The Software-Defined CPU

Michael Roitzsch
Barkhausen Institut
Dresden, Germany

Till Miemietz
Barkhausen Institut
Dresden, Germany

ABSTRACT

Our CPUs contain a compute instruction set, which regular applications use. But they also carry a complex underworld of different CPU modes, combined with intricate trap and exception handling to transition between these modes. These mechanisms are manifold, complex, and are largely outside of meaningful operating system control. We have to take what CPU vendors provide, including potential security problems from unneeded modes. This paper explores the question, whether CPU modes could instead be defined entirely by software. It shows how such a design would function and explores the advantages it enables.

1 INTRODUCTION

In bygone years, operating systems interacted with the CPU in the simple terms of traditional user and kernel mode. Privileged features, like page table manipulation and interrupt handling, were restricted to kernel mode, while user mode handled regular application code. But as the systems community demanded more features to play with, CPU vendors delivered: Hypervisor modes with nested paging enable hardware-supported virtualization, monitor modes enable strongly isolated security contexts [3]. Since CPU performance growth has slowed, CPU vendors started looking at feature diversification, perpetuating the trend of adding more CPU modes: SGX [2], MPK [6], and SEV [1] are the latest additions to the family. This plethora of new modes would not be a problem if they did not also come with a lot of complexity added to our CPUs. Recent years have shown how brittle CPU implementations already are [7, 8], and the new modes certainly do not help [9, 10]. Whether we use these modes or not, all the associated complexity is always present, completely outside operating system control. SGX, for example, is highly complex and assumed to be largely implemented in microcode [4]. But although it is microcode, it is inseparably linked to the silicon. Although firmware merely is software sold by a hardware vendor, the system software cannot influence this microcode to disable unneeded features or to add new ones.

This paper poses the question: What if we could? What if we had the pendulum swing all the way to the other end? Let us assume we could not just change microcode, but instead had CPUs, where the very nature of CPU modes was fully programmable. The goal of such a CPU design would be to throw away *all* existing CPU modes and replace them with software. Let us explore this wild and crazy idea in the remainder of this paper.

2 CPU PROGRAMMABILITY

Intuitively, CPUs should take the top spot on the list of programmable devices. All software is essentially programming the CPU, right? Our programs are translated to an instruction stream, which the CPU consumes and interprets. The instructions we use invoke the CPU-internal function blocks like arithmetic logic units (ALUs), floating-point units (FPUs), load-store units, and branching units. We interpret the resulting state changes as program execution. All user code works this way, but also operating system and hypervisor code consists mostly of these instructions. We call this portion of CPU operation the *CPU data plane,* because its main observable effect is the transformation of input data to output data.

But next to this data plane, there is a whole other world, which does not process data, but influences *how* the CPU processes data. This *CPU control plane* is invoked by way of traps and exceptions and contains the logic of CPU mode switches. This logic is hardwired and complex, which is in stark contrast to the orthogonal and composable function blocks of the data plane. This part of the CPU is exactly what we want to replace in order to realize our vision of a software-defined CPU.

But how should we construct a programmable CPU control plane? Similar to the data plane, we want distinct, composable function blocks which system software can freely orchestrate. Of course, this will be a different kind of software. Its instruction stream does not process data, it rather implements the CPU modes and transitions between them. It does so by programming the control plane function blocks to configure the environment in which the data plane instructions will run. This concept raises two questions: What should these function blocks be? And more importantly, where — that is, in what CPU mode — should the control plane instructions run?

3 THE MODE SWITCH MODE

Here, this concept becomes self-contradicting at first glance, because we are essentially proposing to add yet another CPU mode. We call it the *mode switch mode* (MSM) and it should run the control plane code, which programs the control plane function blocks. But obviously, the goal of this mode is to subsume all other built-in CPU modes by allowing the control plane code to implement all other modes dynamically in software. In that sense, the MSM is indeed a new mode, but it is the last and only CPU mode we will need to worry about ever again.

Necessary Function Blocks. Let us design the MSM bottom-up: What functionality do we require to implement the existing mode switches in software? First, we need a way for the data plane to invoke the MSM. We therefore need to be able to configure the trapping behavior of instructions. Compute instructions can trap as a side effect in some circumstances, like a division by zero. For syscall instructions, their entire purpose is to trap. What happens after such a trap is entirely up to the MSM code, which needs to inspect the trap cause and react on it. But whether an instruction traps or not should be configurable. It may offer interesting opportunities to configure trapping options for all instructions and even configure conditions like division by zero.

Second, memory accesses can raise exceptions within the memory management unit. Currently, page table entries contain permission bits with a pre-defined meaning. The MSM would enable more flexibility by turning the permission bits into small storage areas with no initial meaning and allowing MSM code to configure the MMU to implement a software-defined meaning. The MMU would offer programmable logic, whose input are the page table bits and properties of the memory access, like reading or writing. The function's single-bit result would determine whether an exception occurs.

Third, the MSM code needs to read and write data plane registers critical for control flow, especially the data plane instruction pointer. We think that instruction trapping, MMU configuration, and register access is sufficient to implement traditional user and kernel mode, as we illustrate below. However, for newer CPU modes like SGX, we would need to add specific function blocks to implement their behavior. The benefit of MSM is that it encourages designs, where the minimal primitives are added as composable function blocks, deliberately delegating complex interactions to software. For SGX and SEV, attestation of MSM code and configuration of keys for inline memory encryption may be all that is needed.

The MSM and Traditional Modes. A software-defined CPU according to this proposal would have no modes except for the MSM and regular data plane execution. We have to leave our usual thinking behind that the CPU is always in a specific mode, we just have data plane execution and the MSM. The MSM accesses the function blocks described above to alter the environment in which data plane execution occurs. Only the nature of the environment programmed by the MSM code turns the data plane execution context into a 'user mode' or a 'kernel mode'. These modes are now entirely a software construct emerging from the loaded MSM code. Thus, the MSM code needs to be loaded very early during the boot process. Within the scope of this paper, we leave open whether runtime changes should be allowed and how they would function.

However, when the traditional modes are defined in software, this software must now remember, what mode is currently active. Current CPUs remember the mode as part of their architectural state. This state now needs to be managed by the MSM code. Our thinking is that the MSM should have a small amount of freely usable CPU internal scratchpad memory available. To reduce complexity, the MSM should not have access to regular main memory at all. Otherwise, we would have to deal with the headaches of paging the mode that controls paging semantics. Reading and writing this scratchpad requires s small set of simple memory and control flow instructions as part of the control plane instruction set. State sharing between control plane and data plane is enabled by adding data plane instructions to access the MSM scratchpad. Of course, it is MSM-configurable whether these instructions work or trap.

Walking Through a Mode Transition. Assume we have a software-defined CPU and are currently running in an environment resembling traditional user mode. How would a system call work? What must the MSM code do to implement the user-kernel transition?

The user code issues a system call instruction, which has been configured by a previous MSM execution to trap. This trap invokes the MSM code, which first inspects the trap cause to learn that a transition to kernel mode is requested. It then reads the current mode information stored in the scratchpad to learn that the transition makes sense. MSM then stores the data plane instruction and stack pointer in the scratchpad and sets them to hardcoded values of the kernel entry point. Finally, the MSM reconfigures the MMU permission logic to make kernel pages available. The CPU exits MSM mode and resume data plane execution at the new instruction pointer. The execution environment now resembles traditional kernel mode. The kernel can save the previous 'user' instruction and stack pointer by reading them from MSM scratchpad. We leave the inverse transition back to user mode as an exercise to the reader.

4 DISCUSSION

Are the presented mechanisms sufficient to implement all existing modes? We believe the concept works, but additional function blocks may be needed for some of the more complex modes. Virtualization for example requires to turn nested paging on and off.

Introducing New CPU Modes. With everything defined in software, we can change the strict hierarchical ordering of today's modes. We can arrange them differently, like having virtualization modes (guest kernel, guest user) available within host user mode. We can expose the flushing of micro-architectural state as an MSM function block to offer different trade-offs between side channel mitigation and mode switch latency. We can add in-process sandboxes, which are interesting to just-in-time compiled code like JavaScript: The JIT code region is mapped writable when the JIT executes. But before invoking the created code, we change to a lightweight sandbox mode, where JIT pages are executable, but no longer writable. Other ideas could include nested paging inside user code, which has been explored by Dune [5]. Programming language implementations may also make use of customizable trap behavior, for example to implement integer overflow protection with trapping instead of instruction-based checks.

Microkernels. You may wonder, whether the MSM code is not some kind of microkernel. It certainly shares properties of a microkernel and we think a simple partitioning kernel running a fixed, small number of applications can indeed be implemented entirely in MSM. But the MSM is not capable of dynamically altering page tables or managing memory-backed resources like thread control blocks, simply because MSM has no access to main memory. But it is an interesting thought experiment to apply the microkernel concept of running system services in user code to MSM: Can we augment MSM by having data plane services, for example to emulate missing hardware features in software?

CISC and RISC. In a way, the principles of RISC have only been applied comprehensively to the CPU data plane. There, we moved from CISC's complex, pre-packages bundles of functionality to orthogonal, composable building blocks. The MSM concept tries to do the same for the CPU control plane: reshape a set of complex, pre-packaged CPU modes into orthogonal, composable building blocks orchestrated by software.

5 CONCLUSION

This proposal of a software-defined CPU will result in years of new ASPLOS papers! We must build a performant MSM, which will be fun for the architecture community. We can have arbitrary and dynamic mode graphs instead of the boring hierarchical layers, which will be fun for the operating systems community. We can have programming languages that invent their own modes, which will be fun for the compiler community. We should start today.

REFERENCES

[1] 2022. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. (Accessed on February 23, 2022).

[2] 2022. Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. (Accessed on February 23, 2022).

[3] 2022. Trustzone for Cortex A – TEE Reference Documentation. https://www.arm.com/why-arm/technologies/trustzone-for-cortex-a/tee-reference-documentation. (Accessed on February 23, 2022).

[4] Andrew Baumann. 2017. Hardware is the new Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal (Eds.). ACM, 132–137. https://doi.org/10.1145/3102980.3103002

[5] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[6] Jonathan Corbet. 2015. Memory protection keys. https://lwn.net/Articles/643797/. (Accessed on February 23, 2022).

[7] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *meltdownattack.com* (2018). https://spectreattack.com/spectre.pdf

[8] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *meltdownattack.com* (2018). https://meltdownattack.com/meltdown.pdf

[9] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.