Use of Per-CPU Counters to Protect Kernel-Mode Processes on a Multicore Processor

Joel Agnel Fernandes

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.
Use of Per-CPU Counters to Protect Kernel-Mode Processes on a Multicore Processor

ABSTRACT

When user and kernel mode processes are scheduled to run simultaneously on the same physical processor core via simultaneous multithreading, processor vulnerabilities can result in sensitive data restricted to kernel-mode processes being leaked to user-mode processes. While disabling simultaneous multithreading or use of a scheduler that determines sequence of execution can prevent such leakage, these changes degrade processor performance. This disclosure describes the use of per-CPU counters to track when each physical core of a processor begins and ends the execution of a program thread running in the kernel mode. Upon detection of such a thread, the core is identified as being in an unsafe state and all user-mode tasks are halted using an inter-processor interrupt.

KEYWORDS

- Simultaneous multithreading (SMT)
- Hyperthreading
- Multicore processor
- Virtual core
- Operating system scheduler
- Kernel mode
- User mode
- Inter-processor Interrupt (IPI)
- RIDL
- ZombieLoad

BACKGROUND

Many computing devices use processors that contain multiple physical cores (multicore processors) that enable parallel execution of program threads, referred to as simultaneous multithreading (SMT). The multiple threads typically operate in parallel on separate data. Moreover, each physical core can be addressed as one or more virtual cores, thus enabling additional parallelization by increasing the number of independent instructions in the processing
pipeline. Each virtual core appears to the operating system as a separate processor, making it feasible to schedule multiple processes for each physical processor core. Any of the processes assigned to the virtual cores can continue to execute as long as the needed resources are available.

Processes that execute on a processor can run in kernel or user mode depending on whether they originated from the operating system kernel or a higher-level user application, respectively. In execution, kernel-mode programs are treated as trusted and permitted access to resources without restrictions. In contrast, user-mode processes are untrusted and are subject to various restrictions that prevent them from accessing sensitive resources. However, when user and kernel mode processes are scheduled to run at the same time on the multicore processor, vulnerabilities in processor architecture can result in data restricted to kernel-mode processes leaking to user-mode programs (e.g., as described in [1] [2]).

One way to avoid such security vulnerabilities is to disable SMT functionality. Alternatively, or in addition, user-mode processes can be protected from potential attacks by making changes to the scheduler that determines the sequence in which program instructions are scheduled to be executed across the processor cores. Such changes can introduce bottlenecks and lead to performance degradation.

DESCRIPTION

This disclosure describes the use of per-CPU counters to track when each of the physical cores of a processor begins and ends the execution of a program thread running in the kernel mode. The value of the counter for each physical core is incremented whenever a kernel-mode thread begins execution on the core and is decremented whenever the thread stops. The counter thus helps detect whether a given physical core as a whole is performing any kernel-mode task.
Whenever a given physical core is detected to be executing one or more kernel-mode threads, it is deemed to be in an unsafe state that is vulnerable to attacks, e.g., originating from untrusted user-mode programs.

Per techniques of this disclosure, Whenever a counter indicates that a physical core switches from a safe to an unsafe state, the thread that causes the shift in the state can generate an inter-processor interrupt (IPI) to any ongoing user-mode processes executing on the core in parallel. Any user-mode thread that receives an IPI immediately suspends execution and waits in a busy state until the counter indicates that the core is no longer deemed to be in the unsafe state.

**Fig. 1: Inter-processor interrupt to hold execution of user-mode processes in unsafe state**

Fig. 1 (a) shows a visual depiction of the techniques described in this disclosure. Each of the processes illustrated in Fig. 1(a) is a hardware thread associated with a particular core. A user-mode process (102) is executing on a processor core until time $t_1$. As shown in Fig. 1 (b), the counter value at this time is 0 since no kernel-mode processes are running. At time $t_1$, a
kernel-mode process (104) is scheduled to execute. To ensure security, an inter-processor interrupt (IPI) (106) is generated to indicate that the core is entering an unsafe state, and the counter is incremented to 1.

As Fig. 1 (b) shows, the counter is further incremented when another kernel-model process begins executing in parallel at time $t_2$ and is decremented when the first kernel-mode process stops running at time $t_3$. The entire core is deemed to be in the unsafe state (108) until time $t_4$ when the second kernel-mode thread stops executing after serving a soft interrupt (110). As seen in Fig. 1(b), the counter returns to 0 at time $t_4$ indicating that the core is no longer in the unsafe state. At this time, the user-mode process exits waiting in the busy state (112) induced by the IPI and continues execution from the point at which it was halted.

IPIs are sent only when a kernel-mode process results in switching the state of a core from safe to unsafe. Such a switch results only when the core is idle or executing a user-mode process at the time when a kernel-model process begins executing. On the other hand, if the core is executing other kernel-mode processes, the core is already deemed to be in the unsafe state when a new kernel-mode thread begins executing. As a result, there is no change in the state of the core and no IPI is generated when the new kernel-mode thread begins executing. Such operation minimizes the number of IPIs that are generated, thus reducing overhead.

Sometimes an untrusted user-level process currently executing on a physical core itself needs to perform kernel-mode tasks which leads to the core entering an unsafe state. In such cases, if the other core virtual cores associated with the physical core are idle (not executing any other threads), no IPI is generated. If any other kernel-mode processes begin execution after the user-level process begins its kernel-mode tasks, the user-level process remains waiting in busy mode after completing its execution of kernel-mode tasks. In such cases, entering the busy mode
does not require any IPIs and the waiting continues until all other kernel-mode processes complete execution and the physical core returns to the safe mode.

Notably, as seen in Fig. 1(a), the described techniques ensure protection for soft interrupts since these are nested within kernel-model processes, such as interrupt requests (IRQs) and system calls. Moreover, Fig. 2 shows that the idle loop (114) to enter and exit the unsafe core state is modified to ensure that the unsafe state is exited as soon as possible such that any untrusted user-mode processes (102) that are on hold (112) are not put on hold for an unreasonably long time.

**Fig. 2: Modified idle loop to avoid unreasonable wait times for user-mode processes**

Implementation of the described techniques eliminates the need to wait in a scheduling loop when switching from a trusted task to an untrusted task within the same process. Moreover, if a kernel-mode process begins executing when the other running threads are in trusted user-mode contexts (e.g., system daemon) or in the idle state, no IPI is generated since no untrusted-mode thread needs to be halted.
Techniques described in this disclosure can be implemented in an operating system for any device that incorporates a central processing unit (CPU) with multiple cores capable of simultaneous multithreading. Implementation of the techniques can ensure that user mode processes (which may be malicious) cannot access sensitive data when the processor employs simultaneous multithreading, and that such data does not leak accidentally. The techniques achieve such protection without requiring the simultaneous multithreading feature of the processor to be turned off. The operation is optimized to only generate inter-processor interrupts when necessary which improves efficiency and reduces the need to put threads in a waiting state. As a result, the techniques can provide enhanced security with a relatively low impact on performance.

CONCLUSION

When user and kernel mode processes are scheduled to run simultaneously on the same physical processor core via simultaneous multithreading, processor vulnerabilities can result in sensitive data restricted to kernel-mode processes being leaked to user-mode processes. While disabling simultaneous multithreading or use of a scheduler that determines sequence of execution can prevent such leakage, these changes degrade processor performance. This disclosure describes the use of per-CPU counters to track when each physical core of a processor begins and ends the execution of a program thread running in the kernel mode. Upon detection of such a thread, the core is identified as being in an unsafe state and all user-mode tasks are halted using an inter-processor interrupt. The operation is optimized to only generate inter-processor interrupts when necessary which improves efficiency and reduces the need to put threads in a waiting state. As a result, the techniques can provide enhanced security with a relatively low impact on performance.
REFERENCES

1. Van Schaik, Stephan, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi
   Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue in-flight

2. Schwarz, Michael, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas
   Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling." In
   *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications

3. Core scheduling, available online at https://lwn.net/Articles/780703/ accessed on 24 Nov
   2020.