DFS Covert Channels on Multi-Core Platforms

Murugappan Alagappan*, Jeyavijayan (JV) Rajendran†, Miloš Doroslovački* and Guru Venkataramani*
*Department of Electrical and Computer Engineering, The George Washington University
†Department of Electrical Engineering, The University of Texas at Dallas
Email: {muru_05, doroslov, guruv}@gwu.edu, jv.ee@utdallas.edu

***Abstract*—Covert channels provide a secret communication medium between two malicious processes to exfiltrate information stealthily that violates the security policy of a system. In this paper, we demonstrate a new covert timing channel attack that exploits the CPU operating frequencies with different power governors in real system environment. In particular, we establish how two colluding processes–a trojan and a spy can modulate the CPU frequency to create a powerful, high-capacity and robust covert channel. We implement this covert channel both in a single threaded and simultaneous multi-threading (SMT) environment and show the feasibility of such a communication. Our experiments on Intel Xeon server platform demonstrate dynamic frequency scaling covert channels that can achieve up to 20 bits/second.**

Keywords–covert channel; dynamic frequency scaling; information leakage; system security

I. Introduction

With the advent of cloud computing infrastructure, information leakage is a fast growing concern where multiple tenants share resources and the underlying hardware platforms. Among the many forms of information leakage, covert channels operate through two colluding applications that secretly communicate with each other through a shared resource despite the underlying system security policy prohibiting any such communication between these two applications [1]. Covert channels can be implemented as storage channels that exploit shared memory covertly [2], or as timing channels that simply modulate the resource access timing without leaving any physical evidence to communicate secrets [3]. In contrast to side channels where a victim process unintentionally leaks information to a spy process that monitors system activity, covert channels operate with an insider process that intentionally colludes with a spy to exfiltrate information (see figure 1).

Prior works have studied hardware covert timing channels implemented using branch predictors [4], caches [5], [6], [7], processor-memory bus [8] and random number generator (RNG) modules [9]. In all of these types of attacks, the trojan modulates the access timing of a certain specific hardware resource by intentionally creating contention events allowing the spy to observe the altered access timing on that resource and infer the bit(s) transmitted.

In this paper, we demonstrate a new covert timing channel that exploits dynamic frequency scaling (DFS) feature supported by the CPU where its operating frequency can be altered based on the workload. We show that covert channels can be implemented on real systems through manipulating the power governors that control the CPU frequency settings. We study different implementations of DFS-based covert channels by varying the spy's mode of operation in decoding the CPU frequency. Our experimental results show a high bandwidth timing channel of up to 20 bits/second is feasible.

978-1-5386-2880-5/17/$31.00 © 2017 IEEE

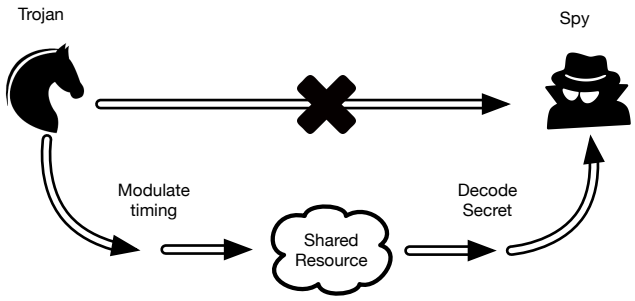

Fig. 1. A covert timing channel using timing modulation on a shared resource to divulge secrets

Power-efficient computing has become a requirement for most modern systems especially as processors are becoming exceedingly power-hungry with their multitude of micro-architectural units. A well-known strategy to manage CPU power is the dynamic voltage and frequency scaling (DVFS), that has been widely supported by a range of systems from personal mobile devices to large scale computers [10]. DVFS provides the capability to scale the voltage and frequency settings of the processors on-the-fly depending on the workload and utilization levels of the CPU cores. Most modern CPUs are equipped with DVFS capability [11]. With such widespread adoption and use, the vulnerabilities exposed by DVFS feature may be exploited by adversaries on multiple types of systems resulting in potentially huge economic losses or even greater damages. We note that our work offers much needed insights into how an adversary could exploit DVFS, and we explore methods to mitigate such attacks.

The contributions of our work are:

- We demonstrate the feasibility of exploiting DFS supported by CPUs for covert timing-channels using Intel Xeon server X5560 platform.
- We study DFS covert timing channel implementation under different CPU power governors and CPU settings such as simultaneous multi-threading (SMT).
- We present experimental results that demonstrate a maximum bandwidth of 20 bits/s using DFS covert timing channels, and we show indirect frequency inference methods that the spy may utilize to infer CPU frequency and evade detection.

II. Background

Most modern multi-core processors export the CPU core frequency information to the user processes. This may be used by the applications to tune the processor's frequency settings such that a good trade-off between application performance and CPU frequency can be achieved. While providing frequency/voltage information might be valuable to design power-aware applications, this may pose a significant threat when adversaries can exploit this for malicious purposes [12].

Different applications may have varied security permission settings and access to a range of critical information. A malicious application that may exfiltrate sensitive secrets to a spy process bypassing system security policy can be dangerous to system integrity. To illustrate this problem with an example scenario, lets consider that they are two applications running concurrently on a system where one is a personal finance management application and the second is a global positioning system navigation application. The personal finance management application has access to sensitive user data such as passwords and details about the customer's financial institutions. In order to operate, this finance management application will also need to access network resources to communicate with customer's bank servers. However, the operating system will scrutinize the packets sent by this application and likely add protection features such as encryption to prevent any information outflows. Therefore, this application cannot directly expose sensitive user data. However, if the financial management application is malicious, it may collude with another *seemingly benign* application that has access to the network but is not likely to be scrutinized by the OS or other security enforcer modules. We note that a largely benign navigation application may be exploited for this purpose since it needs to have network access enabled to support its operation. Through covert timing channel based transmission, the financial application can effectively send sensitive secrets to the outside world without ever being detected. Such situations highlight the need for a more thorough understanding of how covert timing channels operate and methods to mitigate the damages caused by them.

A. CPU Power Governors

A CPU power governor defines the power schemes (policies) for the system CPU [13]. Linux kernel provides instructions to the processor based on the workload needs, which in turn scales the CPU frequency on-the-fly during run-time [14]. CPU frequency scaling serves to reduce the overall CPU power consumption and reduce potential thermal hot spots [15]. If frequency settings are changed without being aware of the workload, the system's performance may be adversely affected.

There are different types of CPU governors in the Linux kernel namely: performance, powersave, ondemand, userspace and conservative. Each governor has two preset parameters namely `scaling_min_frequency` with the lowest available CPU frequency and `scaling_max_frequency` with the highest available CPU frequency. Note that, at any given point of time only one CPU governor may be active per core. We describe the CPU governors briefly below:

Performance: The performance CPU governor always sets the processor frequency to the maximum available frequency in order to increase the performance of the system.

Powersave: The powersave governor always sets the processor frequency to the minimum frequency available in order to reduce the system power consumption.

Ondemand: The ondemand governor is the most common CPU governor used in the Linux kernel [16]. The ondemand governor typically sets the processor frequency based on the CPU usage and the workload assigned to it. The ondemand governor uses different `sysfs` parameters such as `up_threshold`, `sampling_rate_min`, `sampling_down_factor`, `sampling_rate` and `ignore_nice_load`.

Upon setting the ondemand governor, the CPU load is frequently checked. When the load raises above the `up_threshold` preset value, the ondemand governor automatically sets the CPU to run at the highest possible frequency, which is denoted by `scaling_max_frequency`. When the load falls below the preset threshold, the ondemand governor sets the CPU to run at the lowest possible frequency, which is denoted by `scaling_min_frequency`. The `sampling_rate`, which is typically denoted in microseconds, defines how often the kernel needs to monitor the CPU usage to make decisions on whether to change the CPU frequency either to the highest or lowest. The `sampling_rate_min` parameter defines the minimum time in microseconds on how frequently the kernel needs to monitor the CPU usage and make decisions about changing the frequency.

Userspace: The userspace governor sets the processor frequency to a value specified by the user or any userspace program. The userspace governor uses the `scaling_setspeed` governor parameter to change different frequencies within the range of `scaling_min_frequency` and the `scaling_max_frequency`. The userspace governor is typically used along with the `cpuspeed` daemon and is most customizable among all the CPU governors.

Conservative: The conservative governor operates very similar to ondemand governor. However, a major difference between them is that the ondemand governor switches between the `scaling_max_frequency` and `scaling_min_frequency` aggressively whereas the conservative governor does so more gradually.

By default, the CPU governor is set to *ondemand* that automatically scales the CPU frequency based on the CPU workload's throughput. The users have the ability to either change the CPU governor (done via `sysfs` [13]) and to request the CPU to operate at a certain frequency (done via `sysfs` [14]). Note that such frequency changes are done in root permission mode.

III. Threat Model and Assumptions

In our threat model, we assume that there are two covertly communicating processes (namely the trojan and the spy) that subversively communicate despite the underlying system

security policy not permitting the two processes to communicate (via Inter-process communication, networks, file system, or shared memory). The trojan is a process with higher privileges and access to sensitive data, that is normally not allowed to reveal data to the external processes. The spy is a *seemingly benign* process with lesser privileges than the trojan, that is co-located alongside the trojan process sharing micro-architectural (functional) units with it. We assume that the trojan and the spy can run on any core.

The operating system is assumed to be un-compromised so that it diligently preserves legitimate information flows and enforces the right access control. The Linux kernel CPUFreq subsystem is used for controlling DVFS with a range of power governors to choose from (ondemand, userspace, powersave, performance and conservative). The two processes do not require any special privileges to establish a covert channel for power governors like ondemand and conservative. But if the power governor is chosen to be Userspace, the trojan process needs to have super-user (*sudo*) rights to set different frequencies.

IV. DFS Covert Channel

In this section, we show how a trojan and spy process could manipulate the DFS feature supported by the CPUs to implement their covert timing channels. We first show how to construct the covert channel through exploiting the ondemand and userspace CPU governors, and then demonstrate how the spy may potentially evade detection through adopting indirect methods to infer CPU core frequency.

All our experiments were performed on Intel Xeon X5560 CPU with 8 cores and 12 GB of DDR3 memory. Each core has a set of 10 distinct frequencies ranging from 1.59 GHz to 2.79 GHz. The machine runs Ubuntu 16.04.1 LTS operating system, with a generic GNU/Linux kernel version 4.4.0-42. Our experiments assume minimal interference from the external processes in terms of competing workloads that can alter CPU frequency. If such interference is seen, we note that de-noising can be relatively straightforward if the trojan/spy monitor for workloads in the cores that they are about to run, and move to a different core whenever they detect high interference.

A. Building the Dynamic Frequency Covert Channel

In order to demonstrate the feasibility of our covert channels, we execute two malicious process—trojan and spy with two possible scenarios: 1. ondemand CPU governor, 2. userspace CPU governor. The performance and powersave governors cannot be used as it allows the CPU to run only at a single preset frequency.

The goal of the trojan here is to *covertly send* a secret bit by modulating the frequency of its CPU core. The spy then infers the bit as a 0 or a 1 based on the frequency set by the trojan process.

Exploiting ondemand governor: To transmit a bit 1, the trojan increases the frequency of the CPU core to `scaling_max_frequency` by running an intensive computation e.g. perform a dense matrix multiplication. To transmit a 0, the trojan just sleeps for the entire interval bringing down the frequency to `scaling_min_frequency`. The spy periodically samples with non-perturbing reads that reads the frequency of the core through `/sys/devices/system/cpu/cpu*/scaling_cur_freq`. When the frequency is high it infers a 1 and 0 otherwise.

```
char bit;
FILE *file = fopen("bits_to_send.txt", "r");
while ((bit = getc(file)) != EOF)
  if(bit == 1)
    do_intense_computation_for(interval);
  else
    usleep(interval);
fclose(file);
```

Listing 1: Trojan using ondemand governor

To illustrate, let us consider a trojan that wants to send the bits 1010. The trojan would do an intensive computation for the first second, sleep for the next second, repeat the computation for the next and so on. The spy would read the frequency of the CPU core every second and infer the bits sent by the trojan. To detect and correct any bit errors, we note that the transmission could include error correction bits that helps improve reliable transmission between the trojan and the spy. It is worth noting here that the spy can run on any core since the kernel allows any userspace program to read the frequency of other CPU cores in the system without any privilege requirement. We note that, despite the fact that multiple frequency settings are supported by the CPU, it is usually easier to infer the binary-coded data using ondemand governors. The trojan implementing DFS covert channel using ondemand CPU governor is shown in listing 1.

```
FILE *fp=fopen("bits_to_send.txt","r");
char bit;
// list of available frequencies
char *list_of_frequencies[]{
  "1.59Ghz","1.72Ghz","1.82Ghz","1.99Ghz",
  "2.12Ghz","2.26Ghz","2.39Ghz","2.79Ghz"};
while((bit = fgetc(fp))!=EOF){
 set_freq(list_of_frequencies[int(bit)]);
 usleep(interval); }
```

Listing 2: Trojan with userspace governor

Exploiting userspace governor: In contrast to ondemand governor, *userspace* governor allows the trojan process to set the frequency of any CPU. Our system allows the CPU to run using 10 different frequencies. This makes it possible to send decimal-coded messages, one corresponding to each frequency i.e. setting a frequency of 1.59 GHz corresponds to a 0 and 1.72 GHz corresponds to a 1, and so on. The trojan implementing DFS covert channel using userspace governor is shown in the listing 2. The spy program is the same as used in the ondemand governor where it periodically reads the core's frequency through `sysfs` or `/proc/cpuinfo`.

For a successful transmission of a message, the spy process needs to decode the message from the trojan at about the same

rate that it is being sent. In order to efficiently transmit a message, the trojan and the spy should synchronize on two pieces of information: 1. number of distinct frequencies that used by the trojan for modulation, and 2. bit transmission period.

There are several ways a spy process can determine the current frequency setting of a particular CPU. A straightforward method is to read from the `/proc/cpuinfo` file. This method has a high performance overhead and it not suitable for sending messages at a higher rate. The maximum achievable bandwidth using this method is 3 bits/sec with no error. Alternatively, the spy process can read the current frequency of the core directly from `/sys/devices/system/cpu/cpu*/scaling_cur_freq` which also has a maximum achievable bandwidth of 3 bits/sec with no error.

B. Frequency Inference

An indirect way that the spy can use to evade detection and infer the CPU frequency is to run a non-perturbing, lightweight loop for a fixed amount of observation time. Based on the value of loop iteration counter at the end of the preset observation period, the CPU core frequency is inferred.

Note that the spy must know how to infer the communicated bit based on the values of the counters observed, and must calibrate itself to distinguish the band of counter values and the corresponding bits. This can be done so by studying the distribution of counter values that the spy reads over time.

```
unsigned long counter;
while(1) {
  counter = 0;
  increment_counter(interval); }
```

Listing 3: Spy program for inference based method

As an instance, let us consider a scenario where the trojan communicates a specific bit. On the spy side, rather than directly reading a file containing the current CPU frequency, it runs a non-perturbing *timed* loop that runs for a preset amount of time showed in listing 3. At the end of each observation period, a certain counter value is observed. We repeat this experiment for a 10 times, and plot the distribution of values observed during such runs. Figure 2 shows our results, where we see a distribution centered around 180×10^5 when trojan communicates a 0 bit, and another distinct distribution centered around 320×10^5 when the spy communicates a 1 bit. Using this inference method, we clearly see that there exist two separate bands of counter values that helps the spy to distinguish a 1 from a 0 bit transmission. This experiment shows the feasibility of inferring CPU core frequency even if the spy doesn't directly read the system files (note that reads on such files may be tracked to monitor for illegal activity by a spy process).

V. Results and Discussion

In this section we demonstrate the feasibility and the effectiveness of our covert channel using ondemand governor, userspace governor, and inference based methods.

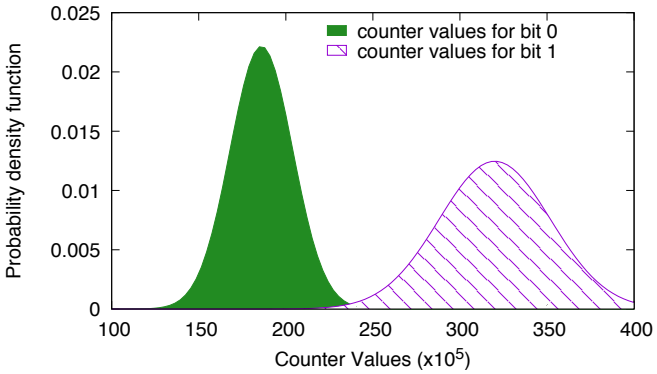

Fig. 2. Gaussian distribution of counter values (10^5)

Figure 3 shows a random 64-bit pattern transmission using ondemand power governor. We show the bit sequence the trojan process is trying to communicate covertly to the spy process. A higher bar indicates a 1 and a lower bar indicates a 0. Below the trojan process are the two methods the spy process uses to infer the frequency. We first show that the spy process is perfectly able to decode the bit pattern by reading the core's frequency set by the trojan from `/proc/cpuinfo`. We also show that spy is able to successfully decode the bit pattern by decoding the frequency from `sysfs` filesystem.

In the the case of ondemand power governor, there are only two frequencies that represent a 0 or 1 corresponding to `scaling_min_frequency` and `scaling_max_frequency` which is 1.59 GHz and 2.79 GHz in our system.

In the case of userspace governor, in our system, we were able to use base 10 to transmit bits from trojan to spy process as we have 10 different frequencies that we can set. For simplicity in figure 4 shows trojan process using base 4, i.e. use 4 different frequencies to communicate to the spy. One can see that spy can decode the frequency and get the corresponding number without any errors.

In this case, we could encode 10 integers (3.25 bits) for every interval. We found that the trojan process could transmit data with intervals up to 10 ms without any errors. This gives us a bit rate of 325 bits/second. Since, this method requires trojan to have super-user (*sudo*) privileges, which is unlikely but possible in the real world scenario, we do not advertise this as our main result.

For demonstration purposes, we showed how the trojan transmits information to the spy using integers, as it can be related to leaking a credit card number, but this can easily be extended to sending any information with high speed and accuracy.

Figure 5 shows the trojan sending a 0 or 1 by either performing a intense computation or sleeping for a given interval. The y-axis for the spy shows the value of the counter for spy. Recollect from listing 3 that in this method the spy program just keeps running a counter for a specific time interval, and decodes the bits based on the counter's value.

One can see that when the frequency is low, the value of the counter is also low. This is because on userspace governor, the core is set to its minimum which means less computation will be done when compared to a core that runs on its maximal

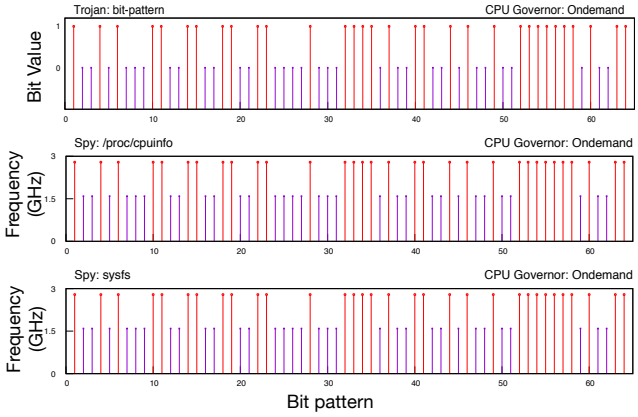

Fig. 3. Bit transmission using ondemand governor

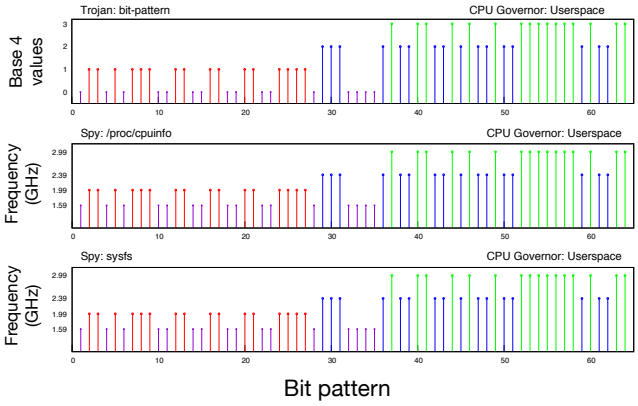

Fig. 4. Bit transmission using userspace governor

frequency. In the case of ondemand governor, the value of the counter will be low because of the interference from the trojan, and the counter will have a high value when there is no interference i.e. trojan is executing its *sleep* command.

Figure 6 illustrates the error rate for each of our methods using ondemand governor. We increased the bit rate by lowering the interval with which the trojan and spy communicate. i.e. for an interval of 100 ms the bit rate is 10 bps. Both `/proc/cpuinfo` and `sysfs` methods exhibit same error rate. The error increases rapidly as we start to send more that 3 bps. We believe this is because of the rate at which the kernel updates the *proc* and the *sysfs* file system. Additionally, we incur the overhead of opening and reading the values from the file.

In contrast, our inference based method is a bit more resilient to the speed up until 20 bps and the error starts increasing thereafter. This is because the inference based method doesn't depend on external factors like the kernel/user writing/reading from any files.

VI. Other Possible Design Options

There are a number of challenges that remain in proving this covert channel to be an effective and an imminent threat. We have not considered background noise in this work, but

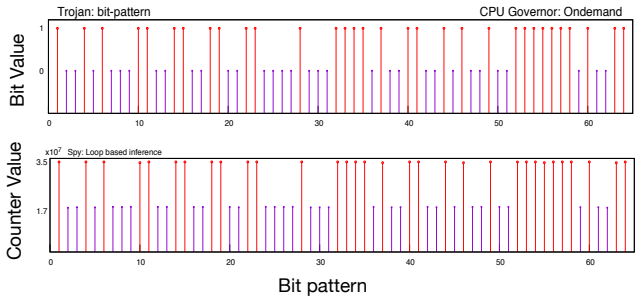

Fig. 5. Bit transmission using inference based method

there are various digital signal processing methods [17] that can be used to eliminate the noise from the actual signal.

Next, the bit transmission rate can be increased in several ways depending on whether or not the spy has root privileges. In the case where the spy has root privileges on the system, the spy could have a kernel module running that constantly monitors the CPU frequency from the registers.

Finally, reading the current frequency does not tell us anything about the average frequency over an interval. Intel has a pair of model specific registers (MSR) defined that are ideal for monitoring average frequency: `IA32_MPERF` (MSR `0xE7`) and `IA32_APERF` (MSR `0xE8`). To determine the average core frequency over an interval, we could use the fixed-function counters "Reference Cycles Unhalted" and "Core Cycles Unhalted".

On the other hand if the spy process is limited to be in userspace mode (no root privileges) we propose couple of options. First, for inline instrumentation, these performance counters (or their programmable equivalents) can be read using the `RDPMC` instruction directly from user mode. Second, for external instrumentation, that might have high performance overhead is performing an inter-processor interrupt so that the target processor will "wake up" and read its own MSRs.

VII. Related Work

There have been numerous works on covert channel research over past decade. Researchers have come out with smart ways of ex-filtrating information stealthily. Recently, Masti et al. [18] demonstrate a covert channel implementation that takes advantage of the thermal sensors information exposed by processors chips. Yao et al. [19] demonstrate a hardware-based covert timing channel implementation that exploits the timing differences in cache read accesses exposed by NUMA-based architectures.

Wu et al. [20] demonstrate a covert communication channel that is based on Intel Quick Path Interconnect (QPI) lock mechanism. Ristenpart et al. [21] present a cross-VM covert channel by exploiting the L2 cache. Evtyushkin et al. [4] present a covert channel though branch predictors. There have also been studies [8] which uses memory bus as covert channels. Recently, there have been some interesting study [9] on using the micro-architectural features like random number generator as covert communication channel.

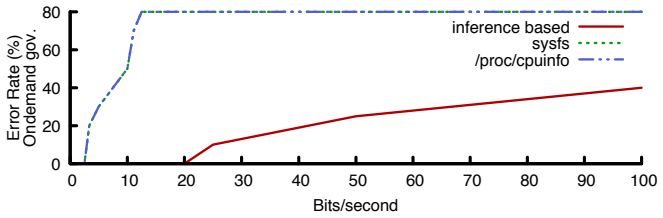

Fig. 6. Error rate for different techniques used by spy process to decode the CPU frequency (sysfs and /proc/cpuinfo overlap for all bit rates)

Side channels have been well studied in the recent times. Typically, side channels are used for crypt-analysis. There have been several attacks which includes the extraction of AES keys [22], DSA keys [23] and RSA keys [24].

On of the possible ways to detect timing channels (irrespective of either side or covert) is to expose less timing information in general [25]. Venkataramani et al. [26] study how to detect contention-based covert timing channels through dynamically tracking conflict patterns on shared hardware resources. One of the popular ways to mitigate covert and side channels is through partition of system resources [27]. There have also been works [28] to detect malware through analyzing existing performance counters proposed.

VIII. Conclusion

In this work, we have presented a new type of covert channel exploiting the popular power management strategies adopted in modern day processors and using them as covert channels to exfiltrate data stealthily between the trojan and the spy processes. The key idea is that the trojan process can modulate the discrete CPU operating frequencies by setting them either directly or indirectly with the help of power governors. In order to read the CPU frequency, the spy process can either read directly from the sysfs, procfs file systems provided by the Linux kernel or indirectly by inference based method. We have demonstrated the feasibility and the potential of achieving a high bandwidth of up to 20 bits/second using dynamic frequency covert channels on Intel Xeon server X5560.

IX. Acknowledgment

This material is based on work supported by the US National Science Foundation under CAREER Award CCF-1149557 and CNS-1618786, and Semiconductor Research Corp. (SRC) contract 2016-TS-2684. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of the NSF or SRC.

References

[1] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014.

[2] JingZheng Wu, Liping Ding, Yongji Wang, and Wei Han. Identification and evaluation of sharing memory covert timing channel in xen virtual machines. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011.

[3] John C Wray. An analysis of covert timing channels. *Journal of Computer Security*, 1(3-4):219–232, 1992.

[4] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: A feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '15. ACM, 2015.

[5] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015.

[6] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[7] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[8] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. Busmonitor: A hypervisor-based solution for memory bus covert channels. *Proceedings of EuroSec*, 2013.

[9] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[10] Intel Turbo Boost Technology 2.0, [Online]. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.

[11] Intel Xeon Server Processors Support. http://www.intel.com/support/processors/xeon/sb/cs012641.htm.

[12] Mengchao Yue, William H Robinson, Lanier Watkins, and Cherita Corbett. Constructing timing-based covert channels in mobile networks by adjusting cpu frequency. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 2. ACM, 2014.

[13] Kernel documentation. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[14] Kernel documentation. https://www.kernel.org/doc/Documentation/cpu-freq/cpufreq-stats.txt.

[15] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th international conference on Supercomputing*, pages 35–44. ACM, 2002.

[16] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*. sn, 2006.

[17] Sebastian Zander. *Performance of selected noisy covert channels and their countermeasures in IP networks*. PhD thesis, Swinburne University of Technology Melbourne, 2010.

[18] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal covert channels on multi-core platforms. In *USENIX Security*, 2015.

[19] Fan Yao, Guru Venkataramani, and Milos Doroslovacki. Covert Timing Channels Exploiting Non-Uniform Memory Access based Architectures. In *Proceedings of the 27th edition on Great Lakes Symposium on VLSI (GLVLSI)*. ACM, 2017.

[20] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, 2012.

[21] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009.

[22] Onur Acıiçmez, Werner Schindler, and Çetin K Koç. Cache based remote timing attack on the aes. In *Cryptographers Track at the RSA Conference*. Springer, 2007.

[23] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002.

[24] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[25] Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of computer security*, 1992.

[26] G. Venkataramani, J. Chen, and M. Doroslovacki. Detecting hardware covert timing channels. *IEEE Micro*, 36(5):17–27, Sept 2016.

[27] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006.

[28] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.