Limitations and Opportunities of Modern Hardware Isolation Mechanisms

Abstract

A surge in number, complexity and automation of targeted security attacks has triggered a wave of interest in hardware support for isolation. Intel memory protection keys (MPK), ARM pointer authentication (PAC), ARM memory tagging extensions (MTE), and ARM Morello capabilities are just a few hardware mechanisms aimed at supporting low-overhead isolation in recent CPUs. These new mechanisms aim to bring practical isolation to a broad range of systems, e.g., browser plugins, device drivers and kernel extensions, user-defined database and network functions, serverless cloud platforms, and many more. However, as these technologies are still nascent, their advantages and limitations are yet unclear. In this work, we do an in-depth look at modern hardware isolation mechanisms with the goal to understand their suitability for isolation of subsystems with the tightest performance budgets. Our analysis shows that while a huge step forward, the isolation mechanims in commodity CPUs are still lacking implementation of several design principles critical for supporting low-overhead enforcement of isolation boundaries, zero-copy exchange of data, and secure revocation of access permissions.

1 Introduction

Despite significant academic interest [22, 83, 88, 89] the performance of hardware isolation primitives remained a low priority in commodity CPUs for decades. On x86 machines, segmentation was deprecated as part of the transition from 32-bit to 64-bit addressing mode, leaving page tables as the only available isolation mechanism. Lack of hardware support was making fine-grained isolation prohibitively expensive. For example, the de-facto standard hardware isolation mechanism *page tables* requires 771 cycles for a cross-subsystem invocation on Intel CPUs (818 cycles on ARM) [1]. To put this number into perspective, modern network processing frameworks like DPDK [19] spend less than a hundred cycles per request [26].

The interest in isolation was revived by a recent surge in complexity and automation of targeted security attacks. Memory Protection Keys (MPK) and Extended Page-Table (EPT) switching with VM functions (VMFUNC) [42] deployed in modern Intel CPUs provide support for memory isolation with overheads gradually approaching the overhead of a function call [35, 58, 63, 82]. Both X86 and ARM are exploring hardware support for sub-page isolation [7, 20]. The latest ARM CPUs introduce 16-byte-granularity memory isolation with the Memory Tagging Extension (MTE) [7, 10]. Intel introduced support for 128 byte sub-page permissions (SPP) [20]. Finally, ARM implements pointer authentication (PAC), i.e., cryptographic signing of pointers in hardware, that can be used to implement both control-flow integrity [50] and subsystem isolation [57]. ARM Morello is the first silicon implementation of the CHERI capability model [89].

In contrast to traditional hardware isolation techniques (e.g., segmentation and page tables), the new primitives are designed to support lightweight cross-subsystem invocations (in the low hundreds and even low tens of cycles). Hence these new mechanisms bring a promise of practical isolation of small untrusted extensions and third-party code that require frequent communication with the rest of the system, e.g., browser plugins [2, 60, 64, 94], database extensions [16, 78], virtualized network functions [6, 36, 41, 55, 69, 73], Web applications [3, 23, 28, 44], serverless cloud and edge platforms [4, 5, 45, 67, 81], and operating system kernels [57, 61, 63].

However, as hardware and software approaches mature, questions of their practicality and relative advantages for isolation arise. On one hand, the overhead of switching the isolation boundary with a hardware instruction is becoming progressively lower, e.g., writing a `pkru` register that changes the current tag in the Intel MPK mechanism introduces an overhead of only 20-26 cycles [35, 66]. On the other hand, modern mechanisms rely on complex compiler or binary instrumentation to enforce isolation [35, 57, 82], introduce significant overheads on cross-subsystem invocations [49], and require high-overhead revocation mechanisms required to ensure safety of zero-copy communication [87, 90].

Our work does an in-depth look at modern hardware isolation mechanisms with the goal to understand their ability to support fine-grained isolation of subsystems with the tightest performance budgets. We first introduce a generic isolation scheme centered around the concept of low-overhead isolation and efficient zero-copy communication. To understand the benefits of each hardware isolation mechanism, we develop carefully optimized implementations of isolation schemes that leverage Intel MPK, ARM MTE, ARM PAC, and ARM Morello.

Our analysis shows that while a huge step forward, modern isolation mechanisms still lack multiple conceptual features that limit their practicality. ARM PAC and ARM MTE are inherently limited by the overhead of additional instructions that are needed to enforce isolation of heaps. Careful performance analysis demonstrates that even the most minimal compiler instrumentation, e.g., a single instruction that copies the MTE tag bits in front of every memory access adds overhead impractical for isolation of modern systems. All isolation schemes are limited by the overhead of saving and restoring general and extended registers which significantly affects the cost of cross-subsystem invocations. Intel MPK suffers from the inability to reflect passing of zero-copied memory regions across all cores of the system, which results in either a restrictive programming model (the buffers passed on one core cannot be accessed from other cores) or expensive cross-core synchronization similar to TLB shootdown. Tag-based schemes like MPK and MTE suffer from the limitation on the number of isolated subsystems. Additionally, MTE suffers from the overhead of retagging which in our experiments is only marginally faster than copying. Capability schemes like CHERI are inherently limited by the lack of centralized metadata that is required for implementing revocation of rights and "move" semantics, i.e., ensuring that the caller looses access to the objects on the heap that are passed to the callee. Hardware architectures that keep access rights in registers, e.g., Intel MPK and CHERI, are facing another inherent limitation: it is impossible to perform revocation of rights across the cores (active capabilities can be retained in registers of other cores).

Our observations allow us to identify several principles that are critical for design of practical, low-overhead isolation mechanisms with support for efficient zero-copy communication. *Software transparency*: architectural mechanisms should avoid relying on expensive compiler instrumentation which becomes prohibitive in modern systems (Intel MPK and ARM Morello satisfy this principle, but ARM MTE and AMR PAC do not). *Core-coherent synchronization of rights*: hardware should support synchronization of access permissions across cores of the system, which is essential for implementing both a general programming model in which memory regions exchanged across isolation boundary are available to all threads and even more important for global revocation of rights (our analysis shows that ARM MTE is the only set of isolation extensions that implements this principle). *Revocation*: support for revocation should not be overlooked as a design principle (pointer-centric isolation architectures like CHERI fail to support revocation which requires an expensive software instrumentation to track propagation of capabilities in software). In the face of frequent communication, the ability to revoke access rights to a specific memory region is as important as grant them.

The above principles, which are largely overlooked in the current generation of hardware isolation mechanisms, can help hardware architects to shape the design of the future hardware isolation mechanisms. Moreover, we argue that current isolation mechanisms can be extended to implement these principles.

2 Background

In 1977, the final Multics report proposed isolating subsystems as a means to improve reliability and security of the kernel [12]. Since then, for over four decades, a range of user-level and kernel projects explored the possibility to provide practical, low-overhead isolation through hardware mechanisms [32,46,91], programming language safety [9,14,39,62,85], and software fault isolation (SFI) [30,56,68,71,86,96,97].

The very first isolation mechanisms root back to the early time-sharing machines which utilized hardware segmentation and later paging to support efficient virtualization of memory. Careful hardware optimizations like the virtually indexed and physically tagged L1 cache enabled TLB lookups to be performed in parallel with the rest of the pipeline, hence ensuring zero-overhead isolation between address spaces. Unfortunately, efficient switching between isolated subsystems was never a goal [25].

Multi-core architectures created an opportunity to isolate computations by running them on separate CPU cores [11,37,61,76]. Cross-subsystem invocations were implemented as messages over coherence protocols that transfer cache-line messages between the cores. While faster than paging-based address space switches, cross-core invocations remained expensive. A minimal call/reply invocation requires four cache-line transactions and takes 448-1988 cycles depending on whether the line is transferred between the cores of the same socket or over a cross-socket link [61]. Unfortunately, to provide low-latency communication, both the caller and callee have to poll for incoming coherence messages during a cross-core invocation. Hence, one of the two cores involved in the cross-subsystem invocation is wasting time checking for the message in a tight loop.

Trying to remove hardware overheads form the invocation path, a range of projects explored isolation mechanisms that enforce isolation entirely in software through techniques of SFI and language safety. Software fault isolation (SFI) strikes a balance between the ability to support isolation in a broad spectrum of programming languages, overheads of enforcing isolation and cost of cross-subsystem communication [2, 15, 27, 54, 72, 86, 94]. SFI enforces segment-like boundaries (i.e., access to a contiguous region of memory) through additional bounds checks in front of all memory access instructions [86]. Building on efficient SFI isolation techniques, WebAssembly became a de facto mechanism to enable near-native performance for a range of resource-demanding Web applications [23,28,44], data streaming platforms [5], serverless and edge platforms [4,45,67,81] as well as providing practical isolation of browser [2,60,64,94] and kernel [15,27,54,75] extensions implemented in unsafe languages that are prone to low-level memory safety violations and vulnerabilities.

The idea of using language safety for isolation roots back to the early language-based operating systems [9, 13, 40, 62, 85]. In a safe language an invocation between isolated subsystems can continue on the same stack (safety ensures isolation of the stack across subsystems) and does not require saving and restoring general and extended registers (calling conventions enforce saving and restoring registers between the caller and callee, and exception handling, i.e., *unwind*, mechanisms allow recovery from a fault in an untrusted subsystem). Unfortunately, language-based isolation relies on a huge TCB of the language itself (e.g., correctness of the type system), it's runtime, trusted libraries, compiler, build environment, etc. [62].

2.1 Modern Isolation Primitives

In the last decade, commodity CPUs introduced a diverse range of hardware primitives aimed at support of practical, fine-grained memory isolation. Some of the new primitives, like Intel extended page table (EPT) switching with VMFUNC and Intel sub-page protection (SPP) are a poor fit for systems with frequent communication (VMFUNC has a high overhead of switching the EPT and requires complex virtualization infrastructure to enforce isolation [63], SPP controls only write accesses which is insufficient for enforcing confidentiality). Other mechanisms, however, bear the promise of improving performance of lightweight isolation schemes.

Intel MPK Memory protection keys (MPK) is a new isolation mechanism introduced by Intel in SkyLake CPUs. MPK allows one to enforce isolation within a single address space, i.e., a single page table, by tagging individual pages with a 4-bit protection key (saved in the unused bits of the pagetable entry). A special register, `pkru`, holds a bitmap that allows access to a combination of tags (i.e., any combination from none to all is possible by setting individual bits in the bitmap). The `pkru` register specifies the access rights for each protection key with two bits per key (access disable and write disable). The read or write access to a page is allowed only if the value of the `pkru` register matches the tag of the page. Crossing between subsystems is performed by updating the bitmask in the `pkru` register, which is a fast operation taking 20-26 cycles [35, 66].

Isolation with MPK requires control over all `wrpkru` instructions throughout the code of the program to prevent unauthorized transitions between address spaces. In the past control over `wrpkru` was demonstrated with either binary rewriting [82] or dynamic validation of all `wrpkru` instructions with hardware breakpoints [35]. Also, MPK enforces checks only on data accesses but does not limit control flow transitions which opens the door for numerous system-interface level attacks that require expensive enforcement [18].

ARM MTE Starting with ARMv8.3-A, ARM SoCs introduce support for memory tagging extensions (MTE) that allow partitioning the address space into 16-byte regions that are colored with one of the 16 tags. The hardware maintains a table that stores mapping between addresses and tags allowing access to the region only if the tag of the pointer (the tag is stored in the upper bits of the pointer matches the tag of the memory region). MTE itself does not directly support isolation – the attacker can change the upper bits of the pointer that contain the tag. To enforce isolation, it is possible to combine MTE with techniques of software fault isolation (SFI), i.e., rely on binary rewriting or compile-time instrumentation to enforce a specific tag on every load and store operation.

ARM PAC Starting with ARMv8.3-A, ARM SoCs support cryptographic pointer authentication (PAC). PAC implements the ability to cryptographically sign a pointer and store the signature in the "unused" upper bits of the pointer. The signature is generated from 1) the pointer value, 2) a secret key protected by the operating system, and 3) a 64-bit program-defined "signing context" that allows the isolation scheme to restrict the use of a pointer in a custom way, for example, allow using the pointer only if the value of the stack pointer (`sp`) is identical at the moment of signing and authenticating the signature. A signed pointer cannot be used directly, but instead has to be authenticated with the same secret key and context. If either the pointer, its signature, or the context is different from the values used during signing, the authentication results in an invalid pointer value that triggers a hardware exception when used. PAC is a powerful mechanism that can be used to enforce control flow [50], spatial and temporal [29, 48] safety and isolation of subsystems [57].

ARM Morello ARM Morello is an experimental architecture that implements the CHERI capability model [33, 89]. It extends all general-purpose registers on AArch64 to be capabilities which include bounds and permissions in addition to addresses, and adds new instructions to support loading and storing of capabilities in memory. Memory operations against a capability are checked against its bounds and permissions, triggering a hardware exception if the constraints are violated. New capabilities can only be derived from existing ones, and Morello guarantees *capability monotonicity*, meaning that a new capability cannot provide access that exceeds the capability it's derived from.

To guarantee the unforgeability of capabilities in both registers and memory, Morello adds protected tag bits that indicate their validity. Each 16-byte memory location is associated with a hidden tag bit indicating whether a valid capability is stored. Similarly, each capability register contains a tag bit which is cleared when the register is modified in a way that violates capability monotonicity.

In Morello, a capability can be *sealed* which causes further changes (address, permissions, or bounds) to invalidate it. For example, a sealed function capability has its address set to the beginning of the function, with its bounds covering the entirety of the function code. An adversary cannot modify the capability to point to the middle of the function, despite the new address being within bounds. Branching to the capability, however, *unseals* it and sets the program counter to the unsealed capability. The function runs from the fixed entry point and can derive further capabilities from the Program

Counter Capability (PCC).

To preserve compatibility with unmodified AArch64 code that isn't capability-aware, Morello introduces the Default Data Capability (DDC) which is used for regular loads and stores. This provides coarse-grained isolation and allows the developer to gradually transition to fine-grained isolation by adding `__capability` annotations to pointers. Furthermore, Morello adds a new execution state known as the *Restricted mode* which has its own Default Data Capability as well as Stack Capability. When the processor is in Restricted mode, accesses to DDC and the stack register are automatically switched to the restricted counterparts.

Intel CET Intel introduced control-flow enforcement technology (CET), a hardware feature to mitigate ROP-style attacks (return-oriented programming, jump-oriented programming, call-oriented programming) by enforcing coarse-grained control flow integrity. It consists of a 1) shadow stack (`SHSTK`) to protect the return addresses that can be corrupted by buffer-overflow attacks and, 2) indirect branch tracking (IBT) that protects the forward control flow of the program. `SHSTK` records the return addresses in a hardware-protected stack region along with the regular stack; when `ret` instruction is executed, the return addresses are compared to generate an exception if there is a mismatch. `SHSTK` provides write-protected pages (using unused combination of read, write, and dirty bits in the pagetable) to store return addresses.

3 Threat model

We assume that hardware isolation mechanisms are used to implement process-like isolation boundaries across mutually mistrusting subsystems that implement a larger system, i.e., individual network functions that form a single service chain [6, 36, 41, 55, 65, 69, 73], loadable kernel extensions and device drivers [15, 38, 54, 61, 63, 79], etc. Specifically, the isolation architecture protects the state of each subsystem, i.e., its heap, data region, stack of each thread, etc., from accidental and malicious accesses by other subsystems, and provides a way for controlled communication in which subsystems can invoke each other interfaces. We assume that we trust the hardware implementation of the isolation mechanism itself, the compiler instrumentation pass, small TCB that implements passing and revocation of rights in the call gate trampolines. We leave side channel attacks outside of the scope of this work.

4 Design principles for efficient isolation

Historically, system designers chose to give up isolation for performance due to high overhead of isoaltion mechanisms. In modern systems that are designed operate at the speed of modern high-throughput I/O interfaces, i.e., process millions of requests per second, tight performance budgets create several unique requirements:

Low-overhead enforcement To be practical, isoaltion is allowed to impose only minimal overhead. In the past, segment and page-based isolation schemes enforced isolation boundaries with no additional overhead. Modern mechanisms, however, is less transparent. ARM PAC and MTE rely on a compiler pass that adds additional instructions to enforce isolation along with the hardware. Similarly, as we demonstrate below, while designed to avoid software support, CHERI capability architecture requires software support to track propagation of capabilities and implement efficient revocation. Unfortunately, even the fastest SFI implementations introduce significant performance impact on the isolated system (we provide a detailed breakdown of SFI overheads on x86 and ARM machines in Section 6).

Fast switching of isolation boundaries Fine-grained isolation comes at a cost of frequent crossings of isolation boundaries. Historically, overheads of cross-subsystem invocations remained high [46]. A typical invocation required a transition into a privileged execution mode to wtich the switch the address space (even a well-optimized sequence took hundreds of cycles [1]). Fortunately, novel hardware isolation primitives, provide support for switching the isolation boundary that approaches an overhead of a function call. The software however should be carefully optimized to leverage low-overhead hardware primitives. To minimize the overhead of cross-subsystem invocations, we implement them as synchronous transitions that do not change the thread of execution between caller and callee subsystems. Specifically, upon the invocation the caller saves its state on the stack, switches into the callee subsystem, picks a new stack inside the callee subsystem, and continues execution inside the callee.

Zero-copy passing of data Isolation of I/O intensive systems, e.g., operating system device drivers [21, 54, 61, 63], network processing frameworks [65], databases [47], etc., requires frequent passing of data between isolated subsystem. In such systems overheads of copying data between subsystems are prohibitive. An isolation mechanism should support low-overhead zero-copy passing of data acros isolated subsystems.

"Move" semantics and revocation The requirement to support zero-copy results in a unique challenge: the need to revoke or transfer access rights from caller to callee when the reference to an object is passed in a cross-subsystem invocation. Such "move" semantics is critical for preventing a range of time-of-check-time-of-use attacks which allow the caller to manipulate the object after it was passed to the calle. Internally, the ability to "move" the access right relies on the ability to revoke the access right from the caller. Revocation is surprisingly challenging. First, isolation schemes like CHERI allow unrestricted propagation of capabilities and hence revocation requires either a pass over the entire memory of the subsystem [87, 90] or as we demonstrate in this work compiler instrumentation to track propagation of all capabilities in memory. Second, revocation should be enforced across the cores. This is challenging as cores can have capabilities loaded in registers. As a result, synchronization

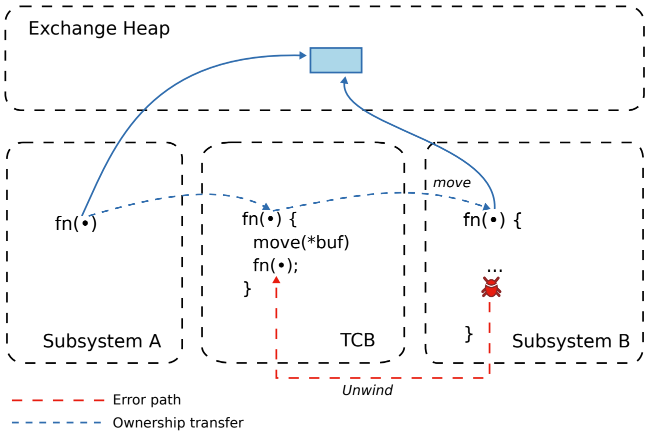

Figure 1: Heap isolation and unwind

of access rights requires an expensive inter-processor interrupt (IPI) or alternatively a restricted programming model in which processing of a specific object is pinned to one CPU core.

In the sections below we implement the following isolation scheme. To fully leverage the low-overhead nature of the hardware mechanism, we implement a *migrating threads* model of invocation [31] that avoids overheads of dispatching messaes and switching the threads, i.e., the same thread of execution transitions between caller and callee address spaces. To support clean termination and unloading of crashing subsystems, we enforce *heap isolation* invariant across subsystems, i.e., subsystems never hold pointers into each others private heaps [40, 62]. Specifically, we orchestrate isolated subsystems as a collection of isolated private heaps and a special special shared *exchange heap*—a heap that allows allocation of objects that can be exchanged across subsystems. Moreover, objects on the shared heap are owned by exactly one subsystem and are moved between them on cross-subsystem invocations. This allows us to avoid a case when objects left in an inconsistent state by the crashing subsystem are accessible on the shared heap by other, healthy subsystems. Furthermore, single-ownership on the shared heap provides natural semantics for implementing zero-copy communication if the underlying isolation mechanism supports it. The IPC subsystem tracks ownership of objects on the shared exchange heap (in the following sections we show several ways of enforcing single ownership and providing ownership tracking specific to each isolation mechanism). When a subsystem crashes, the TCB deallocates all objects on the shared heap owned by the subsystem and deallocates its shared heap. The unwind mechanism allows us to return errors from the crashing subsystem to its callers.

5 Isolation with Modern Mechanisms

To understand advantages and limitations of modern isolation mechanisms we develop several isolation schemes that leverage recent hardware extensions: Intel MPK, ARM MTE, ARM PAC, and ARM Morello.

5.1 Intel MPK

To enforce isolation we tag private heaps of individual subsystems with one of the 15 available tags (tag 0 is privileged and allows unlimited access to the entire address space and hence is reserved for the TCB). To enforce control flow we combine MPK with Intel CET. We rely on indirect branch tracking (IBT) to protect the forward edge of indirect transitions and utilize the write-protected shadow stack to protect the return edges. Attacker can potentially find a validated control flow entry inside the TCB that might contain a gadget with the `wrpkru` instruction. We there instrument indirect branch instructions and call instructions with a memory load from the target address to a temporary regsiter. This triggers validation of the MPK tag for the loaded memory.

CET allows us to enforce control flow integrity on both forward and backward edge without additional overheads of introducing instruction bundling and masking forward and return transitions to the beginning of a bundle.

To implement support for zero-copy communication, we support allocation of special regions of memory on the shared exchange heap that are also tagged with one of the available MPK tags. On cross-subsystem invocations the IPC trampoline changes the current tag granting access to the callee's heap as well as updates access permissions for buffers passed as arguments (i.e., it revokes access from the caller and granting it to the callee). This organization allows us to implement single-ownership on the exchange heap, i.e., only one subsystem can access each shared region at a time, on cross-subsystem invocations the buffers are "moved" between subsystems.

The above isolation scheme has several limitations inherent to MPK. First the total number of isolated subsystems and exchange buffers is limited to 15. Second, the cross-subsystem invocation updates the tag only on the CPU core on which invocation is performed, and hence the buffers on the exchange heap are accessible by only that CPU. This means that multithreaded applications cannot access shared exchange buffers from different cores, and potentially require a copy into the private heap that is accessible from all cores.

MPK trampoline Since we need to change accessible tags on cross-subsystem invocations, MPK trampoline needs to update the current value of the `pkru` register. Naive implementation of the trampoline would require two `wrpkru` instructions on both call and return paths – one to switch into the TCB and then another one to switch into the callee. To avoid overhead of expensive `wrpkru` instructions, we leverage the ability to save information about the caller's tag on the CET stack and restore it after invocation returns.

Specifically, we first read the current MPK permissions with the `rdpkru` instruction and check that the caller has permissions for the regions on the shared heap it is passing to the callee (Listing 1, lines 2–4). We then, utilize a small thunk trampoline to reserve space on the protected CET stack and save the current value of the `pkru` register there (lines 5–9). On

```
1    ; check buffer ownership before entering B
2    rdpkru ; rax contains permissions of A
3    and rax, r10 ; r10 has the buffer(s) A wants to pass to B
4    assert rax == 0 ; A has the permissions of the buffer(s)
5    call .reserve_ssp ; decrement ssp and rsp by 8 bytes
6  reserve_ssp:
7    pop ; restore rsp
8    rdsspq r9 ; copy ssp to r9
9    wrssq [r9], rax ; save pkru on the top of the shadow stack
10   ; grant buffer ownership to B
11   mov rax, PKRU_B
12   xor r10, 0xFFFFFFFFFFFFFFFF
13   and rax, r10
14   ; switch to B
15   wrpkru
16   ; locate stack
17   ...
18   call dispatch
19   ...
20   ; return path
21   ; load pkru of caller from the shadow stack
22   rdsspq r10
23   mov rax, [r10]
24   mov r10, 0x1
25   incsspq r10 ; increment ssp by 1 x 8 bytes.
26   ; switch back into A
27   wrpkru
28   ; restore the stack, restore register state...
```

Listing 1: MPK trampoline between two isolated subsystems, `A` and `B`

return from the calee, we read the saved value of `pkru` from the CET stack, and restore it (lines 22–27). Here we assume the case when the callee "borrows" the buffer on the exchange stack and returns it back when the invocation returns.

To perform safety checks, we maintain read-only metadata about liveness of the callee allowing the caller to perform the liveness check before entering the trampoline. Similarly, the callee checks that the caller is alive before returning, and if not calls into the TCB instead. To implement unwind, the TCB accesses the shadow stack to unwind execution to the next "live" subsystem in the return chain.

Discussion In the past MPK was suggested as an in-process isolation mechanism without control-flow enforcement (CFI) [35, 82]. While plausible for simple isolation scenarious (one isolated subsystem and no zero-copy passing of buffers), MPK alone becomes an easy target for a variety of attacks wihtout control flow enforcement. Without CFI, all `wrpkru` instructions are reachable to the attacker. They can redirect control flow to steal buffers passed in zero-copy invocations from other subsystems, break control flow between subsystems (e.g., in a call chain of from `A` to `B`, and then to `C` an attacker can return from `C` directly into `A`, and even return into a random subsystem `D`). CFI not only eliminates the need in complex binary rewriting required to protect MPK sandboxes from control flow attacks [18,35,82] but also allows for semantically complex transfer of rights on cross-subsystem invocations.

5.2 ARM MTE

To enforce isolation with MTE, we tag private heap of the program with one of 15 the available tags. Since the tag is stored in the upper bits of the pointer (memory address of the load or store instructions) attacker can change the bits of the tag by overflowing the pointers. We enforce the tag for each

```
...
; load value at x2 to x1
and x3, x2, #0xFFFFFFFFFFFF ;clear pac
lshr x4, x2, #48 ;size + pac is the index to metadata table
lshr x5, x2, #56 ;size stored at upper bits of the pointer
shl x5, -1, x5 ;use size to create a bit mask
and x5, x5, x2 ;mask with size to get the start of object
shl x4, x4, #3; metadate table offset = index * 8
ldr x6, [$metadate_tbl, x4] ;load object's metadata
autda x5, x6 ;autda will SIGILL when fails
ldr x1, [x3]; load uses ptr with no pac
...
```

Listing 2: ARM PAC Memory Check (AArch64 ASM)

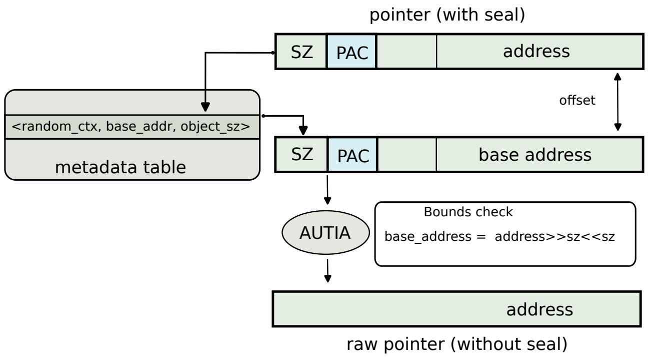

Figure 2: Enforcement with PAC

memory access by copying it from a reserved register into the address register with a bit-field `bfi` instruction.

MTE checks are not enforced for memory accesses relative to the stack pointer. We, therefore, disable the use of the stack pointer in the program and instead replace it with another general register. Similarly, MTE checks are not enforced for control flow transitions. We rely on techniques of Native Client to enforce all control transfers to a specific segment [72]. Specifically, we enforce instruction bundling and alignment to 16 byte boundary and enforce all control flow transitions to stay within the segment and land on the beginning of the bundle.

To support unwinding of execution from a faulting subsystem, the trampoline first switches into the TCB that records state of the caller and then enters the callee subsystem.

Zero-copy communication MTE provides a way of supporting zero-copy communication across isolated SFI subsystems. Specifically, it is possible to use MTE to enforce both boundaries of a private subsystem heap as well as to control access for fine-grained memory objects passed across subsystems. We tag entire private heap can be tagged with the subsystem's MTE key. Objects allocated on the shared exchange heap can be passed across subsystems. To ensure fault isolation between subsystems, we enforce single ownership of objects on the shared heap. Upon invocation the TCB in the trampoline first checks the ownership of the memory region by reading the tag and comparing it with the tag of the caller, and then changes it to the tag of the callee. Combined with single ownership, MTE provides support for reclamation of resources allocated on the shared heap.

5.3 ARM PAC

In contrast to MTE and MPK that provide access control over the address space, PAC provide a way to check validity of individual memory pointers [29, 48, 57]. To utilize pointer authentication, we allocate all shared and private heap objects with an alignment that matches the size of the object rounted up to the nearest power of two (Figure 2). We then use unused PAC bits (63-56th) of the pointer to store the size of the object as a power of two. The pointer and signing context includes: the base address bits of the pointer with size bits in upper bits, and a random identifier which we generate when the object is allocated (Listing 2). To ensure liveness of the object, we store the context in a global metadata table indexed by the PAC signature (if we detect a collision in the table we pick a new random identifier to re-generate a new PAC signature that indexes into a new entry). Before each load and store, we authenticate the pointer by first looking up its context in the global metadata table (authentication fails if any of the PAC signature, size bits, or the address itself are changed). If the pointer participates in a pointer arithmetic, we do not change the PAC since we use only base bits of the address for signing. If the pointer arithmetic operation leads outside of the signed power of two region the base bits of the pointer change and the pointer fails authentication. When object is deallocated the metadata is updated to store an invalid context. If the object is moved to another subsystem in a cross-subsystem invocation, the pointer is re-signed with a new random metadata value (this effectively revokes all aliases that might remain in the caller subsystem making them invalid).

5.4 ARM Morello

Morello provides fine-grained isolation through explicit invocation of capabilities as well as coarse-grained isolation through the executive and restricted Default Data Capabilities (DDCs). To transition between the executive and restricted modes, sealed function capabilities are used for both the forward and return edges. As in MPK and MTE, we separate the memory into a shared exchange heap and private heaps, and allow objects on the exchange heap to be *moved* between domains in a zero-copy manner.

Capabilities may be freely duplicated on the instruction set level, and a domain can store capabilities in memory and subsequently load them. To enforce move semantics, we track capability stores by instrumenting them with a compiler pass. The compiler pass inserts calls to a trusted runtime which adds the store destination to a hash map. On the IPC boundary, the trampoline traverses the map to revoke in-memory copies of moved capabilities.

Capabilities in Morello are resident in registers, meaning that a thread may retain access to a region of memory even after it's deallocated by another thread in the same domain. To implement move semantics, we associate each capability to a shareable buffer with an owning thread. In each capability, the user-defined permission bits are used to store the ID of the owning thread. We store the current thread ID as a sealed capability in the Compartment ID (`cid`) register. Like capability stores, we also instrument capability loads to ensure each thread can only load capabilities that it owns.

```
...
// save caller RSP
mrs x15, rsp_el0
str x15, [c14]

// switch to callee RSP
add c14, c12, x13 // c14 = callee_cap + x13
ldr x15, [c14]
msr rsp_el0, x15

// switch to callee RDDC
msr rddc_el0, c12

// prepare target capability
mov x9, x10
seal c30, c12, rb

// branch to target in restricted mode
blrr c30
...
```

Listing 3: ARM Morello DDC Switching

```
  ldr c1, [x2]

  ; Begin instrumentation
  mrs c3, cid_el0
  gcperm x5, c1
  and x5, x5, 0b111100 ; Extract user-defined permission bits
  cmp x5, x3
  b.eq passed
  brk 0
passed:
  ; End instrumentation
  ...
```

To reduce the overheads resulting from inserted checks, we combine coarse-grained isolation through DDCs with explicit capabilities for objects on the shared exchange heap. This enables the use of existing code with minimal adaptation. Most loads and stores of domain-local data are integer operations and therefore not subject to instrumentation.

Morello trampoline For each valid cross-subsystem transition, we generate a trampoline page which contains both the trampoline code as well as a *context*. The context includes the destination address as well as the DDCs of the caller and callee domains. The TCB creates a sealed capability to the trampoline page and passes it to the caller domain.

To evaluate the overheads of switching between the Executive and Restricted modes, we build two trampolines for Morello. The first trampoline remains in Restricted mode and loads the DDC of the callee from the context by offsetting the now-unsealed Program Counter Capability. The second trampoline switches to Executive mode and loads the DDC of the callee using the Executive DDC.

6 Analysis: Limitations and Opportunities

To reason about the performance of different isolation mechanisms, we leverage several hardware platforms. For x86 we use a Framework Laptop 13 with Intel Core i7-1165G7 and 32GB of DDR4 RAM. The machine runs 64-bit NixOS Linux with a 6.5 kernel configured without any speculative execution attack mitigations (`mitigations=off`) reflecting the trend of recent Intel CPUs addressing a range of speculative execution

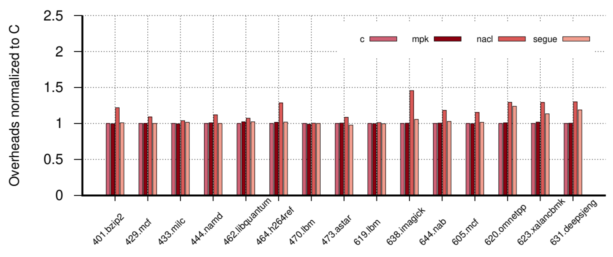

Figure 3: specCPU2017 and specCPU2006 on x86

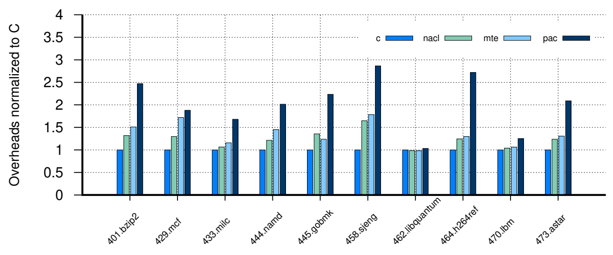

Figure 4: specCPU2017 and specCPU2006 on AArch64

attacks in hardware. In all experiments, we disable hyper-threading, turbo boost, CPU idle states, and frequency scaling to reduce variance in benchmarking.

To evaluate isolation mechanisms based on ARM MTE and PAC, we use a Pixel 8 phone with a Tensor G3 SoC (1x Cortex-X3, 4x Cortex-A715, 4x Cortex-A510) [34]. The phone runs Android 14 with the 5.15 Generic Kernel Image (GKI), and we run all workloads on the Cortex-X3 core with the frequency fixed at 2.9 GHz.

6.1 Overhead of enforcement

To understand the overhead of enforcing isolation, we run a collection of SPEC 2006 and SPEC 2017 benchmarks on Intel (Figure 3), ARM and Morello (Figure 5) CPUs. As a base line we also measure performance of a pure SFI isolation scheme similar to Google NaCl [94].

A high level observation is that all isolation schemes that rely on compiler instrumentation demonstrate high overhead. On x86 NaCl has average overhead of 17.4% respectively, whereas a combination of MPK and CET has a negligible (0.4%) overhead. On ARM, NaCl-like SFI has 18% overhead on average. Due to more reserved registers and additional software checks for stack pointer, MTE has 20% average overhead. PAC, bacasue of its complex software checks for memory operations, performs the worst, with over 100% overhead.

On Morello we execute SPEC in two modes: with and without hardware capabilities enabled. Surprisingly, enabling hardware capabilities improves performance at least relative to the regular ARM baseline. Note, however, that in absoute terms Morello hardware is significantly slower than the ARM chip used on the Google Phone. On SPEC benchmarks Morello

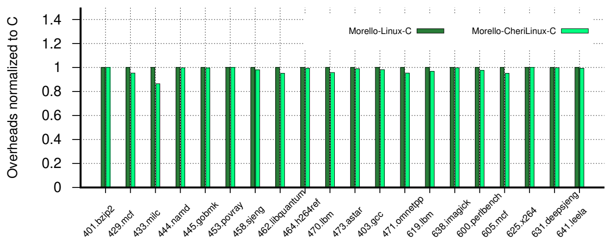

Figure 5: specCPU2017 and specCPU2006 on Morello

hardware is slower than the ARM chip from 1.1x to 2.5x, averaging at 1.4x slowdown.

Performance breakdown To understand inherent performance limitations of the current isolation approaches, we develop a version of a purely-software SFI that is conceptually similar to well-optimized modern SFI implementations [2,72]. Control over implementation allows us to enable individual isolation mechanisms to generate an accurate breakdown of performance overhead for each mechanism. We validate that our implementation performs on-par with modern SFI schemes by comparing it with state of the art WASM compiler (we omit these results for brevity, but at a high level our SFI implementation is faster than WASM).

Similar to NaCl we rely on address masking which is the fastest way of enforcing segment bounds. For example, on x86, we clear the upper bits of the 64bit register (`rax`) by introducing an idempotent operation on the 32bit part of the same register (e.g., `mov`). The regular `mov` instruction is then used to combine the base of the isolated segment (`r15` with the 32bit offset inside it.

```
; load value at [rax] to rcx
mov eax, eax ; eax contains 0-4GB
mov rcx, [r15, rax, 1] ;memory access within [r15 + 0-4GB]
```

A reserved register (`r15`) is designated to keep the base of the segment. As x86 allows complex addressing modes with arithmetics on multiple registers, we use load effective address (`lea`) instead of `mov` to compute the address and then clear its top 32 bits.

On ARM we utilize a bit-field instruction that copies a subset of bits containing the number that encodes the base of the segment (stored in a reserved `x28` register) into the address register.

```
; load value at x2 into x1, x28 contains the base address
bfi x2, x28, #32, #31 ;x2 = segment + 0~4GB
ldr x1, [x2]; load is safe
```

Instead of enforcing complete control flow integrity, we implement a lighter approach of grouping instructions into basic instruction blocks that are aligned in memory [72, 94]. This allows us to avoid allocating additional registers required to protect the stack. To ensure integrity of the bounds checks, we mask indirect control flow transitions and returns from procedures to land at the beginning of a basic instruction block.

To understand the overheads of four individual mechanisms

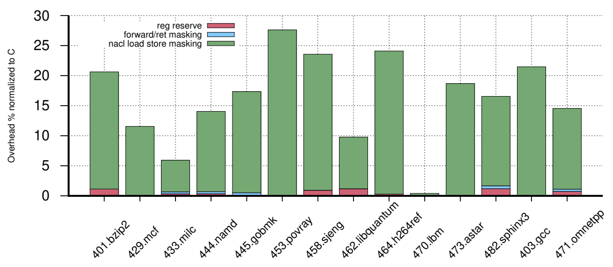

Figure 6: SPEC 2006 Performance Breakdown on x86

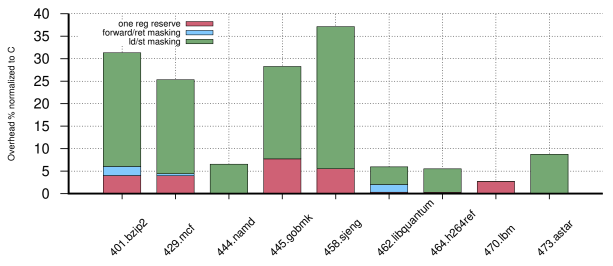

Figure 7: SPEC 2006 Performance Breakdown on ARM AArch64

– register reservation, control flow enforcement on forward and return edge, re-grouping instructions, and masking of the address itself – we selectively enable these mechanisms on a collection of SPEC 2006 benchmarks on Intel (Figure 6) and ARM (Figure 7) CPUs.

On average, software instrumentation introduces an overhead of 17.4% with the max of 28%, which is in-line with previous studies [2,43]. Most of the overhead comes from the added instructions that load the target addresses for load/store into 32bit registers. On Intel, these instructions add from 0.4% (`lbm`) to 28% (`povray`) overhead, and 2-36% on ARM.

We observe that shadow stack and indirect branch tracing have negligible overhead (less than 1% on average). Reserving one register for storing the segment has (0-2)% performance overhead one x86 architecture and less than 1% on ARM. For MTE based isolation, we reserve 2 additional registers (for replacing stack pointer and keeping the MTE tag), which increases the overhead to 2 to 5%. Instruction bundling has less than 1% overhead on both x86 and ARM. And forward/return edge address masking used in NaCl has around 1% overhead on x86 and 0.5% on ARM.

Limitations ARM PAC and ARM MTE are inherently limited by the overhead of compiler instrumentation required to enforce isolation. Our analysis shows that even the minimal instrumentation, e.g., a single instruction that masks the address or enforces an MTE tag, adds overhead prohibitive to modern workloads. Of course, this instruction is on the critical path of the pipeline as the load and store after it depends on it.

Possible solutions The limitation above allows us to identify the following design principle:

Transparency *Architectural mechanisms should avoid relying on expensive compiler instrumentation for enforcement of isolation.*

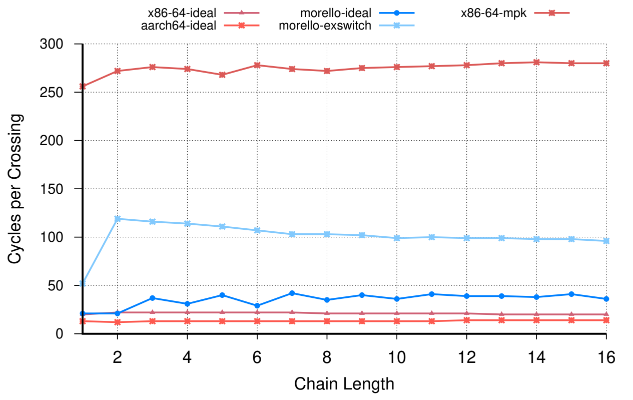

Figure 8: Overhead of cross-subsystem invocations

On x86 machines, MPK implements this design principle resulting in only minimal overhead. Alternatively a special mode of execution that enforces the bounds checks in hardware can be used to minimize the overhead of enforcing the isolation boundary. To confirm this intuition we develop an alternative isolation scheme that relies on the `gs` segment register to enforce the bounds check on every memory access (`segue`, Figure 6). By using (`gs`) instead of (`r15`) to keep the base, we 1) free a general purpose register, 2) avoid emitting an extra (`lea`) before a memory access, and 3) free an operand in memory access operation.

```
  ; load value at [rax + rbx] to rcx
  mov rcx, gs:[eax, ebx, 1] ;memory access within [gs + 0-8GB]
```

The use of (`gs`) eliminates added instructions and therefore, reduces the overhead down to 4%.

For ARM MTE, it is possible to eliminate the overhead of copying the MTE tag on each memory access is to keep the tag associated with the heap of the current subsystem in a protected register instead of each pointer. The tag from the register can be applied in the MTE check against the tag of the accessed memory region. This essentially eliminates the need for compiler instrumentation as the tag comes from a protected register instead of a pointer.

To verify the benefits of such approach, we developed a compiler pass that avoids overwriting the tag bits of the pointer on each memory access assuming that it will come from a CPU register (`MTE++`). We further assume that the register holding the tag can be accessed within the same cycle and hence introduces no visible overhead. On average SPEC benchmarks incur only 2% overhead due to control flow enforcement and reservation of one register.

6.2 Cross-subsystem Invocations

We analyze the overheads of cross-subsystem invocations for our isolation schemes on ARM and Intel machines (Figure 8). To keep the overhead in perspective and reason about both total costs of hardware boundary switching and the cost of saving and restoring the state of the thread across invocations,

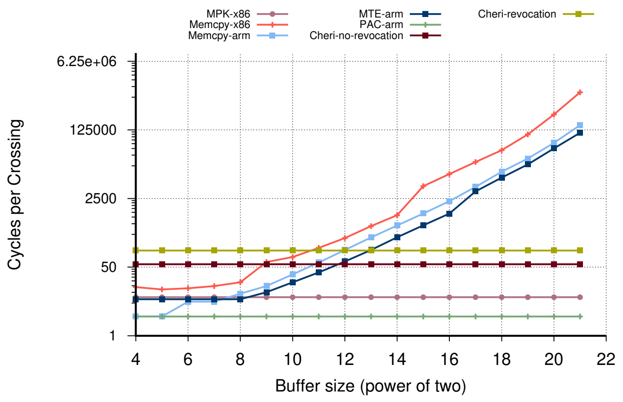

Figure 9: IPC overhead with buffers

we implement a version of the *ideal* trampoline, i.e., a cross-subsystem invocation primitive that assumes a one cycle cost of changing the isolation boundary. Specifically, instead of accessing the hardware, e.g., updating `pkru` register on Intel or changing the RDDC capability on Morello, we just invoke a `nop` instruction (`c-ideal`).

To understand the overhead of saving and restoring extended registers, in all configurations we evaluate two setups: with (`hw-fxsave`) and without (`hw-no-fxsave`) saving extended registers.

Our experiments measure invocation overheads on a chain of cross-subsystem invocations. We vary the length of the chain from 1 to 16. Each invocation simply invokes the next subsystem in a chain and then returns. In all experiments, we measure the total time to execute ten million iterations. An ideal implementation needs 20-22 cycles to perform a null cross-subsystem invocation on Intel, 12-14 cycles on Pixel 8, and about 40 cycles on the Morello Development Platform. On Morello, relying on hardware support to switch to Executive mode turns out to be faster than deriving capabilities from the unsealed PCC at a difference of about 100 cycles.

Limitations Overhead of cross-subsystem invocations is limited by both the cost of changing the hardware isolation boundary and by the cost of saving and restoring general and extended registers.

Possible improvements Arguably, additional optimizations are possible to improve the overhead of saving general and extended registers as well as optimizing implementation of crossing the isolation boundary. Save and restore of extended registers contributes the most significant fraction of the cross-subsystem invocation and arguably can be optimized in hardware.

6.3 Zero-Copy

To understand the benefits of zero copy, we analyze the overheads of passing data in cross-subsystem invocations by varying the size of the object in the incrementing powers of two from 4 to 22. We benchmark several IPC variants: 1) retagging memory through changing an MPK key (`MPK-x86`); 2) memory copying for NaCl-like SFI scheme on Intel and ARM (`memcpy-x86` and `memcpy-arm`); 3) MTE retagging on with the `stg` instructions at user and kernel privilege levels (`stg-user-arm` and `stg-kernel-arm`); and 4) pointer re-signing for ARM PAC SFI scheme (`PAC-arm`); 5) passing a capability on Morello CHERI (`cheri`). Not surprisingly passing a CHERI capability and writing MPK register show the lowest overhead, followed by a lightweight PAC pointer re-signing, and relatively fast hash table update for SFI. These operations do not depend on size of the passed object. Surprisingly, MTE retagging is only slightly faster than memory copy.

Limitations Main limitation of the MPK scheme is the inability to reflect passing of zero-copied memory regions across all cores of the system. During the invocation, the `pkru` register is updated on one core to reflect the change in access rights between the caller and callee subsystem. This change however is local to the core. Updates of `pkru` registers on other core require an expensive cross-core synchronization similar to a TLB shootdown, i.e., a traditional inter-processor interrupt (IPI) or a similar synchronization scheme.

This limitation leads us to the following design principle:

> **Core-coherent synchronization of rights** *Hardware should support synchronization of access permissions across cores of the system.*

Hardware support is essential for implementing both a general programming model in which memory regions exchanged across isolation boundary are available to all threads and global revocation of rights (which we discuss below).

Furthermore, tag-based schemes like MPK and MTE suffer from a limitation on the number of isolated subsystems (and in case of MPK exchanged buffers as in order to support zero-copy passing of buffers each buffer requires a separate tag).

Finally, MTE suffers from the overhead of retagging which in our experiments is only marginally faster than copying.

Possible solutions Implementation of core-coherent zero-copy passing of memory regions for MPK might be possible through a combination of an instruction set extension which provides controlled access to the tag bits in the page table and support for core-coherent TLBs. Controlled access to the page table bits would allow tag updates in a manner similar to MTE. We can then assign each isolated subsystem a unique tag and update the tag the memory region to pass it from caller to callee in a cross-subsystem invocation. Note, it is possible to implement a similar scheme in software by mapping the pages of the page table inside the area accessible by TCB, but the risks are high since a compromise of the TCB provides attacker with unrestricted access to the page table and hence a system wide control of memory. Implementation of cross-core coherent updates, however, would require hardware support for coherent TLBs – an update of the tag should be immediately reflected on all the cores, hence invalidating stale TLB entries that might contain the old tag (we discuss this below in Section 6.4).

Increasing the number of supported tags to a practically large number is challenging. For example, in case of MPK the tag occupies unused bits in the page table entry of each page. Increasing the number of tags will require changes to the page table organization, e.g., entries and overall layout of the page table.

In case of MTE the tag is limited by the number of unused bits in the pointer. At the moment MTE is using only 4 bits of the top unused byte, so theoretically the use of all 8 bits can increase the number of tags to 256. Since MTE does not waste a tag for each zero-copied buffer, in practice 256 tags can be sufficient for isolation of typical applications, e.g., network functions, device drivers in the kernel, etc. Another solution is to keep the tag in a special tag register instead of the pointer itself (as we discussed above this also can eliminate the overhead of compiler instrumentation required to enforce the tag on each pointer).

To address the overhead of retagging, it is possible to implement support for variable granularity of tag enforcement. Smaller tags would allow for finer granularity of isolation, while lager tags can support faster passing of data across subsystems. It is possible to support different tag sizes in a single address space by keeping the tag size information in the page table.

6.4 Revocation

To understand the overhead of implementing revocation for the Morello architecture we measure the overhead of first passing the capability inside an isolated subsystem and then implementing compiler instrumentation that btoh tracks capability stores in memory as well as capability invalidation on passing to another subsystem (Figure 9, `cheri-revocation`). We implement an efficient hash table that tracks memory addresses where each capability is saved and invalidate all capabilities that are moved into another subsystem. The overhead of revoking one capability reaches 120 cycles.

Limitations Capability schemes like CHERI are inherently limited by the lack of centralized metadata that provides support for revocation. Unrestricted propagation of capabilities requires additional software mechanisms to either scan private heap of the subsystem for capabilities that need to be revoked [87, 90] or instrumentation that tracks memory locations where capabilities are saved (the approach we suggest in this paper). Both techniques incur significant overheads.

An additional challenge is tracking and revoking rights to complex recursive data structures, e.g., linked lists and even arrays of pointers. A revocation of the root capability requires traversal of the data structure and revocation of all leaves. Due to significant overheads of recursive revocation our work makes a tradeoff and limits the expressiveness of data structures passed across subsystems, i.e., to avoid recursive revocation we allow only simple plain old data structures to be exchanged on the heap (e.g., data buffers).

Hardware architectures that keep access rights in registers, e.g., Intel MPK and CHERI, are facing another inherent limitation: it is impossible to perform revocation of rights across the cores. For example, in a capability system like CHERI it is possible to share capabilities across cores by saving them on one core and loading them in registers of another core. This significantly complicates revocation of a capability, since the core that revokes a capability needs to ensure that no instances of the same capability exist in registers of other cores. Similar to TLB invalidation, such check requires an expensive cross-core synchronization mechanism, e.g., an inter-processor interrupt (IPI).

A similar problem exists on Intel MPK. In contrast to CHERI, there is no explicit way to pass access permissions between the cores. MPK tag registers that reflect permissions to access memory are per-core, and hence in contrast to CHERI, not the revocation, but the zero-copy passing of rights requires either a cross-core synchronization or a restricted view of memory in which memory regions exchanged in cross-subsystem invocations are accessible on a single core. Revocation becomes simple, but synchronizing access permissions between the cores requires support from software, e.g., an IPI.

ARM PAC suffers from a time-of-check time-of-use attack

Possible solutions Conceptually, revocation relies on two architectural mechanisms: 1) a metadata region that allows invalidation of all references and 2) support for coherent synchronization across the cores.

> **Revocation** *Hardware must support revocation as a first-class citizen.*

At the moment, MTE is the only solution that leverages a centralized in-memory metadata region and hence implements support for revocation. A revocation of all pointers is possible by simply updating the access bits in the MTE metadata table, and the update is synchronized across all CPUs.

An MPK scheme can potentially be extended with support for revocation but would require support for core-coherent TLBs. Hardware architects explored support for coherent TLBs in the past [8, 17, 51, 70, 74, 80, 84, 92] which arguably can be brought to modern commodity CPUs. Controlled and secure modification of the tag bits stored inside the page table can be provided with additional extensions that allow access to the bits themselves but not to the rest of the page table entry.

Implementing revocation for CHERI capabilities is arguably most challenging due to distributed nature and unrestricted flow of capabilities. A classical object capability approach is to revoke access rights with proxies [59], i.e., the capability grants access to a proxy object which is controlled by the granting authority and can stop functioning effectively revoking the access right. A similar approach is possible in hardware by introducing a "proxy" capability that serves as a distributed metadata associated with a memory region. In such approach, instead of pointing to the memory region, a

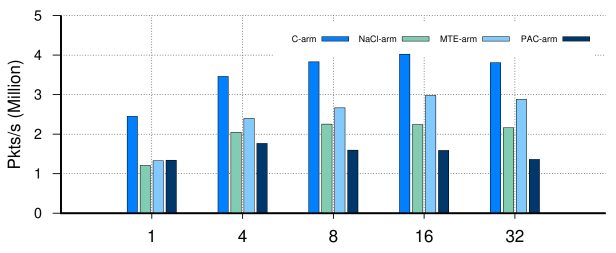

Figure 10: NF overheads on varying batch sizes AArch64

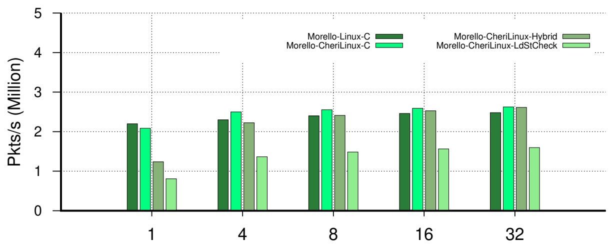

Figure 11: NF overheads on varying batch sizes Morello Board

regular CHERI capability is pointing to a proxy capability which in turn allows access to memory. If proxy capabilities are restricted from being loaded in registers, and stay in memory all the time, they can be revoked in a coherent manner across the cores.

6.5 End-to-End Application Use-Cases

Network function virtualization To understand the impact of cross-subsystem invocations on real-world applications, we implement a network function virtualization framework similar to Netbricks [65]. Today, a wide range of *network functions* (NFs) handle the most complex network tasks such as intrusion detection, packet filtering, load balancing, etc. NFs are typically deployed as a part of a *service chain* that together processes a stream of packets.

In a modern network, NFs are often built as independent software by third-party vendors and have a set of unique requirements centered around performance, isolation, and reliability. NFs often have conflicting reliability and security goals and require isolation [53, 55, 77, 93, 95]. Isolation of NFs remains a challenging problem due to stringent performance requirements of packet processing applications [6, 36, 41, 55, 69, 73]. Traditional mechanisms that can enforce isolation boundaries — hardware primitives, software fault isolation (SFI), and language safety — impose overheads that are too high for systems that execute at line rate.

All versions operate on a batch of packets (we form the batch using the C DPDK functions which are a trusted part of the system, and hence require no isolation). We implement four network functions: (1) **TTL** which decrements the time-to-live field in a packet's IPv4 header, (2) **NAT** which rewrites the source IP and port of a packet according to a mapping, (3) **ACL Firewall** which allows or drops a packet based on a list of pre-defined rules, and (4) **Maglev** which is a load balancer developed by Google to evenly distribute incoming client flows among a set of backend servers [24]. For each

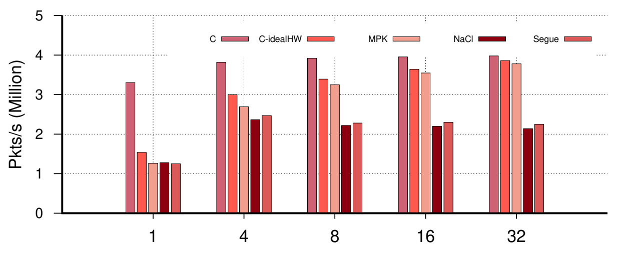

Figure 12: NF overheads on varying batch sizes x86

new flow, Maglev selects one of the available backends by performing a lookup in a hash table, the size of which is proportional to the number of backend servers (65,537 in our experiments). Consistent hashing allows even distribution of flows across all servers. Maglev then records the chosen backend in a hash table, a *flow tracking table*, that is used to redirect packets from the same flow to the same backend server. The size of the flow tracking table is proportional to the number of flows (we choose 1 M flows for our experiments). Processing a packet requires a lookup in the flow tracking table if it is an existing flow, or a lookup of a backend server and an insertion into the flow tracking table to record the new flow.

On small batch sizes, the cost of MPK-based isolation impacts the performance of the network function chain (Figure 12, Figure 10), resulting in 45% less throughput on average (batch sizes 1$\tilde{8}$). On larger batch sizes an ideal zero-cost primitive and MPK come close to the performance of non-isolated code, at an average of only 8% of overhead. All SFI schemes remain slow. NaCl-style isolation on both X86 and ARM results in an average overhead of around 50%.

Video processing We implement a video processing pipeline that extracts frames from an input video to produce an animated thumbnail (GIF), as an approximation of a workload commonly found on serverless platforms [52]. The pipeline is split into two compartments: (1) **Frame Extractor** which decodes the input video into raw frames, extracting one frame for every 100 frames, and (2) **Thumbnail Encoder** which encodes the extracted frames into an animated GIF file.

We implement the pipeline in C using FFmpeg, and apply five different isolation mechanisms: NaCl, Segue, MPK, MTE, and PAC (Figure 13, Figure 14). With Intel MPK and ARM MTE, the extracted frames are retagged and passed between the compartments. However, with NaCl, they are copied to the second compartment, resulting in an extra overhead.

MPK comes close to the performance of non-isolated C. SFI schemes suffer from overheads of inserted memory checks and reserved registers. On Arm, NaCl-like segment isolation has average of 17-20% overhead for both Frame Extractor and Thumbnail Encoder. Similar to SPEC benchmarks, overhead of MTE is slightly higher than NaCl and is around 20%. PAC has the worst performance, with over 100% overhead on both compartments. On x86, MPK performs extremely close to non-isolated C with less than 0.3% overhead. NaCl-like isolation however, suffers high overhead (35%) on

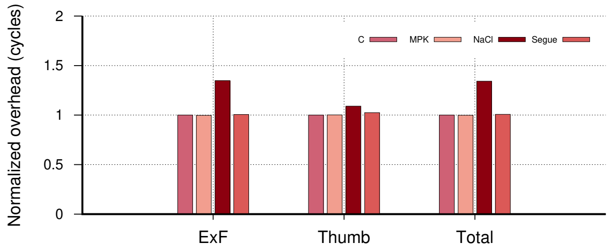

Figure 13: Overhead of FFmpeg (X86)

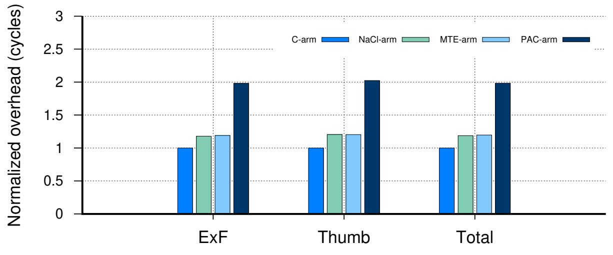

Figure 14: Overhead of FFmpeg (ARM)

Frame Extractor and 10% overhead on Thumbnail Encoder.

For video processing, buffer passing takes less than 0.1% of the total execution time for all the tests. Despite that PAC and MPK have near-zeor overhead on buffer passing, buffer passing has no significant impact on the performance of this workload.

7 Conclusions

After decades of relatively slow adoption, we finally see a renewed interest in hardware isolation mechanisms. We study the overheads of recent mechanisms aimed at support of low-overhead isolatioon and derive a set of practical design principles which we argue are critical for achieving a near zero-cost isolation and communication overhead. Our analysis identifies a number of inherent bottlenecks in recent hardware mechanisms that can be hopefully useful for development of the future hardware mechanisms.

References

[1] seL4 Performance. https://sel4.systems/About/Performance/.

[2] WebAssembly Specification. https://webassembly.github.io/spec/core/.

[3] OpenArena Live. https://openarena.live, 2019.

[4] Akamai. Serverless Computing with Akamai Edge Workers. https://www.akamai.com/products/serverless-computing-edgeworkers, 2015.

[5] Alexander Gallego. Redpanda Wasm engine architecture. https://redpanda.com/blog/wasm-architecture, 2021.

[6] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. XOMB: Extensible open middleboxes with commodity servers. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS'12, pages 49–60, New York, NY, USA, 2012.

[7] Arm. Armv8.5-A Memory Tagging Extension white paper. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[8] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding tlb shootdowns through self-invalidating tlb entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 273–287, 2017.

[9] Godmar Back and Wilson C Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[10] Steve Bannister. Memory Tagging extension: Enhancing memory safety through architecture, August 2019. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/enhancing-memory-safety.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP'09, pages 29–44. ACM, 2009.

[12] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, Mitre Corporation, 1976.

[13] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 267–283, 1995.

[14] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 1–19, 2020.

[15] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP'09, pages 45–58. ACM, 2009.

[16] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Comput. Syst.*, 6(1):28–50, February 1988.

[17] Moon-Seek Chang and Kern Koh. Lazy tlb consistency for large-scale multiprocessors. In *Proceedings of IEEE International Symposium on Parallel Algorithms Architecture Synthesis*, pages 308–315, 1997.

[18] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[19] Intel Corporation. DPDK: Data Plane Development Kit. `http://dpdk.org/`.

[20] Intel Corporation. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. `https://kib.kiev.ua/x86docs/Intel/ISAFuture/319433-034.pdf`, 2018.

[21] W. de Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, pages 136–140, 2008.

[22] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. XPC: Architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA'19, pages 671–684, New York, NY, USA, 2019.

[23] Dylan Schiemann. Zoom on Web: WebAssembly SIMD, WebTransport, and WebCodecs. `https://www.infoq.com/news/2020/08/zoom-web-chrome-apis`, 2020.

[24] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, pages 523–535, March 2016.

[25] Kevin Elphinstone and Gernot Heiser. From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 133–150, 2013.

[26] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. The Case for Writing Network Drivers in High-Level Programming Languages. In *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–13. IEEE, 2019.

[27] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 75–88, 2006.

[28] Evan Wallace. WebAssembly cut Figma's load time by 3x. `https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/`, 2017.

[29] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal Memory Safety via Robust Points-to Authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[30] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[31] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference (WTEC '94)*, pages 97–114, 1994.

[32] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J Elphinstone, Volkmar Uhlig, Jonathon E Tidswell, Luke Deller, and Lars Reuther. The SawMill multiserver approach. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 109–114. ACM, 2000.

[33] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff. The arm morello evaluation platform—validating cheri-based security in a high-performance system. *IEEE Micro*, 43(3):50–57, 2023.

[34] GSMArena. Google Pixel 8's Tensor G3 GPU tests show weak performance but decent efficiency. https://www.gsmarena.com/google_pixel_8s_tensor_g3_gpu_tests_show_weak_performance_but_decent_efficiency-news-60199.php, 2023.

[35] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 489–504, July 2019.

[36] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. MSwitch: A Highly-Scalable, Modular Software Switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR'15, New York, NY, USA, 2015.

[37] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 255–266, San Jose, CA, June 2013.

[38] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating Device Driver Isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.

[39] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, apr 2007.

[40] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[41] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 445–458, Seattle, WA, April 2014.

[42] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2020. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html.

[43] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX Annual Technical Conference*, pages 107–120, 2019.

[44] Jordon Mears. How we're bringing Google Earth to the web. https://web.dev/earth-webassembly/, 2019.

[45] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers, 2018.

[46] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[47] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 627–643, Carlsbad, CA, October 2018.

[48] Yuan Li, Wende Tan, Zhizheng Lv, Songtao Yang, Mathias Payer, Ying Liu, and Chao Zhang. PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1901–1915, 2022.

[49] Zhaofeng Li, Tianjiao Huang, Vikram Narayanan, and Anton Burtsev. Understanding the overheads of hardware and language-based ipc mechanisms. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*, 2021.

[50] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. PACStack: an Authenticated Call Stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374, 2021.

[51] Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. Ecotlb: Eventually consistent tlbs. *ACM Trans. Archit. Code Optim.*, 17(4), sep 2020.

[52] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, OSDI'22, pages 303–320, 2022.

[53] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 218–233, 2017.

[54] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-Principal Modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, page 115–128, 2011.

[55] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 459–473, Seattle, WA, April 2014.

[56] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.

[57] McKee, Derrick and Giannaris, Yianni and Perez, Carolina Ortega and Shrobe, Howard and Payer, Mathias and Okhravi, Hamed and Burow, Nathan. Preventing Kernel Hacks with HAKC. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[58] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, 2019.

[59] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006.

[60] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 699–716, 2020.

[61] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association.

[62] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 21–39, November 2020.

[63] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20)*, pages 157–171, 2020.

[64] Nathan Froyd. Securing Firefox with WebAssembly. `https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly`.

[65] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 203–216, Savannah, GA, November 2016.

[66] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, July 2019.

[67] Pat Hickey. Lucet Takes WebAssembly Beyond the Browser. `https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.`, 2019.

[68] Mathias Payer and Thomas Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual execution Environments (VEE '11)*, pages 157–168, 2011.

[69] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-Switch: A Scalable Software Virtual Switching Architecture. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 13–24, San Jose, CA, June 2013.

[70] Bogdan Romanescu, Alvin Lebeck, Daniel Sorin, and Alecia Bracy. Unified instruction/translation/data (unitd) coherence: One protocol to rule them all. pages 1–12, 01 2010.

[71] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[72] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*, pages 1–11, 2010.

[73] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pages 323–336, San Jose, CA, April 2012.

[74] Byeong Seong, Donggook Kim, Yangwoo Roh, Kyu Park, and Daeyeon Park. Tlb update-hint: A scalable tlb consistency algorithm for cache-coherent non-uniform memory access multiprocessors. *IEICE Transactions*, 87-D:1682–1692, 07 2004.

[75] Christopher Small and Margo I. Seltzer. VINO: An Integrated Platform for Operating System and Database Research. Technical Report TR 30-94, Harvard University, Division of Engineering and Applied Sciences, 1994.

[76] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010.

[77] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. In-Net: In-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015.

[78] Mark Sullivan and Michael Stonebraker. Using Write Protected Data Structures To Improve Software Fault Tolerance in Highly Available Database Management Systems. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB'91, pages 171–180, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[79] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[80] Patricia Teller, R. Kenner, and Marc Snir. Tlb consistency on highly-parallel shared-memory multiprocessors. pages 184 – 193, 02 1988.

[81] The Istio Project. WebAssembly in the Istio Proxy (Envoy). `https://istio.io/latest/docs/concepts/wasm/`.

[82] 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 Security Symposium (USENIX Security '19)*, pages 1221–1238, August 2019.

[83] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero. CODOMs: Protecting Software with Code-centric Memory Domains. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480, June 2014.

[84] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, page 340–349, USA, 2011. IEEE Computer Society.

[85] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-Based Operating System for Java. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393. 1999.

[86] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216. ACM, 1993.

[87] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, 2020.

[88] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 31–44, 2005.

[89] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[90] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G Neumann, Simon W Moore, Robert NM Watson, et al. CHERIvoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 545–557, 2019.

[91] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-Process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2989–3002, New York, NY, USA, 2022. Association for Computing Machinery.

[92] Zi Yan, Ján Veselý, Guilherme Cox, and Abhishek Bhattacharjee. Hardware translation coherence for virtualized systems. *SIGARCH Comput. Archit. News*, 45(2):430–443, jun 2017.

[93] Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. HyperNF: Building a high performance, high utilization and fair NFV platform. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 157–169, New York, NY, USA, 2017.

[94] Bennet Yee et al. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *SSP*, 2009.

[95] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless fine-grained NFs for flexible per-flow customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 3–17, New York, NY, USA, 2016.

[96] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: Fully Verified Software Fault Isolation. In *11th Intl. Conf. on Embedded Software*. ACM, 2011.

[97] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *21st ACM Conference on Computer and Communications Security (CCS)*, pages 558–569, 2014.