average(Corr_Queue) then
                return BiasP Red flag;
```

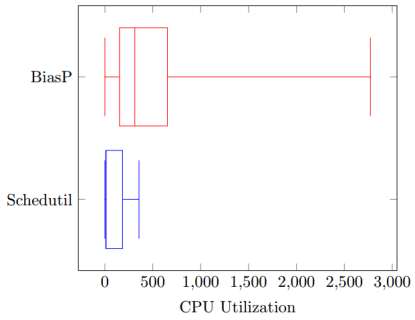


Figure 5: Range of CPU utilization observed in the lowest P-state

tuning the P-state using memory mapped registers, rather than using the CPUFreq framework [24]. We use the lowest P-state as a filter to judge a potential red-flag location. Next, we check if there is a sudden transition in the correlation of the CPU utilization and P-state decision to confirm the red flag. The P-state and the correlation buffer length can be varied to reduce false positives. In our case, we found that a buffer length of 3 was able to eliminate the false positives completely. The entire detector is shown in Figure 7. We tested the proposed algorithm using the test data which was collected by running BiasP on the Pixel 3 device. The dataset consists of 15 iterations of 60s of regular use of the device without the exploit targeting any app, and 15 iterations of the exploit targeting different apps. The detector was able to identify all the exploit locations with no false positives.

Practical challenges: Since BiasP can be carried out by a kernel-level malware, the proposed detector needs to be shielded from the malicious kernel. The logical step to design a protected detector would be to implement it in the hardware. While the frequency

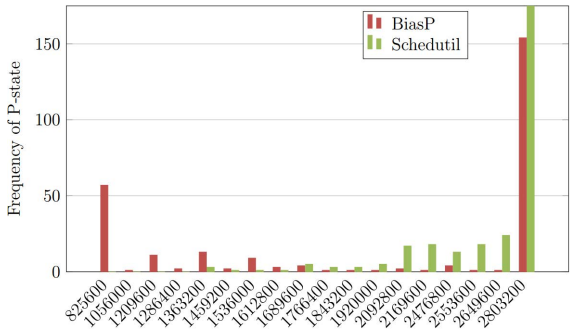


Figure 6: Frequency distribution of P-states under BiasP and Schedutil

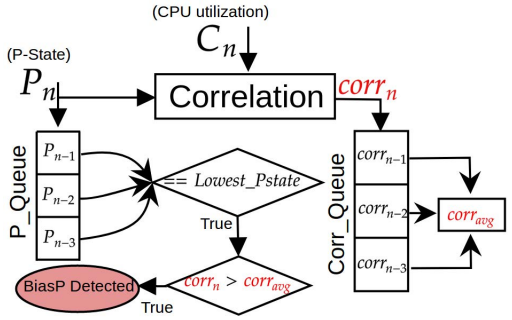


Figure 7: Correlation based detector for BisaP

of the CPU core can be obtained directly at the hardware stack, the CPU utilization used by our detector is calculated at the kernel stack. We need a corresponding measure of utilization at hardware stack which gives us an estimate of the utilization at the kernel stack. Monitoring the frequency of instructions dispatched from the instruction queue can provide us with an estimate of the utilization at the hardware stack. This will be explored in future work.

6 RELATED WORKS

A malicious self-modifying code which flushes the trace-cache of a Simultaneous Multi-Threading (SMT) processor can slow down the victim-thread by a factor of 10-20 [16]. This exploit is only applicable to processors supporting simultaneous multi-threading [19]. Hasan et al. frequently accesses a shared resource to create a hot spot [17]. This results in the power management throttling down the CPU pipeline to cool down the hot spot. The attack was demonstrated on simulations of an SMT processor. The proposed solutions for the prevention of the above attacks do not address our exploit. Unlike [16], we are not unfairly utilizing a shared resource. The selective sedation method proposed to tackle heat stroke-based attacks in [17] operates on the fundamental assumption of a malicious code over-using a hardware resource. We are exploiting the framework which controls the allocation of an essential shared resource, i.e., CPU clock frequency and supply voltage. Therefore, the proposed detectors in [16] and [17] do not consider our exploit.

Gao et al. shows the deficits in cgroups, enabling malicious users to hog system resources and degrade performance of co-resident containers [15]. We present a new attack surface, i.e., DVFS to achieve the same objective. A recent attack CLKscrew exploits the unfettered access of DVFS drivers over the memory-mapped registers which control the CPU clock frequency and supply voltage [24]. The implications of this attack are far more severe than the proposed exploit, however, its baseline assumptions and methodology make it a complicated attack to execute. Different from all previous work, our work shows that DVFS can be used as a backdoor to undermine resource allocation fairness in modern SoCs.

7 CONCLUSION

We demonstrate CPUFreq, the DVFS framework in Linux-kernel based devices, as a potential source of undermining CPU resource allocation. We devise a robust task-detection methodology using the debugging tools, which are pre-installed on all Linux-kernels since version 2.6. As a proof-of-concept, we demonstrate the working exploit on a commercial smartphone which runs a Linux-kernel based OS. The exploit increases the frame processing time of targeted apps by up to 200% and the number of dropped frames by up to 947%, significantly degrading the QoS experienced by the end-user. We show a 10-66% reduction in the number of retired instructions of the targeted apps under consideration. We propose a detector which identifies scenarios where the DVFS framework is being used for undermining fairness in resource allocation.

8 ACKNOWLEDGEMENT

This material is based on work supported by Semiconductor Research Corporation and TxACE under #2810.002

REFERENCES

[1] [n.d.]. *Application ID.* Retrieved November 21, 2019 from https://developer.android.com/studio/build/application-id
[2] [n.d.]. *Building Kernels.* Retrieved November 21, 2019 from https://source.android.com/setup/build/building-kernels
[3] [n.d.]. *CFS Scheduler.* Retrieved November 21, 2019 from https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt
[4] [n.d.]. *Linux control groups.* Retrieved November 21, 2019 from http://man7.org/linux/man-pages/man7/cgroups.7.html
[5] [n.d.]. *Linux CPUFreq Governors.* Retrieved November 21, 2019 from https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
[6] [n.d.]. *Linux Programmer's Manual PROC.* Retrieved November 21, 2019 from http://man7.org/linux/man-pages/man5/proc.5.html
[7] [n.d.]. *Overview of CPU scheduling.* Retrieved November 21, 2019 from http://man7.org/linux/man-pages/man7/sched.7.html
[8] [n.d.]. *Performance Counters for Linux (PCL) Tools and perf.* Retrieved November 21, 2019 from https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/developer_guide/perf
[9] [n.d.]. *Scheduler Extensions.* Retrieved November 21, 2019 from https://android.googlesource.com/kernel/msm/+/android-msm-marlin-3.18-nougat-dr1/Documentation/scheduler/sched-zone.txt
[10] [n.d.]. *Simpleperf.* Retrieved November 21, 2019 from https://developer.android.com/ndk/guides/simpleperf
[11] [n.d.]. *Test UI performance.* Retrieved November 21, 2019 from https://developer.android.com/training/testing/performance
[12] Ionut Arghire. 2019. *Linux Crypto-Miner Uses Kernel-Mode Rootkits for Evasion.* https://www.securityweek.com/linux-crypto-miner-uses-kernel-mode-rootkits-evasion
[13] Yogesh Babar. 2012. *Understanding Linux Process States.* https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf
[14] Jonathan Corbet. 2013. *Per-entity load tracking.* https://lwn.net/Articles/531853/
[15] Xing Gao et al. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19).* Association for Computing Machinery, New York, NY, USA, 1073–1086. https://doi.org/10.1145/3319535.3354227
[16] Dirk Grunwald et al. 2002. Microarchitectural Denial of Service: Insuring Microarchitectural Fairness. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35).* IEEE Computer Society Press, Los Alamitos, CA, USA, 409–418.
[17] Jahangir Hasan et al. 2005. Heat Stroke: Power-Density-Based Denial of Service in SMT. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05).* IEEE Computer Society, Washington, DC, USA, 166–177. https://doi.org/10.1109/HPCA.2005.16
[18] Tejaswini Kolpe et al. 2011. Enabling improved power management in multicore processors through clustered DVFS. In *2011 Design, Automation Test in Europe.* 1–6. https://doi.org/10.1109/DATE.2011.5763052
[19] Deborah T. Marr et al. 2002. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal* 6, 1 (2002), 1.
[20] Vikram Mulukutla. 2016. *sched: Introduce Window Assisted Load Tracking.* https://lwn.net/Articles/704903/
[21] Venkatesh Pallipadi and Alexey Starikovskiy. 2006. The ondemand governor. In *Proceedings of the Linux Symposium,* Vol. 2. 215–230.
[22] Steven Rostedt. 2008. *ftrace - Function Tracer.* https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[23] Guillermo Suarez-Tangil et al. 2014. Evolution, Detection and Analysis of Malware for Smart Devices. *IEEE Communications Surveys Tutorials* 16, 2 (Second 2014), 961–987. https://doi.org/10.1109/SURV.2013.101613.00077
[24] Adrian Tang et al. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17).* USENIX Association, Vancouver, BC, 1057–1074.
[25] Mark Weiser et al. 1994. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI '94).* USENIX Association, Berkeley, CA, USA, Article 2. http://dl.acm.org/citation.cfm?id=1267638.1267640