Software-Defined CPU Modes

Michael Roitzsch
Till Miemietz
Barkhausen Institut
Dresden, Germany

Christian von Elm
Technische Universität Dresden
Dresden, Germany

Nils Asmussen
Barkhausen Institut
Dresden, Germany

ABSTRACT

Our CPUs contain a compute instruction set, which regular applications use. But they also feature an intricate underworld of different CPU modes, combined with trap and exception handling to transition between these modes. These mechanisms are manifold and complex, yet the layering and functionality offered by the CPU modes is fixed. 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. We show how such a design would function and explore the advantages it enables. We believe that pushing all existing modes under a common design umbrella would enforce a cleaner structure and more control over exposed functionality. At the same time, the flexibility of software-defined modes enables interesting new use cases.

KEYWORDS

processor modes, mode transitions, microcode

ACM Reference Format:
Michael Roitzsch, Till Miemietz, Christian von Elm, and Nils Asmussen. 2023. Software-Defined CPU Modes. In *Proceedings of Workshop on Hot Topics in Operating Systems (HotOS '23).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595894

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 and monitor modes enable isolated security contexts [5]. In the recent past, the trend of adding CPU modes[1] perpetuated: SGX [4], MPK [10], and SEV [3] are among 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 [7]. The last years have shown how brittle CPU implementations already are [15, 17, 22], and the new modes certainly have their share of weird interactions [20]. In addition, the complexity associated with a mode is present, whether the system uses it or not. Consequently, the isolation promises between modes and between protection domains implemented by a mode (like address spaces) have become more difficult to reason about.

At the same time, the systems community has no shortage of ideas for new CPU modes: MPK is being abused for intra-application sandboxing [14], nested paging in user mode would help with garbage collection [8]. But although details of existing modes are implemented in microcode, they are inseparably linked to the silicon. Although firmware is ultimately software, the operating system cannot influence this microcode to disable unneeded modes or to add new ones.

This paper poses the question: What if we could? What if we approached the construction of CPU modes from a completely different perspective? 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 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, because programs are represented by an instruction stream, which the CPU consumes and interprets. The instructions invoke CPU-internal function blocks like arithmetic logic units (ALUs) or load-store units. We experience the resulting state changes as program execution. All code works this way, applications as well as operating systems. We call this portion of CPU operation the *CPU data plane,*

HotOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595894

[1]In this paper, we define "CPU modes" quite broadly as different execution semantics and thereby a switch between modes as a stateful change to these semantics.

because its main observable effect is the transformation of program state in memory.

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. Its main effect is the transition of one CPU execution state to another, like the switch from user mode into kernel mode. These transitions are hardwired and complex, which is in stark contrast to the orthogonal and composable function blocks of the data plane.

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-packaged bundles of functionality to orthogonal, composable building blocks that a compiler stitches together. Software-defined CPU modes envision the same for the control plane: reshape a set of complex, pre-packaged CPU modes into orthogonal, composable building blocks orchestrated by software. Once a programmable mode substrate is in place, all CPU modes needed in a concrete system can be configured into the CPU at runtime.

But what is the potential gain of this increased flexibility? Surely we do not conduct this thought exercise only for a more aesthetic CPU design. We believe that this flexibility will help in two ways: First, it can help reduce the overall system complexity by only establishing modes that are needed for a given software stack. If a hypervisor mode or a trusted execution mode are not needed, then not configuring them into the CPU reduces the system's attack surface.

On the other hand, extending a system with bespoke modes tailored to its use cases has potential to improve its security, efficiency, and flexibility. Inspiring examples are:

- Type II hosted hypervisors can benefit from nesting the guest kernel and guest user virtualization modes inside host user mode. This way, they would no longer require a detour through the host kernel for trap-and-emulate functionality.
- Dune [8] showed that virtualization can be used to provide applications access to privileged CPU features and thereby, for example, improve the performance of garbage collection. Software-defined CPU modes could enable the same benefits, but without the – in this case unwanted – drawbacks of virtualization: expensive VM entries and exits.
- Just-in-time compiled languages like JavaScript typically collide with security measures such as write-xor-execute [11]. A restricted sandbox mode inside user mode can isolate JIT code by swapping write and execute permissions.
- Similarly, when mixing code written in type-safe and non-type-safe languages, the latter can be isolated by an in-app sandbox. Currently, Intel MPK is used to emulate such a mode [14].
- Configuring the side-channel mitigations when transitioning from user to kernel mode can enable trade-offs between isolation and mode switch latency [18].
- Mapping devices into user mode like SPDK and DPDK requires lightweight kernel-like isolation, when the device mapping is shared between distrusting processes [19].

This list is certainly not exhaustive, but it illustrates that software-defined CPU modes offer interesting opportunities for system-level improvements and are thus worth exploring.

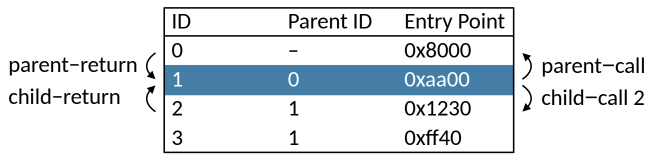
3 IMPLEMENTATION VARIANTS

In this section, we first examine how a mode transition currently works in order to extract requirements for a software-defined mode switch. We then propose two implementation variants with increasing flexibility, but also increasing deviation from the way CPUs operate today. We initially focus the explanation on the well-known transition between user and kernel mode, but also discuss a novel sandbox mode intended to restrict portions of applications within user mode. This additional mode illustrates the flexibility of the implementation variant by demonstrating how to add a mode that does not exist in current commodity CPUs.

3.1 Current Mode Switch Behavior

We discuss a mode switch in three phases: trigger, reconfiguration, and state transfer. Switches between user mode and kernel mode are triggered for different reasons. Hardware interrupts and exceptions are hard-wired in the CPU to force a transition into kernel mode. Exceptions can be raised as instruction side effects like touching unmapped memory regions or division by zero. Furthermore, dedicated instructions exist to explicitly trigger the transition like `sysenter` or `syscall` on x86 CPUs.

After initiating the switch and before executing the first instruction in the new mode, the CPU is being reconfigured internally by built-in mode-switch logic. These reconfigurations affect multiple orthogonal subsystems of the CPU. Specifically from user to kernel mode, instruction availability and page-table permission-bit evaluation changes. Instructions such as those accessing machine registers like the root page-table pointer or other security-critical functionality are fully available in kernel mode. In user mode, executing those instructions would be denied and raise an exception. Page tables contain permission bits in each page-table entry. Those bits are evaluated by the memory-management unit (MMU) and include a kernel or supervisor flag. With this flag, memory pages can be configured to be accessible from kernel mode, but inaccessible from user mode. After the mode



ID	Parent ID	Entry Point
0	–	0x8000
1	0	0xaa00
2	1	0x1230
3	1	0xff40

parent–return
child–return
parent–call
child–call 2

Figure 1: Examplary mode configuration table with parent and child calls (current mode in blue).

transition, accessibility of certain memory pages therefore changes as the MMU now evaluates page-permission bits according to their kernel mode semantics.

The mode switch is completed by a limited transfer of state between the exited and the entered mode. On x86 CPUs, the user-mode value of instruction and stack-pointer registers must be made available to kernel-mode code, otherwise they cannot be recovered at a later kernel exit. This is because the mode transition overwrites these registers with pre-defined values for kernel entries. x86 transfers these registers by pushing their values to the stack, other architectures use different means like dedicated transfer registers. When all of these steps are finished, the mode switch is completed by fetching the first instruction from the kernel entry point, to which the instruction-pointer register now refers.

From this description, we can identify independent control-plane responsibilities that are touched upon during the switch:

- Some instructions like `sysenter` switch modes as their main behavior, for others like division by zero it is a conditional side effect. Which instructions trigger a switch and under which conditions?
- To which mode do the various trigger reasons switch?
- During the switch, trigger behavior is reconfigured: instructions may raise an exception in the exited mode, but execute normally in the entered mode, or vice versa. Which instructions change in which way?
- MMU page-permission interpretation is reconfigured: a permission flag may raise a page fault in the exited mode, but not in the entered mode, or vice versa. How does the permission bit semantics change?
- Which state of the exited mode is made available to the entered mode?

The behavior of any concrete mode transition is currently baked into hardware. Implementing software-defined CPU modes means offering fine-grained configuration hooks for these responsibilities.

3.2 Mode Configuration Table

Our first proposal is to represent the currently active execution mode with a configuration vector. In its simplest form, the hierarchy of available modes is initialized at system boot by feeding them into a processor configuration table. A simplified example of such a table is depicted in Figure 1. Each table row represents one processor mode and instructions can trigger transitions upwards or downwards in this table.

Each mode configuration vector consists of individual fields for aspects of the CPU control plane that are now configurable instead of hardcoded. In particular, each vector contains:

- the id of the mode's creator or parent,
- an entry-point address to which the CPU jumps when the mode is entered,
- trap behavior for instructions,
- MMU permission semantics for page-table entries to specify accessible pages.

The table is organized such that the initial mode after boot is located in the top row and less privileged modes follow in subsequent rows. Instructions like `sysenter` and `sysexit` are replaced with two new instructions: `parent-call` and `parent-return`, which trigger a mode transition upwards and downwards in the configuration table, respectively. State transfer works similarly to current processor implementations by pushing the instruction pointer of the exited mode to the stack or special registers. Additionally, we propose the instructions `child-call` and `child-return`. In contrast to `parent-call`, `child-call` takes the destination mode as an argument and the hardware ensures that it's a child of the current mode. Additionally, `child-call` hides state like the instruction pointer from the callee and `child-return` implicitly resumes execution at the prior location.

Implementing Existing and New Modes. Traditional user and kernel mode can be implemented by loading a mode table with two rows, one for kernel mode and one for user mode. The entry-point address for kernel mode points to the same starting point for kernel code that the traditional `sysenter` instruction would use. The system boots into kernel mode and the first user process can be launched using `parent-return`. MMU permissions are configured such that supervisor pages are accessible in kernel mode, but not in user mode. Applications issue system calls using `parent-call` and the kernel uses `parent-return` to resume execution of the application.

An additional sandbox mode within user mode can easily be configured with a third table row. Applications continue to run in user mode by default, but can use `child-call` to enter the sandbox, whereas the sandboxes uses `child-return` to return to the application. Memory isolation between application and sandbox can be achieved by changing the MMU permission semantic for the sandbox mode (e.g., adding another bit to the page-table entries). Furthermore, certain instructions can be configured to trap when used inside the sandbox.

Many Modes in Tree Structure. An initial design might place the mode table in dedicated CPU configuration memory that can be written once at boot time before entering the first mode. However, when systems want to configure many modes, a better solution would be to use a portion of DRAM and configure beginning and end of the table with CPU configuration registers. In order to allow quick mode transitions without costly DRAM accesses, we propose a Mode Lookaside Buffer (MLB). The similarities to page tables are obvious.

The design discussed so far only allows for a linear hierarchy of modes. However, by amending each mode vector with a parent field, the stored table logically becomes a tree of modes. The `parent-call` instruction now transitions to the parent of the current mode based on the parent field, whereas the `child-call` instruction takes the child id as an argument to support multiple children per mode. The hardware is responsible to check whether the desired mode is in fact a child mode of the current mode.

Now with an unbounded number of modes, a bit per mode in page-table entries is not sufficient to control memory access anymore. However, todays page-table construction conflates physical-to-virtual address translation and access-control responsibilities into one mechanism. Similar to other works [1, 6], we suggest to disentangle the two, dedicating the current radix-tree-based page tables exclusively for address translation and a separate data structure for access control. It needs to be evaluated whether another radix tree or a simple region list is the better solution for access control. This separation allows each mode to reference its own permission data structure. The current Translation Lookaside Buffer (TLB) would gain a companion Permission Lookaside Buffer (PLB) which caches permission information.

Mode Configuration at Runtime. Adding an extra bit to each mode configuration vector allows to distinguish, whether a mode can change the mode configuration table at runtime. In a traditional user/kernel setup, the kernel would have this privilege and would thus be able to add new modes like the aforementioned sandbox mode at runtime. System calls can be devised by which applications can instruct the kernel about the desired mode configuration for additional per-application modes. To prevent that all applications running in the same user mode can enter the sandbox mode, the kernel clones the mode of the calling application on creation of the first child mode. Therefore, by default all applications run in the same user mode, but applications that created a child mode receive their own copy of user mode with a child mode attached to it.

3.3 Software-defined Mode Transitions

The mode configuration table already provides several advantages over the state of the art like more flexibility for applications and faster transitions into and out of sandboxes. However, the implementation of mode transitions and in particular the state transfer between the exited and entered mode is defined by the CPU (e.g., with microcode) and unchangable by software. For example, a transition between a virtual machine and the hypervisor requires more state transfer than the instruction and stack pointer. This is because the virtual machine also works on its own copy of the control registers and therefore they need to be included in the state transfer. Another example is an enclave, where it is desirable to encrypt most of the state in enclave mode before switching to the host OS.

Control-Plane Code. Addressing use cases like virtualization and enclaves therefore requires an even more flexible approach. We propose to replace mode transitions completely with software. That is, whenever a mode transition is required the CPU executes a special piece of software, called control-plane code, that implements the mode transition by reconfiguring the CPU and transferring the state. We see two variants for its implementation. First, a dedicated mode called *mode-switch mode*, which executes the control-plane code. This approach looks self-contradicting at first glance, because we are essentially proposing to add yet another CPU mode. However, 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.

Alternatively, the CPU could be equipped with a separate control-plane processor that executes the control-plane code. Whenever a mode transition is required, the CPU signals the control-plane processor, which executes the code and thereby reconfigures the function blocks of the CPU to perform the mode transition. Afterwards, the CPU simply continues executing with the new configuration (possibly with changed instruction pointer, MMU permissions, etc.). Similar to FlexSC [21], using a dedicated core for control-plane code could be faster than a transition between modes on the same core. Furthermore, a clear separation between the CPU and the core for control-plane code could protect the control-plane code from side-channel attacks.

Keeping State. Both variants need 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. Additionally, the control-plane code must remember the currently active mode. Current CPUs remember the mode as part of their architectural state. We propose control-plane code to have a small amount of freely usable scratchpad memory available. To reduce complexity, the control-plane code 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 a 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 scratchpad. Of course, it is control-plane configurable whether these instructions work or trap.

We believe that these facilities are sufficient to also support use cases like virtualization and enclaves. For virtualization, the control-plane code would be responsible to save and restore all registers including control registers. However, nested paging would still need to be a hardware building block for efficiency reasons. Similarly, enclave control-plane code could encrypt and decrypt the state during mode transitions to the host OS, but needs hardware-accelerated encryption for this state and for main memory to make enclaves efficient. Additionally, the control-plane code can be responsible for key management and the measurement state for attestation.

4 INITIAL RESULTS

We have built an early prototype of the mode configuration table approach to study its feasibility and performance. We used the gem5 hardware simulator [9] and the 64-bit RISC-V instruction set architecture as a foundation. We extended gem5 by the instructions described above, as well as an MLB and PLB. The PLB allows to configure memory protection for each CPU mode with variable-sized memory regions. We also ported xv6 [2], a simple re-implementation of UNIX version 6, to make use of `parent-call` and `parent-return` for system calls instead of `ecall` and `sret`. We configure gem5 to use a CPU and system clock frequency of 1 GHz, the out-of-order CPU model, and classical caches. The CPU has a single core containing 32 KiB L1 instruction cache, 32 KiB L1 data cache and 512 KiB L2 cache and accesses gem5's DDR3 1600 8x8 model as main memory. The MLB and PLB both contain 64 entries and are software-loaded. Since the MLB is only required during mode transitions, it is searched sequentially. In contrast and like the TLB, the PLB implements a fully parallelized search and adds no additional latency as it needs to be consulted for every instruction. All experiments are performed with 10 warmup rounds and 90 repetitions.

In a first experiment, we evaluated how calls to the OS using parent calls perform in comparison with traditional system calls and function calls. The results are depicted in Figure 2 (left). As expected, parent calls are slower than function calls, but faster than system calls. However, as these are early results based on simulation, we merely conclude that parent calls are competitive to traditional system calls.

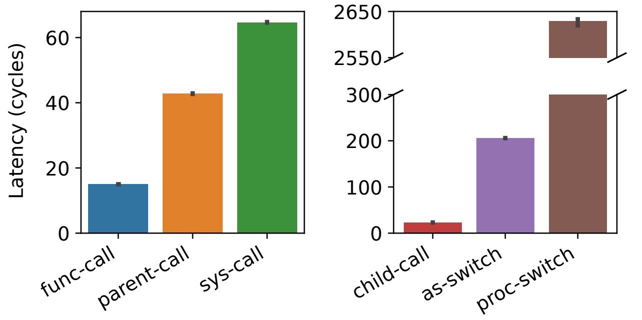


Figure 2: Latency of parent calls to implement syscalls (left) and child calls to implement sandboxing (right).

We performed a second experiment to study how child calls can enable fast transitions into and out of sandboxes. Mode configuration tables enable sandboxes by allowing applications to create a new CPU mode for a part of their application. The application can change the protection for parts of its address space to restrict the sandbox to specific regions. The instructions `child-call` and `child-return` are used to enter and leave the sandbox, respectively. We compare this approach (`child-call` in the figure) with the typical way to isolate parts of an application on traditional architectures: different address spaces and process switches to enter and leave the sandbox (`proc-switch`). Since the performance of process switches depends on the operating system, we also compare to a raw address-space switch (`as-switch`) with a tagged TLB and therefore no TLB flush, which can be seen as the lower bound on traditional architectures. The results are depicted in Figure 2 (right), which shows the performance for entering and leaving a sandbox and compares the three mentioned approaches. As can be seen, `child-call` and `child-return` are two orders of magnitude faster than two process switches and still one order of magnitude faster than two address-space switches. The primary reasons for the performance advantages are that child calls avoid address-space switches and do not involve the OS kernel.

5 DISCUSSION

The flexibility and early performance results look promising, but also raise several questions that we want to briefly discuss.

System Security and Complexity. Moving more responsibility from hardware to software raises the question of whether it increases or decreases the overall system complexity and security. On the one hand, isolation of applications on todays systems is already not only dependent on the hardware, but also on the privileged software layers, which would not change with our proposal. On the other hand, it is unclear how the system complexity changes if more functionality is provided in software instead of in hardware. However, we believe that current CPUs have evolved beyond a manageable complexity and should therefore be simplified. We also believe that a single and general hardware mechanism that can be used in various ways by software is easier to verify and reason about than the current state: various different hardware mechanisms that have been designed independently, leading to unforeseen interactions [20].

Memory Protection. CPU modes inherently interact with memory protection due to the desire to restrict modes differently. We proposed to separate translation and protection and to only relate protection with modes. This separation allows more lightweight per-mode protection of specific memory regions instead of requiring different address spaces even when only the protection should differ. However, it remains to be studied whether variable-sized and fine-granular regions or same-sized and coarse-grained regions are preferable.

Flexibility and Performance. Both the mode configuration table and the control-plane code approach offer advantages over traditional CPU-mode designs. Mode configuration tables provide more flexibility for software and showed promising early performance results. However, as state transfer is still hardwired, use cases like virtualization or secure enclaves would still need to be baked into hardware. While software-defined mode transitions have the flexibility to cover these use cases, it remains to be seen whether the performance is sufficient. Considering that current CPUs implement mode transitions and enclaves partially in microcode, we believe that a software-based implementation is feasible.

Compiler-initiated Mode Switches. We believe that compilers can become one of the primary users of software-defined CPU modes. If applications define their own modes and switch between them without involving OS-specific primitives, compilers can sandbox parts of applications without manual intervention of the developer. For example, instead of generating function calls into an untrusted library, the compiler can introduce a new mode for the library with restricted memory access and generate child calls instead, similar to Glamdring with SGX enclaves [16]. Like for function calls, the compiler defines the calling convention between these isolated parts and potentially uses a dedicated memory area, accessible by caller and callee, to exchange data.

Related Work. The Alpha architecture [12] features PALCode, which can software-define instructions. PALCode was designed to accelerate OS primitives, but it cannot implement new CPU modes. Metal [13] is an early-stage software implementation of CPU modes, but proposes to trap on individual memory accesses to implement protection.

6 CONCLUSION

The systems community every now and then comes up with ideas that would benefit from a new, bespoke CPU mode, but without software-configurable modes, these ideas remain thought experiments. We think the examples we have briefly enumerated here together with our reasonable initial results demonstrate the potential of software-defined CPU modes. Realizing this idea requires a concerted effort of the CPU design, operating system, and compiler communities. Should we choose to overcome the inertia of the status quo, this concept could spark a whole new era of operating system and programming language design opportunities.

ACKNOWLEDGMENTS

This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101092598 (COREnext) as well as the German Research Council DFG through grant 502457159 (FOSSIL). It is also financed on the basis of the budget passed by the Saxon State Parliament in Germany.

REFERENCES

[1] [n. d.]. Mill Computing – Memory. http://millcomputing.com/w/index.php?title=Memory&oldid=4227. (Accessed on May 22, 2023).
[2] [n. d.]. Xv6 for RISC-V. https://github.com/mit-pdos/xv6-riscv. (Accessed on January 27, 2023).
[3] 2022. AMD Secure Encrypted Virtualization (SEV). https://developer.amd.com/sev/. (Accessed on February 23, 2022).
[4] 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).
[5] 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).
[6] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L Shaw, and Robert NM Watson. 2017. Separating translation from protection in address spaces with dynamic remapping. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems.* 118–124.
[7] 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.* ACM, 132–137. https://doi.org/10.1145/3102980.3103002
[8] 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.* USENIX Association, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay
[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (May 2011), 1–7. https://doi.org/10.1145/2024716.2024718

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

[11] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. 2010. JITsec: Just-in-time security for code injection attacks. In *Benelux Workshop on Information and System Security (WISSEC)* (Nijmegen, The Netherlands).

[12] Digital Equipment Corporation 1996. *Alpha Architecture Handbook* (3 ed.). Digital Equipment Corporation, Maynard, MA, USA.

[13] Fatemeh Hassani. 2020. *Implementation of the Metal Privileged Architecture*. Master's thesis. University of Waterloo, Waterloo, ON, Canada. https://uwspace.uwaterloo.ca/handle/10012/16197

[14] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *European Conference on Computer Systems (EuroSys)* (Rennes, France). ACM, 132–148. https://doi.org/10.1145/3492321.3519582

[15] 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

[16] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, P Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for Intel SGX. USENIX.

[17] 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

[18] Till Miemietz, Maksym Planeta, Viktor Reusch, Jan Bierbaum, Michael Roitzsch, and Hermann Härtig. 2022. Fast Privileged Function Calls. In *11th Workshop on Systems for Post-Moore Architectures (SPMA)* (Rennes, France). https://www.barkhauseninstitut.org/fileadmin/user_upload/Publikationen/2022/202204_Miemietz_SPMA_FastCalls.pdf

[19] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. 2021. Simurgh: A Fully Decentralized and Secure NVMM User Space File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (St. Louis, MO, USA). ACM. https://doi.org/10.1145/3458817.3476180

[20] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Annual Network and Distributed System Security Symposium (NDSS)*.

[21] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924946

[22] 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.