Metal

An Open Architecture for Developing Processor Features

Siyao Zhao
University of Waterloo
Waterloo, Ontario, Canada
sy2zhao@uwaterloo.ca

Ali José Mashtizadeh
University of Waterloo
Waterloo, Ontario, Canada
mashti@uwaterloo.ca

ABSTRACT

In recent years, an increasing number of hardware devices started providing programming interfaces to developers such as smart NICs. Processor vendors use microcode to extend processors' features such as Intel SGX and VT-x. This enables processor architects to quickly evolve processor designs and features. However, modern processors still lack general programmability as microcode is inaccessible to system developers. Developers still cannot define custom processor features. We argue that processors should expose this capability to developers, which enables new operating system and application designs.

We propose Metal, a novel open architecture that enables system developers to define custom instructions with microcode level overhead. We implement a prototype of Metal on a 5-stage pipelined RISC processor with minimal additional hardware resources. We demonstrate Metal's capability by building a variety of architectural extensions such as user defined privilege levels. We also discuss other potential applications and future directions for Metal.

ACM Reference Format:

Siyao Zhao and Ali José Mashtizadeh. 2023. Metal: An Open Architecture for Developing Processor Features. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595915

1 INTRODUCTION

A trend in computing is to enhance programmability across the system stack to provide developers with more features and flexibility. For example, the Linux kernel offers eBPF [4] to modify system behavior without needing to write kernel modules. Smart NICs and storage devices enable custom computation inline with network and storage processing before reaching the OS. Programmable P4 switches [2] expand programmability outside the host to process and manipulate network traffic flows.

In recent years, the number of extensions to popular instruction set architectures (ISA) has skyrocketed. Most of these extensions are implemented largely in microcode [13], e.g., Intel SGX and VT-x.

Unfortunately, processors have seen little effort to enable programmability for developers. Processor vendors commonly use microcode to implement processor programmability, which is complex even for processor architects. Higher level intermediate instruction encodings, such as Intel XuCode, enable processor architects to write microcode almost as efficiently as native assembly. However, processor vendors have yet to open up this programmability to developers.

The growing popularity of microcode as a way to deploy processor features raises security concerns [13, 15]. Due to the lack of transparency, microcode auditing is difficult, which leads to security through obscurity. Recent works [32, 33] manage to reverse engineer x86 microcode and inject vulnerabilities. Implementing complex software concepts, such as virtualization, in low level microcode also increases the probability of introducing security vulnerabilities.

We group microcode into two categories: *horizontal* and *vertical*. Horizontal microcode is relatively simple and decodes into a few micro-ops that expose an optimized hardware function. However, implementing complex architectural features using horizontal microcode is difficult. To address the complexity, processor vendors develop intermediate instruction encodings called vertical microcode, such as Intel XuCode [1], IBM's Millicode [24] and Alpha's PALcode [19]. Vertical microcode is much closer to the processor's native assembly and often thousands of micro-ops long. This simplifies complex architectural feature development.

To promote the evolution of processor architectures and new operating system designs, we propose that processor vendors should provide an open architecture to system developers to define new instructions. This enables developers to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotOS '23, June 22–24, 2023, Providence, RI, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595915

build new architectural features and applications. Implementing this open architecture should involve relatively simple logic and a few basic hardware components.

This paper presents Metal, a processor hardware extension that enables developers to rapidly evolve the processor's architecture through *software* instead of hardware. Metal offers a vertical microcode like programming interface consisting of the native instruction set plus a few Metal specific instructions. To achieve microcode level overhead, we dedicate a RAM for storing Metal code which is collocated with the processor's instruction fetch unit.

Unlike other intermediate instruction encodings discussed above, Metal enables not only processor architects but also system developers to create new architectural extensions. Processors expose the fundamental building blocks, which Metal uses to create higher level extensions and abstractions. OSes can share a common set of extensions across different architectures to enhance compatibility and portability.

We provide a proof-of-concept Metal implementation on a 5-stage pipelined RISC processor, adding 14% more logic cells. Our processor exposes various architectural features to Metal, such as direct physical memory access and instruction interception. This enables us to implement high level architectural extensions, such as user defined privilege levels and custom page tables. We also discuss other potential Metal applications, such as virtualization and security enclaves.

2 DESIGN

Metal is an open architecture that exposes underlying microarchitectural features to system developers. Developers can use Metal to add architectural extensions to processors such as user defined privilege levels and custom page tables. Metal introduces a new privileged operation mode *Metal mode* and a microcode-like programming interface *mcode*, which developers use to program custom extensions. Unlike microcode, mcode consists of the host processor's native assembly plus several Metal specific instructions.

In Metal mode, the processor executes mcode and accesses internal architectural features. At boot time, Metal loads a collection of mcode subroutines called *mroutines*, which extend the architecture's instruction set. Metal assigns each mroutine with a unique entry number, which serves as entry points into Metal mode.

Critically, Metal stores mroutines in a RAM collocated with the processor's instruction fetch unit to offer microcode level overhead. The low overhead enables the development of latency sensitive extensions that are otherwise only possible with microcode. The RAM partitions code and data into separate segments, which hold mroutines and mroutine private data. Accesses to the RAM do not alter processor caches as the locality of the RAM already offers cache-like access speed. This also prevents side channels on the RAM.

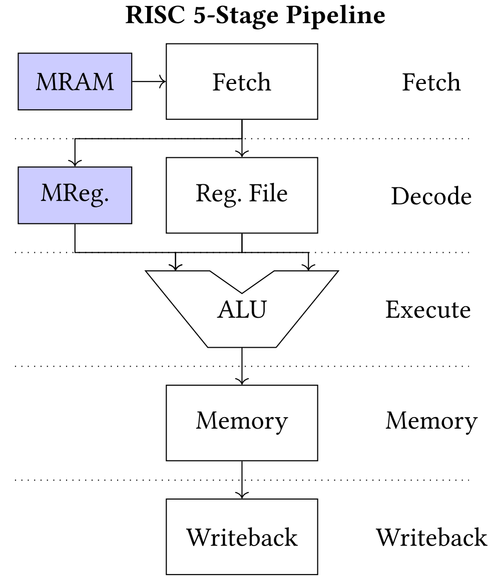

Figure 1: The overview of our Metal enabled 5-stage pipelined RISC processor. Metal adds a RAM that holds mcode and a Metal register file for storing Metal's internal state.

Instruction	Description
`menter`	Enter Metal mode
`mexit`	Exit Metal mode
`rmr`	Read Metal register
`wmr`	Write Metal register
`mld`	Load from MRAM
`mst`	Store to MRAM

Table 1: New Metal instructions. Applications executing in normal mode invoke `menter` to enter Metal mode. The rest are only available in Metal mode.

We develop a proof-of-concept implementation of Metal on a 5-stage pipelined RISC processor. Figure 1 shows the high level workflow and additional components. We add a small RAM (*MRAM*) to store up to 64 mroutines and a Metal register file (*MReg.*) containing 32 Metal exclusive registers *m0*–*m31* to store Metal's internal state.

2.1 Metal Instruction Extension

Table 1 lists the new Metal instructions. Applications executing in normal mode call `menter` with an mroutine entry number to enter Metal mode. Once in Metal mode, the processor stores the caller's return address into Metal register m31 and executes the corresponding mroutine.

`menter` is not a privileged instruction in the traditional sense as Metal does not define privilege levels besides normal mode vs. Metal mode. Developers can freely define custom privilege levels that suit their use cases by checking callers' privilege levels in mroutines.

Upon entering Metal mode, the processor gains full access to architectural features and the rest of Metal instructions. Applications use `rmr` and `wmr` to read and write Metal registers. `mexit` exits Metal mode and resumes the execution from the stored address in Metal register m31.

Metal provides `mld` and `mst` instructions to load and store from MRAM's data segment. The data segment holds mroutine private data used for bookkeeping, e.g., the pointer to the page table structure for custom page tables.

Metal mroutine programming resembles embedded system development. To avoid allocation failures, developers must statically allocate resources including Metal registers used across invocations or the MRAM data segment. Metal mroutines are non-interruptible for simplicity and to allow Metal to implement interrupt delivery mechanisms. Static allocation and non-interruptibility improve performance, security and reliability by eliminating potential resource exhaustion and simplifying mroutine verification.

Metal only defines a few new instructions and leaves the flexibility of exposing architectural features to the processor. Processors expose architectural features as either Metal instructions, control registers or memory mapped IO.

2.2 Fast Metal Mode Transition

Metal mode transitions must be fast to make extending the instruction set practical. Metal achieves low overhead in two ways. Critically, we collocate MRAM that stores mroutines with the processor's instruction fetch unit to achieve microcode level overhead.

We also optimize `menter` and `mexit`. When entering and exiting Metal mode, the processor replaces `menter` with the first instruction from the target mroutine during the decode stage. We stall instruction fetching to also replace `mexit` with the next instruction in the original stream. When returning to the application, Metal achieves virtually zero overhead.

2.3 Exposing Architectural Features

A key feature of Metal is accessing low level architectural features through mroutines. In our implementation, the processor exposes the following architectural features to Metal that enable the example applications in Section 3. The processor exposes these features to Metal through instructions and memory mapped registers only available in Metal mode.

Access to Physical Memory. Modern processors manage memory through paging. The memory management unit (MMU) translates virtual pages to physical pages via page tables. However, there is no mechanism to access unmapped physical memory. Our implementation allows mroutines to bypass paging and access physical memory in Metal mode. This enables developers to implement custom page tables.

	Baseline	Metal	%Change
Number of Wires	170,264	197,705	16.1%
Number of Cells	180,546	206,384	14.3%

Table 2: Hardware resources for adding Metal to our 5-Stage pipelined processor. The table provides an upper bound as modern processors are more complex.

Page Keys and Address Space IDs. Page keys provide an extra level of indirection for page permissions to accelerate batch permission changes. Address space IDs allow TLBs to cache multiple address spaces. Our processor exposes TLB modification instructions, and it supports page keys and address space IDs. This allows Metal to create arbitrary execution contexts and change page permissions rapidly.

Interrupt and Exception Delivery. Our processor delegates all exception and interrupt delivery to Metal. We assign specific mroutines to handle interrupts and exceptions. This allows Metal to define custom privilege levels and deliver interrupts to any privilege level, e.g., user level interrupts.

Instruction Interception. Our implementation allows intercepting any instruction with an mroutine. For instance, developers can intercept loads and stores dynamically to implement transactional memory or patch an insecure instruction at runtime.

2.4 Hardware Resources

To estimate the hardware resources required to build Metal. We develop a proof-of-concept 5-stage pipelined RISC processor in Verilog with and without Metal. We synthesize our processor using the Yosys [8] open source synthesis tool and the Synopsys [7] standard cell library.

Table 2 shows the hardware resources in terms of wires and cells used by our processor with and without Metal. In our implementation, Metal only consumes 14% more cells and 16% more wires. This serves as an upper bound of the required hardware resources as most modern processors have more complex pipelines with multiple issue.

3 APPLICATIONS

To demonstrate the capabilities of Metal, we developed multiple architectural extensions. This helps us understand what systems can benefit from Metal's encapsulation of architectural features and the ability to intercept instructions with low overhead.

```
kenter:
 addq $t0, $zero, 1            # Set m0 (kernel mode)
 wmr $m0, $t0
 andq $a0, $a0, 0x00ff         # Limit to 256 syscalls
 sllq $a0, $a0, 3              # Compute entry point
 ldq $t0, $t0, SYSCALL_TABLE
 addq $t0, $t0, $a0
 rmr $ra, $m31                 # Save return address to $ra
 wmr $m31, $t0                 # Load entry point to $m31
 mexit                         # Return to kernel mode

kexit:
 wmr $m0, $zero                # Clear m0 (user mode)
 wmr $m31, $ra                 # Move user IP into $m31
 mexit                         # Return to user mode
```

Figure 2: The assembly of system call entry (`kenter`) and exit (`kexit`) mroutines.

3.1 User Defined Privilege Levels

Metal enables new OS privilege separation models beyond the basic user mode vs. kernel mode distinction. Developers can implement multiple privilege levels with defined transitions using mroutines.

To demonstrate this, we first implement a traditional kernel-user privilege model in Metal. We provide two mroutines, `kenter` and `kexit`, which transition from userspace to the kernel and back. We reserve the Metal register `m0` to hold the current privilege level. All mroutine calls that access or modify privileged resources, e.g., the TLB, are protected by a privilege check that triggers an exception if violated.

The assembly of both mroutines is shown in Listing 2. `kenter` takes a system call entry number as the parameter in GPR `a0`. It updates the current privilege level in `m0`, computes the syscall entry point, and jumps to the kernel system call entry point. We use temporary register `t0`, as defined in the ABI, to compute the entry point and save the userspace return address in register `ra`. `kexit` loads the address stored in `ra` and calls `mexit` to return to userspace.

In general, processor privilege switching involves setting architectural state and returning control to the target entry point regardless of the number of privilege levels. All privileged mroutines, such as interrupt handlers, in our traditional kernel-user model should check or set Metal register `m0` and raise an exception in case of privilege violation.

In-process Isolation. Alternatively, applications can use multiple privilege levels internally to implement in-process isolation to protect sensitive data. For example, isolating sensitive cryptographic keys in OpenSSL from the rest of the application. On modern processors, in-process isolation usually requires a form of control flow integrity (CFI) [45] to protect the transition code. However, recent works show that CFI is inherently unsafe [20]. Metal enables developers to safely encapsulate the transition code without CFI.

3.2 Custom Page Tables

OSes can implement custom memory management data structures with Metal. The Linux kernel team has pressured multiple processor vendors to implement radix tree based page tables similar to x86. This comes at the cost of restricting multiple page size support into fewer buckets, leading to poor performance [40]. With Metal, OSes can implement custom memory management data structures in the page fault exception handling mroutine.

Critically, the proximity of MRAM to the instruction fetch unit enables fast exception dispatching with costs similar to microcode implementations. This greatly closes the performance gap between hardware and software managed TLBs with the flexibility of user defined data structures.

We implement a radix tree based page table using direct physical memory access and exception handling provided by the processor. In a few lines of assembly, we walk an x86-style radix tree on page fault. We populate the processor's TLB mappings from the page table. If the page is not present or the access violates the page protection, we deliver the exception to the OS.

3.3 Transactional Memory

We also implemented a proof-of-concept transactional memory extension based on software transactional memory (STM) techniques. We created several new mroutines: `tstart` starts a transaction, `tabort` aborts the transaction, and `tcommit` commits the transaction. We intercept all memory access instructions within a transaction and invoke `tread` and `twrite` instead, which perform and record the memory accesses. Upon `tcommit`, all accessed memory addresses within the transaction are inspected for conflict.

The benefit of using Metal is that neither compilers nor developers need to replace loads and stores with calls into an STM library. Instead, Metal turns on and off interception of loads and stores at runtime when it needs to track memory accesses for transactional memory. Our implementation is under 100 instructions and closely resembles TL2 [18].

3.4 User Level Interrupt

User level interrupt is a new processor feature that allows unprivileged userspace processes to handle hardware interrupts and perform userspace IO. Intel plans to support user level interrupt in their next generation processors [5]. User level interrupt is especially useful for high performance kernel bypass libraries such as DPDK [3] and SPDK [6].

Currently, both DPDK and SPDK interact with NICs or storage devices by polling in user mode, which consumes all cores used by the application. With user level interrupt, such applications only need to be notified via interrupts

when data is available from underlying devices, reducing CPU occupancy and power consumption.

Metal supports user level interrupt by handling the processor's interrupt delivery. When an interrupt occurs, Metal invokes specific mroutines to optionally redirect the interrupt to processes running at lower privilege levels. The mroutines ensure that the target process to receive the interrupt is currently running on the core and interrupt the process without changing the privilege level. Developers control whether a specific privilege level is allowed to process interrupts.

3.5 Other Applications

Metal is a great platform to develop architectural extensions by combining and encapsulating architectural features. We present a few potential architectural extensions including capabilities, security enclaves, virtualization and control flow protection, which are often implemented in microcode. We also discuss nesting Metal to accommodate multitenancy.

Hardware Capabilities. Capability based security is not a new idea. The IBM System/38 [25] and Intel iAPX 432 processors [35] implement capabilities in hardware using microcode. The CHERI processor [46] implements a capability based security model with a coprocessor. Similar to prior systems, Metal can support capabilities by writing mroutines to create and manipulate domains and capabilities.

Security Enclaves. Security enclaves offer code and data integrity. Intel SGX [14], AMD Secure Encrypted Virtualization (SEV) [29], and ARM TrustZone [11] all implement enclaves using both hardware and microcode. Sanctum [15] and Sanctorum [34] offer software security enclaves with little hardware support. PrivateCore, a security startup, builds hypervisor-based memory encryption that offers similar functionality to AMD SEV without hardware support [42].

Metal's flexibility in defining privilege levels enables developers to implement enclave extensions. Developers create a trusted execution layer that runs at a higher privilege level than the host OS. After Metal loads and verifies an enclave, the enclave runs in the trusted execution layer which the host OS cannot access.

Virtualization. Virtualization consolidates multiple workloads into a single machine. Virtualization was first introduced in the IBM VM/370 [16] and implemented in numerous other architectures including VAX [23], Alpha [30], POWER [26], Intel VT-x [27], AMD SVM [10], ARM [12], SPARC [44], and MIPS [39]. Most architectures implement virtualization in microcode. For example, the IBM zSeries virtualization extensions are implemented mostly in Millicode [24], and the Alpha hypervisor is built exclusively using PALcode [30].

Developers can use Metal to implement virtualization. For example, Metal allows hypervisors to implement nested page tables. Multiple privilege rings also provide better host-guest isolation. Privileged instructions can be intercepted and trapped by Metal for proper handling.

Control Flow Protection. Metal can offer similar application control flow protection as existing techniques such as shadow stacks [41] and control flow integrity [9]. Metal eliminates the compiler dependency for protecting key materials from existing CFI systems such as cryptographic control flow integrity [38]. Instead, applications can store cryptographic keys inside Metal registers or MRAM.

Nested Metal. Modern data centers often host multiple applications in separate VMs on a single machine. Metal should allow VMMs, OSes and applications to define their own mroutines for performance and security. For example, defining VM abstractions for a VMM and in-process isolation for a TLS enabled webserver.

The challenge is composing mroutines from different layers without undermining the integrity of other layers. For example, a higher guest OS layer can define custom memory management, but should not affect the lower VMM layer's memory management. Furthermore, the layered design also requires that mroutines are reentrant as instruction intercept can occur during mroutine execution. Reentrancy increases the difficulty of mroutine development and verification.

We are actively exploring nested Metal that supports multiple layers of mroutines where mroutines belonging to a layer can be swapped during a context switch. Interrupts propagate from lower to higher layers so that VMMs and OS kernels can decide which VM or application the interrupt belongs to. Instruction interception proceeds in reverse, with higher layers intercepting the instruction first so that applications can customize individual intercept behavior. The intercept propagates downward through layers that intercept the same instruction, which only occurs when the higher layer's intercept handling mroutine reuses the instruction.

4 DISCUSSION

Vendor Incentives. Metal provides an open architecture to processor architects and system developers to rapidly develop and evolve processor features. Existing processor vendors do not document microcode to protect intellectual property (IP). Metal proposes using vertical microcode in the native instruction set to separate IP concerns and provide a new layer for innovation.

We propose that processor vendors should provide developers with fundamental architectural features and delegate higher level abstractions and encapsulation to developers via Metal. Metal offers three major advantages to vendors.

First, vendors can implement architectural features in microcode to protect intellectual property while offering the

flexibility of Metal. Metal does not expose internal details of the processor's microarchitecture.

Second, processor vendors differentiate themselves by developing new performance enhancing architectural features. For example, software implementations of memory encryption, such as PrivateCore's, suffer from high CPU overhead. Processor vendors then introduce hardware memory encryption, such as Intel SGX, to significantly accelerate the process.

Third, delegating abstraction and encapsulation to Metal accelerates the adoption and evolution of new technologies. Metal enables system developers to encapsulate an architectural feature, e.g., memory encryption, into higher level extensions such as security enclaves and encrypted virtual machines based on end user needs.

Intel is replacing SGX with Trust Domain Extensions (TDX) [28] that resembles AMD SEV. Had Intel and AMD instead offered memory encryption as an architectural feature, system developers could have iterated on high level abstractions like SGX and TDX more rapidly.

Security. Currently, Metal disables interrupts in mroutines as interrupts complicate development and verification. Developers must consider the consequences of interrupts occurring at any instruction and mroutine reentrancy. For example, VMWare spent years making the world switch between the VMM and the ESX kernel safe to non-maskable interrupts (NMI), because the code had to handle interrupts while reconfiguring all processor state. Additionally, Intel and AMD recently announced solutions to problems with interrupt and exception handling dating back decades [36].

Recent processors suffer from timing and speculation side channel attacks such as Spectre [31] and Meltdown [37]. Metal does not cache MReg. or MRAM and shifts the responsibility to developers for main memory accesses. For sensitive mroutines, developers should insert speculation barriers and cache control instructions to eliminate side channels.

5 RELATED WORK

Processor architects commonly use microcode to simplify instruction decoding and support complex instructions consisting of hundreds to thousands of micro-ops.

Processor architects often find microcode limiting how quickly they can introduce and develop features. At least three intermediate instruction sets have been developed including Millicode [24] on the IBM zSeries, PAL code [19] on the Alpha and XuCode [1] on Intel's recent x86 processors.

IBM uses Millicode [24] to implement complex instructions and achieve backward compatibility. For example, the zSeries processors have two types of transactional memory. First, a hardware transactional memory that supports small transactions of just a few cache lines. Second, a hybrid software transactional memory written in Millicode that supports large transactions. Virtualization instructions and other complex ISA features are also written in Millicode.

Intel develops XuCode [1] as an intermediate microcode to aid their developers in implementing Intel SGX [14] features. XuCode is tightly coupled with SGX, running in the SGX reserved memory region. Rather than writing complex microcode, SGX developers implement features in a higher level instruction set.

Finally, Alpha PALcode [19] is unusual because it is written in native assembly, but still serves the same purpose as other intermediate microcodes. PALcode bridges the architectural features with the OS and firmware through an API. DEC supplies two PALcode implementations to support running Unix vs. VMS and Windows NT. Alpha implements its privilege model through PALcode, which allows the processor to support two rings on Unix and four rings on VMS. In many ways, PALcode is the inspiration for Metal. One major difference is that PALcode resides in main memory. A no-op PALcode call takes approximately 18 cycles on the Alpha [43], making it impractical to encapsulate or emulate low latency instructions, unlike Metal.

Historically, many architectures have provided access to their microcode. Despite the difficulties of microcode programming, this open architecture enables research and development including optimizing Prolog on the VAX [21] and improving game performance on the Xerox Alto [17].

Recently, researchers have reverse engineered newer processors. One group of researchers reverse engineered the AMD K8 and K10 microcode [33], and even created benign and malicious microcode updates [32]. Several researchers extracted the entire Intel microcode ROM and the keys used to encrypt microcode updates [22].

6 CONCLUSION

We propose that processor vendors implement an open architecture for developing architectural extensions. Vertical microcode, such as Intel's XuCode, IBM's Millicode and Alpha's PALcode, proves the viability of developing architectural extensions in intermediate instruction encodings. Metal gains inspiration from vertical microcode and offers a microcode-like programming interface for system developers and researchers to develop new architectural extensions. With compiler support, it can be practical to write hardware features in high level languages such as C.

ACKNOWLEDGMENTS

We thank Emil Tsalapatis for the valuable discussion towards the development of Metal. We thank Fatemeh Hassani for her master's thesis work on prototyping Metal in Verilog. We also thank the committee for their valuable feedback.

REFERENCES

[1] XuCode: An Innovative Technology for Implementing Complex Instruction Flows. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/xucode-implementing-complex-instruction-flows.html, May 2021.

[2] P4 Open Networking Foundation. https://opennetworking.org/p4/, Apr 2022.

[3] Data Plane Development Kit. https://www.dpdk.org/, Feb 2023.

[4] eBPF - Introduction, tutorials and community resources. https://ebpf.io/, Feb 2023.

[5] Intel Instruction Set Extensions Technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.htm, Feb 2023.

[6] Storage Performance Development Kit. https://spdk.io/, Jan 2023.

[7] Synopsys Standard Cell Libraries. https://www.synopsys.com/dw/ipdir.php?ds=dwc_standard_cell, Feb 2023.

[8] Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/about.html, Feb 2023.

[9] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[10] Advanced Micro Devices, Inc. Secure Virtual Machine Architecture Reference Manual. (33047), December 2005.

[11] Thaynara Alves and Don Felton. Trustzone: Integrated Hardware and Software Security. *Information Quarterly*, 3:18–24, January 2004.

[12] Arm Limited. Arm Architecture Reference Manual for A-profile architecture. (042523), April 2023.

[13] Andrew Baumann. Hardware Is the New Software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 132–137. ACM, 2017.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[15] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[16] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM J. Res. Dev.*, 25(5):483–490, sep 1981.

[17] Josh Dersch. The Xerox Alto Part 2: Microcode. https://engblg.livingcomputers.org/index.php/2017/06/23/the-xerox-alto-part-2-microcode/, June 2017.

[18] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, page 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] Digital Equipment Corporation. *PALcode for Alpha Microprocessors: System Design Guide.* May 1996.

[20] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 901–913, New York, NY, USA, 2015. Association for Computing Machinery.

[21] J. Gee, S. W. Melvin, and Y. N. Patt. The Implementation of Prolog via VAX 8600 Microcode. In *Proceedings of the 19th Annual Workshop on Microprogramming*, MICRO 19, page 68–74, New York, NY, USA, 1986. Association for Computing Machinery.

[22] Dan Goodin. In a first, researchers extract secret key used to encrypt Intel CPU code. https://arstechnica.com/gadgets/2020/10/in-a-first-researchers-extract-secret-key-used-to-encrypt-intel-cpu-code/, Oct 2020.

[23] Judith S. Hall and Paul T. Robinson. Virtualizing the VAX Architecture. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, page 380–389, New York, NY, USA, 1991. Association for Computing Machinery.

[24] Lisa Heller and M. Farrell. Millicode in an IBM zSeries processor. *IBM Journal of Research and Development*, 48:425 – 434, 06 2004.

[25] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. IBM System/38 Support for Capability-Based Addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, page 341–348, Washington, DC, USA, 1981. IEEE Computer Society Press.

[26] IBM Systems and Technology Group. *PowerISA Version 3.0 B.* March 2017.

[27] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2 (2A, 2B, 2C, & 2D): Instruction Set Reference, A-Z(325383-079US), March 2023.

[28] Intel Corporation. Intel Trust Domain Extensions. (343961), February 2023.

[29] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. *White paper*, 2016.

[30] Paul A. Karger. Performance and Security Lessons Learned from Virtualizing the Alpha Processor. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 392–401, New York, NY, USA, 2007. Association for Computing Machinery.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *Commun. ACM*, 63(7):93–101, jun 2020.

[32] Benjamin Kollenda, Philipp Koppe, Marc Fyrbiak, Christian Kison, Christof Paar, and Thorsten Holz. An Exploratory Analysis of Microcode as a Building Block for System Defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1649–1666, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse Engineering x86 Processor Microcode. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1163–1180, Vancouver, BC, August 2017. USENIX Association.

[34] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. Sanctorum: A lightweight security monitor for secure enclaves. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1142–1147, 2019.

[35] Henry M Levy. *Capability-based Computer Systems.* Digital Press, 1984.

[36] Linus Torvalds. x86 - why unite when you can fragment? https://www.realworldtech.com/forum/?threadid=200812&curpostid=200822.

[37] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[38] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[39] MIPS Tech, LLC. MIPS64 Architecture for Programmers. Volume IV-I: Virtualization Module of the MIPS64 Architecture, December 2013.

[40] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading)*, OSDI '02, page 89–104, USA, 2002. USENIX Association.

[41] Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Trans. Comput.*, 55(10):1271–1285, oct 2006.

[42] PrivateCore. PrivateCore: Home. https://privatecore.com/.

[43] Sebastian Schönberg. The L4 Microkernel on Alpha, Design and Implementation. Technical report, University of Dresden, September 1996.

[44] Sun Microsystems, Inc and Fujitsu Limited. *SPARC JPS2: Common Specification.* September 2003.

[45] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1221–1238, USA, 2019. USENIX Association.

[46] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 457–468. IEEE Press, 2014.